Alma Mater Studiorum
Università di Bologna

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

Tecnologie e framework per la programmazione multi-agente

Relatore e Co-relatore:

Chiar.mo Prof.
Andrea Omicini

Dott. Ing.
Stefano Mariani

Presentata da:

Roberto Giuliani

Sessione III
Anno Accademico 2015/2016

Parole chiave:

Agent
Multi-Agent System MAS
Agent-Oriented Software Engineering
Agent Framework

Alla mia famiglia e
ai miei cari

Indice

Introduzione

Negli ultimi anni le tecnologie informatiche sono state centro di uno sviluppo esponenziale!
L'informatica viene considerata come la scienza che si occupa dell'incremento e del perfezionamento delle tecnologie, le quali sono in qualche modo condizionate dalle innovazioni di tale disciplina.
Fra gli incalcolabili nuovi trend che si sono affacciati negli ultimi anni sul panorama informatico, il paradigma per la programmazione ad agenti è uno dei più interessanti, in accordo con i recenti e prossimi sviluppi delle tecniche in generale.
Tale paradigma permette di concepire entità sofware attive, in grado di presentare un comportamento autonomo e pro-attivo; tutti concetti che portano ad una marcata analogia con il comportamento umano.
I sistemi software convenzionali erano, e sono tuttora, sviluppati prevalentemente per mondi statici in cui il contenuto informativo è certo e completo, mentre i sistemi moderni devono fronteggiare risorse limitate in contesti variabili secondo modalità non prevedibili e per i quali si dispone di conoscenze locali, dunque parziali.
Per questi motivi la ricerca iniziò a rivolgersi verso lo studio dell'interazione fra sistemi e della soluzione dei problemi in una prospettiva più sociale: i primi passi verso la nascita di un nuovo paradigma di programmazione.
Come si vedrà infatti nei capitoli seguenti esiste un profondo legame tra i meccanismi di ragionamento dell'essere umano e la definizione/caratteristiche degli agenti.
Questo elaborato tratterà tali argomenti partendo da un punto di vista generico ed introduttivo, volutamente esaustivo, per portare il lettore ad una comprensione ampia e soddisfacente su questo ambito di studi, per poi concentrarsi sullo stato dell'arte di tutti i framework e sugli ambienti di sviluppo attualmente in uso. Inoltre presenterà un insieme di caratteristiche e differenze che li lega tra loro.
Si giungerà infine alla conclusione che gli agenti e il loro paradigma costituiscono l'avvenire, concepito però come evoluzione, non come rivoluzione!

Capitolo 1

IL CONCETTO DI AUTONOMIA

Nell'ambito dell'informatica non è ancora stata sviluppata una precisa e coerente definizione di autonomia, per questo motivo, per il momento, si prende spunto da altri settori, come:
definizione del termine ricavata dal dizionario:

derivante dal Greco AUTONOMOS,
perseguire le proprie leggi; [23]

oppure:

il diritto e la condizione di autogoverno, l'indipendenza, la libertà; [23]

lo stato di essere liberi dal controllo o dal potere di qualcun altro; [23]

Da tali frasi riusciamo a comprendere il principale significato del concetto di autonomia, come la capacità di un agente di agire secondo la sua moralità senza essere influenzato dai desideri altrui, in grado perciò di fare le proprie scelte e prendere le proprie decisioni.
Questo concetto è connesso con i valori di **libertà** e di **scelta**.
Una caratteristica da sottolineare, grazie all'abilità del ragionamento, riguarda l'indipendenza da sé stessi, con la quale si ha la possibilità non solo di decidere, ma anche di cambiare la propria idea nello scorrere del tempo e nel momento in cui un cambiamento possa avvenire, per esempio nell'ambiente, aspetto analizzato più approfonditamente nei capitoli a seguire.

Un'ulteriore ed essenziale prospettiva da citare riguarda l'autonomia nei sistemi artificiali complessi. L'autonomia trova differenti significati in molteplici ambiti e questo è uno dei più interessanti.
I sistemi naturali esibiscono caratteristiche che i programmatori cercano da sempre di comprendere, catturare e portare a livello computazionale. A tale scopo nacque nel 2006, grazie alla collaborazione di *Lui and Tsui*, il *Nature-Ispired Computing*, NIC, dove viene messa in evidenza l'indipendenza dei componenti e l'organizzazione dei sistemi.
La coordinazione viene presentata come la chiave dei possibili problemi nella realizzazione di questi sistemi, infatti molti di essi, nel mondo naturale, sembrano essere governati da semplici ma potenti meccanismi di *coordination*. Introducendo questi modelli come il nucleo di sistemi più complessi da quando sono stati concepiti per affrontare le interazioni.
Uno dei principali problemi riguarda l'ambiente esterno, argomento trattato dettagliatamente nei capitoli a venire. Esso corrisponde ad un mediatore tra i componenti, che permette, indirettamente, la comunicazione e la coordinazione. Inoltre possiede una sua struttura, richiedendo nozioni di locazione e garantendo ai componenti possibilità di muoversi nel suo spazio.
Un secondo problema si concentra sui comportamenti stocastici, definiti meglio con il termine inglese *Stochastic behaviour*. I sistemi complessi vengono considerati con modelli probabilistici, questo è dovuto al fatto che i meccanismi di non determinismo del suo funzionamento possono prendere il sopravvento sul sistema stesso. Dato che non è possibile rappresentare tutte le proprietà, questi meccanismi cercano di catturare almeno tutte le dinamiche della coordinazione. [23]

Capitolo 2

L'AGENTE SOFTWARE

2.1 Definizioni

L'approccio comune nel trattare argomenti tecnici come questo è quello di iniziare con una o più definizioni, in modo tale da fissare dei concetti cardine nel lettore e muoversi successivamente, durante approfondimenti e deduzioni, consapevoli che chi si dedica alla lettura possa comprendere tali ragionamenti.
Tra le numerose definizioni presenti nell'informatica troviamo particolarmente interessanti quelle di *Nicholas R. Jennigs* e *Michael Wooldridge*.
Essi definiscono:

> *Un agente come un sistema informatico situato in un certo ambiente, capace di azioni autonome e flessibili al fine di ottenere i suoi obiettivi di progetto.* [35]

L'agente viene pertanto descritto come un'entità il cui funzionamento è continuo ed autonomo, da qui la spiegazione precedente del concetto di autonomia, in grado di operare per conto di altri utenti o agenti, ma anche in maniera indipendente da essi.
L'agente è situato in un ambiente e con esso deve necessariamente interagire, in particolar modo mostrando capacità di comunicazione e cooperazione con gli altri agenti eventualmente presenti nel sistema.
L'ambiente in cui è immerso potrebbe essere di natura fisica oppure simulata, implementata quindi via software.
Nel primo caso l'agente sarà reale, fisico, come anche i sensori e gli attuatori che lo compongono; nel secondo invece, essendo l'ambiente una realtà

simulata, i sensori e gli attuatori possono essere anch'essi entità software, in grado di interagire con l'ambiente di supporto.

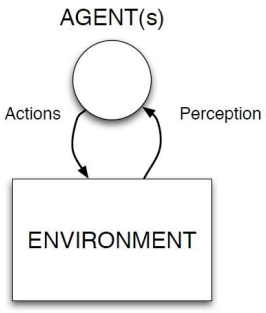

Figura 2.1: Agent Software

Michael Wooldridge aggiunge:

un agente è un'entità che agisce attivamente, modificando l'ambiente piuttosto che subirlo passivamente. Opera in modo autonomo, senza l'intervento diretto dell'uomo, cercando di raggiungere il suo obiettivo massimizzando una qualche funzione valutativa. È un sistema di alto livello che sembra soggetto a desideri, giudizi o a qualche altra funzione di tipo cognitivo. È un sistema intenzionale. Più correttamente si tratta di un sistema che, data la sua complessità, è conveniente descrivere tramite gli Stati Intenzionali.

Anche *J.Horberg* basa la sua definizione di agente sul concetto di autonomia ma introduce un nuovo elemento, la mobilità:

L'agente è un'entità mobile, che viaggia attraverso la rete ed è in grado di duplicarsi su altri computer per eseguire il compito assegnatogli. [14]

Il termine **Sistemi Intenzionali** è stato inciso dal filosofo *Daniel Dennett* per descrivere:

Entità il cui comportamento può essere previsto attraverso il metodo dell'assegnazione di credenze, desideri e capacità razionali. [10]

A fine paragrafo faremo un riferimento a tale modello, definito come modello BDI.

2.2 Conclusioni

In conclusione:

> Un agente può essere visto come un'entità software responsabile per l'esecuzione di un determinato compito, contenente un certo livello di *intelligenza* e operante per conto di un utente o di uno/più agenti.

2.3 Caratteristiche

Jennings sottolinea inoltre come sia possibile suddividere il termine agente in un significato debole, o Weak; espressione impiegata per indicare un software che possiede le seguenti capacità:

- *Autonomia:* capacità di operare senza l'intervento umano.

- *Capacità sociale:* capacità di interagire con altri Agenti o possibili umani, tramite alcuni tipi di agent-comunication: Speech-Acts, Artifact based Interactions, Signals, Environments traces. Il concetto di Coordinazione è un aspetto sociale, di cooperazione, di collaborazione, ma anche di competizione con altri.

- *Reattività:* capacità di reagire a stimoli esterni. Le applicazioni nei domini del mondo reale sono caratterizzate da alte condizioni dinamiche, cambiamenti di situazioni, informazioni incomplete, risorse scarse. Un sistema reattivo è un sistema in grado di rispondere a questi cambiamenti in *tempo* per essere considerati utili.
  Purely reactive agents - decidono cosa eseguire senza far riferimento agli avvenimenti passati, alla storia passata. Cioè le reazioni sono puramente basate sul presente, senza alcun confronto con gli stati passati.

- *Iniziativa:* definita anche pro-activeness, capacità di prendere l'iniziativa. Far verificare qualcosa, piuttosto che aspettare che accada.

Ed un significato forte, o Stronger, dove agli agenti vengono assegnati attributi tipicamente associati all'attività mentale umana quali:

- *Intenzioni:* il volgersi della volontà verso un determinato fine.

- *Conoscenza:* il complesso di nozioni che possiede.

- *Desideri:* ottenere qualcosa.

- *Obblighi:* dovere imposto.

- *Mobilità:* è la capacità di un agente di spostarsi all'interno dell'ambiente in cui opera, per poter raggiungere le risorse di cui necessita o le informazioni di cui è alla ricerca. Per gli agenti fisici il movimento avviene all'interno di un ambiente reale, mentre per gli agenti software si intende il trasferimento da un nodo all'altro di una rete di terminali. La mobilità rappresenta un ulteriore aspetto di autonomia, in quanto la scelta del percorso da seguire o dove fermarsi sono decisioni lasciate al giudizio dell'agente.

- *Cooperazione:* si ha quando più agenti sono impegnati nel conseguimento di un obiettivo comune, questo implica sia aspetti di comunicazione che di coordinamento, necessari per organizzare i partecipanti all'operazione seguendo una certa strategia, e per permettere lo scambio dei risultati parziali raggiunti.

- *Apprendimento:* si tratta della capacità di fare tesoro delle esperienze passate, per servirsene al momento di dover prendere una determinata decisione.

- *Adattamento:* indica la proprietà dell'agente di modificare il proprio comportamento in seguito alle condizioni dell'ambiente circostante. L'agente in questo caso si dice *adattativo* se è in grado di cambiare autonomamente il proprio comportamento per adeguarsi alle situazioni in cui si viene a trovare.

- *Sincerità:* incapacità di comunicare intenzionalmente false informazioni, questo perché l'agente non è in grado di scegliere tra menzogna e verità.

- *Benevolenza:* un agente non può avere obiettivi da raggiungere in contrasto con quelli di altri agenti.

- *Razionalità:* questa astrazione si basa sul concetto che le azioni compiute da un agente siano volte al perseguimento di determinati obiettivi e che, in nessun caso, cerchino di ostacolarne il raggiungimento.

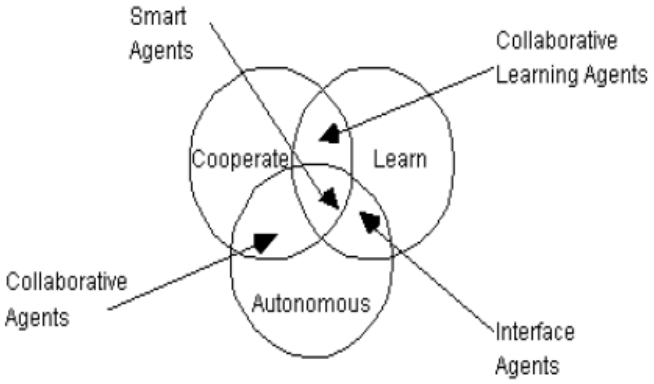

Figura 2.2: Agent Software: Tipologie

Un agente non deve necessariamente possedere tutte le proprietà citate, sta al programmatore decidere, in base agli obiettivi da conseguire, quali progettare.
Alcune, come l'adattamento e l'apprendimento, risultano correlate tra loro e pertanto può risultare difficoltoso anche solo distinguerle.

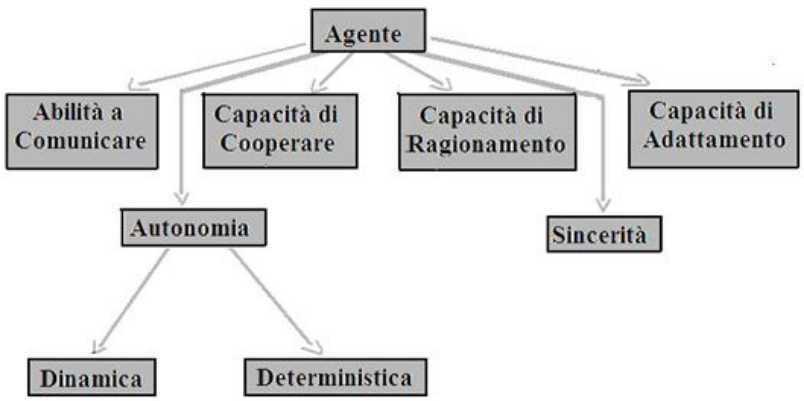

Figura 2.3: Agent Software: Caratteristiche

2.4 I vari tipi di Agente Software

Perciò, a seconda delle loro caratteristiche architetturali, gli agenti possono essere divisi in cinque differenti macro-categorie:

Deductive reasoning agents

Caratteristica di tali agenti è la modellazione e rappresentazione simbolica per la realizzazione dei sistemi intelligenti e dell'ambiente. Compito del progettista è, quindi, quello di tradurre il mondo reale in una esatta rappresentazione simbolica formale e far sì che gli agenti siano in grado di comprenderla ed operarvi al suo interno, tramite la definizione di veri e propri teoremi i quali l'agente utilizzerà per effettuare le dovute deduzioni logiche.
Le azioni dell'agente sono quindi governate da ragionamenti logici, in grado di ipotizzare le condizioni di verità dei simboli che rappresentano l'ambiente.

Practical reasoning agents

Deducibile direttamente dal nome, gli agenti *practical reasoning* hanno la proprietà del vero e proprio ragionamento pratico, che consiste in una fase di deliberazione, detta *deliberation* (ovvero i goals, gli obiettivi dell'agente che deve perseguire) e da una fase più pratica detta *means-end reasoning*, ovvero lo svolgimento di tali intenzioni (i cosiddetti plans o piani d'azione) che permettono di portare a compimento gli obiettivi precedenti.
Tali agenti cercano di rappresentare, in maniera più soddisfacente possibile, le abitudini dell'essere umano. Il quale non pratica solo ragionamenti logici ma, li porta a proprio frutto, per risolvere problemi o migliorare condizioni.

Reactive agents

L'esigenza negli ultimi anni di avere agenti in grado di muoversi in ambienti strettamente legati al tempo ha portato allo sviluppo e realizzazione di questa differente sezione. In generale, gli Agenti Reattivi portano comportamenti quali *situatedness* e *reactivity*. Dove implica che un agente è strettamente legato all'ambiente in cui si trova e reattivo ai cambiamenti che si presentano nell'arco del tempo.

Hybrids agents

Sezione a parte quella riguardante gli Agenti Ibridi, nella quale si cerca in principal modo di dividere in strati specifici, definiti *layer*, gli aspetti attivi e proattivi del loro comportamento.
E, come implica il nome, rappresentato tutti gli agenti che contengono ca-

ratteristiche di diverse tipologie, comuni a quelle citate in precedenza. In questa categoria rientrano anche gli agent i quali non è possibile classificare altrimenti.

Mobile agents
Gli Agenti Mobili sono coloro in grado di muoversi attraverso le reti di calcolatori, andando ad interagire con differenti Host, ricercando e raccogliendo informazioni per conto di altri agenti o trasmettendo direttamente loro stessi. L'idea di base è che tali agenti siano indirizzabili nella rete e che possano riprendere la propria computazione in nodi differenti, trasferendo programma e stato.

2.5 L'ambiente

Una volta progettato un agente esso viene collocato in un ambiente con il quale può scambiare informazioni e sul quale può eseguire azioni. Occorre quindi definire il termine ambiente sotto un punto di vista informatico:

Un ambiente prevede le condizioni sotto la quale un'entità (agente o oggetto) può esistere. Esso definisce le proprietà del mondo nel quale un agente funziona e agisce.

Inoltre l'ambiente, non consiste solo di tutte le entità presenti, ma anche di quei principi e processi sotto la quale gli agenti esistono e comunicano. Come già menzionato precedentemente occorre distinguere la tipologia di ambiente a cui ci si trova di fronte, se è fisico o simulato. Nel primo caso, i sensori e gli attuatori dell'agente sono componenti hardware in grado di agire, modificare e collaborare con l'environment esterno, costituito quindi da oggetti tangibili. Altre volte si parla di environment simulato, o virtuale, cioè quando è costituito da un software che simula la realtà da modellare.
Questa tipologia si rivela particolarmente utile nelle prime fasi di sviluppo di un sistema multi-agente. Essendo l'ambiente fisico una componente complessa da rappresentare, dove i sistemi dovranno agire, si può adoperare una strategia semplificata, del tipo *Step-by-Step*, dove inizialmente si simula l'ambiente e si verifica l'idoneità del sistema e, per concludere, un volta accertata, si esegue un'implementazione reale.
Inoltre l'ambiente simulato può essere impiegato per tutte quelle tipologie di organizzazioni che non sono fisicamente situate (un'applicazione web o un sistema distribuito).

Vi sono inoltre ulteriori classificazioni che permettono di evidenziare le proprietà differenti degli ambienti. Tali sono comunemente generate dal punto di vista dell'agente:

- *Accessibile/Inaccessibile:* un ambiente è accessibile ad un agente quando quest'ultimo tramite i propri sensori è in grado di percepire ogni singolo aspetto dell'ambiente al fine di scegliere il giusto corso di azioni.
  Un ambiente di questo tipo è molto utile in quanto mantiene autonomamente memoria di tutto lo stato, togliendo tale compito all'agente. Esso potrà accedere alla memoria del sistema semplicemente percependo di nuovo l'ambiente ed ogni qualvolta che desidera. Il caso duale corrisponde ad un ambiente inaccessibile.
  Una ulteriore considerazione è se le risorse disponibili sono ampie o ristrette.

- *Deterministico/Non Deterministico:* un ambiente è deterministico quando lo stato successivo in cui esso si troverà è interamente determinato dallo stato corrente e dalle azioni svolte dagli agenti stessi. Nel caso duale l'ambiente è non deterministico ed è facile intuire che un ambiente accessibile, per la maggior parte dei casi sia anche deterministico; così come per un ambiente inaccessibile sia in molte situazioni non deterministico.

- *Statico/Dinamico:* come si può immaginare, un ambiente dinamico è un ambiente che può mutare improvvisamente il suo stato ed il suo comportamento.
  Mentre un ambiente statico, che semplifica notevolmente il processo di costruzione del sistema, corrisponde al caso duale, di un ambiente che raramente modifica il suo stato interno.

- *Discreto/Continuo:* si definisce un ambiente discreto quando vi sono un numero finito di azioni e percezioni che l'agente può effettuare. Contrariamente al caso di un ambiente continuo.

- *Diversificato:* quanto omogenee o eterogenee sono le entità dell'ambiente?

- *Volatile:* quanto può cambiare l'ambiente quando l'agente sta lavorando al suo interno?

- *In relazione alla locazione:* gli agenti hanno una locazione distinta nell'ambiente o potrebbero cambiare di posizione in relazione alla

condivisione dell'ambiente con gli altri agenti? O tutti gli agenti sono virtualmente collocati? Nel caso lo fossero, in che modalità di struttura? Sistema di coordinate, distanze metriche, posizionamento relativo?

Quindi per concludere possiamo affermare che l'ambiente ha un compito essenziale nella progettazione dei sistemi e che, senza tale, un agente è effettivamente senza utilizzo!

Capitolo 3

Multi-Agent System MAS

3.1 Introduzione

Dopo aver appreso il significato del termine agente, per il lettore dovrebbe essere semplice comprendere che un Sistema Multi-Agente MAS sia un sistema dove più agenti, presenti nello stesso ambiente, si muovono con l'intento di raggiungere diversi obiettivi, di cooperare le loro azioni e di far emergere un comportamento di tipo intelligente.
Generalmente ogni agente ha una propria sfera di influenza, cioè una porzione dell'ambiente che è in grado di controllare parzialmente. Nella maggior parte dei casi le varie sfere d'influenza si sovrappongono, risultando in una parte di ambiente controllata da più agenti: in un sistema di questo tipo, essi possono instaurare fra loro diversi tipi di rapporti organizzativi, come di parità o di subordinamento, ed avere un certo grado di conoscenza l'uno dell'altro.

Ogni agente è di per sé un sistema complesso da progettare e in misura maggiore lo sono i Sistemi Multi-Agente.
È inoltre da puntualizzare che un MAS può avere un alto grado di non determinismo dovuto ai possibili fallimenti delle azioni e che ogni agente ha una visione parziale e probabilmente distorta dell'ambiente stesso: ciò consente di modellare i MAS ad alto livello, rendendoli molto affini con il mondo reale, ma allo stesso tempo aumentandone la difficoltà.
Nonostante le elevate informazioni e le capacità possedute da ogni agente esso non sempre è in grado di risolvere un determinato compito da solo.
Per questa ragione la cooperazione per il raggiungimento degli obiettivi deve necessariamente prevedere lo scambio di informazioni tra gli agenti stessi.
Un *Multi-Agent System* MAS ha l'obbligo di controllare le forme di comunicazioni ed imporre leggi e regole riguardanti tali.

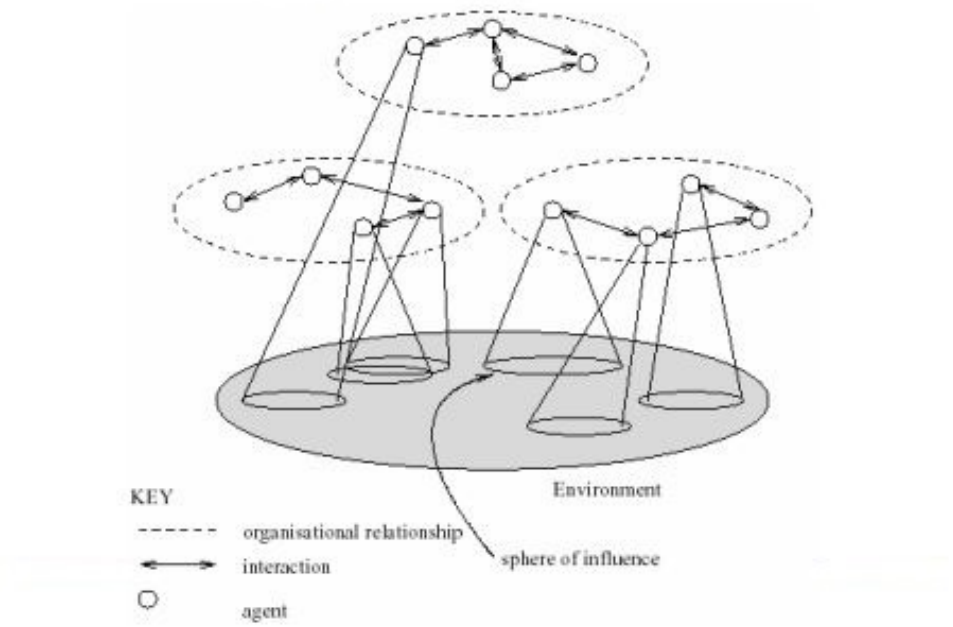

Figura 3.1: MAS Multi-Agent System

Nel momento in cui un agente fa parte di un rispettivo MAS, esso è in grado di comunicare e scambiare informazioni con i rispettivi membri del MAS, ma non solo, ogni agente può delegare i suoi obiettivi ad altri, oppure chiederne aiuto. In pratica, una volta entrati in un MAS, gli agenti collaborano tra loro per ottenerne il meglio.
Questo tipo di sistema si basa sui concetti di agente, contenitore di agenti, denominato *container*, e insieme di tutti i contenitori, detto *piattaforma*, ciò permette di modellare una struttura software complessa costituita da sistemi cooperanti tra loro per il raggiungimento di un obiettivo comune.

Come si può vedere dall'immagine ogni agente può richiedere, utilizzare o semplicemente osservare una o più risorse presenti nell'ambiente. Inoltre viene anche presentata una forma di comunicazione e collaborazione tra gli agenti stessi presenti nel medesimo MAS. Tali strutture saranno studiate ed analizzate nei capitoli successivi, ma utili comunque adesso per presentare al lettore una forma di introduzione.

In relazione alla cooperazione nasce la necessità di definire uno standard per la comunicazione, che comprenda:

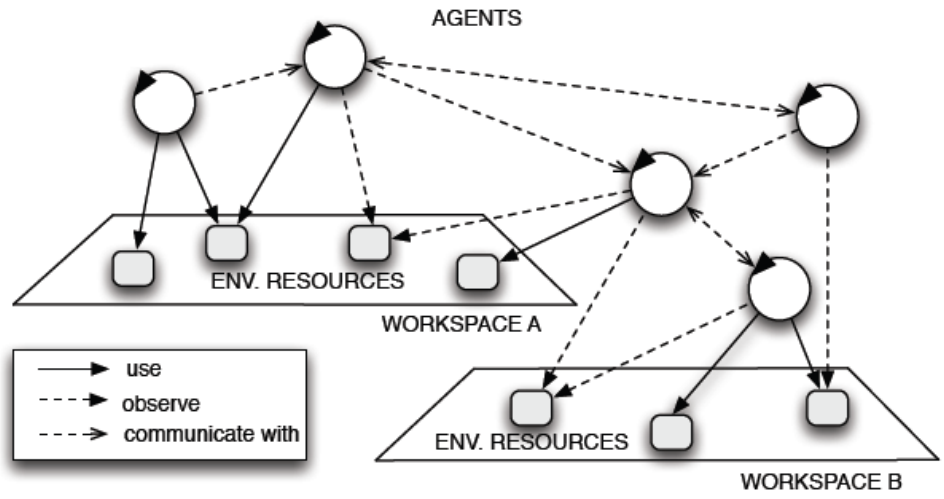

Figura 3.2: MAS and Resources

1. *Un protocollo ed un linguaggio di comunicazione:* che descriva le tecniche di comunicazione tra gli agenti ed un linguaggio che sia indipendente dalla struttura fisica degli stessi.

2. *Un formato per il contenuto dell'informazione:* che sia univoco e che dia la possibilità ad ogni agente di poter riconoscerne il contenuto.

3. *Una o più ontologie:* cioè fissati il linguaggio ed il formato per il contenuto dell'informazione, bisogna definire le ontologie per poter dare un significato apprezzabile ai contenuti trasmessi.

3.1.1 Motivi e ambiti di utilizzo

I sistemi multi-agenti stanno, negli ultimi anni, prendendo prepotentemente campo in differenti ambiti di sviluppo, tra i quali, fornire soluzioni a problemi inerentemente distribuiti (es. Air Traffic Control), permettere soluzioni in cui l'esperienza è distribuita (ad es. sanità), migliorare le performance se la comunicazione è mantenuta al minimo e garantire elevate forme di affidabilità come i *recover from failure*.
Vi sono comunque altri ambiti meno astratti e più vicini alle esigenze del commercio e della produzione che ricavano guadagno grazie all'approccio dei sistemi multi-agenti. Attraverso la messa a punto dei vincoli per esempio, si può cercare di comprendere quale sarà la combinazione più efficace per pervenire al risultato atteso come durante la costruzione di un ponte.

Il fattore importante è quindi il comportamento dell'insieme, non certo il comportamento individuale.

Un'altra caratteristica molto importante che consente di simulare correttamente i comportamenti di ogni singola entità è l'autonomia. Infatti permettere la rappresentazione di singoli comportamenti crea vantaggi in tutti quegli ambiti dove, come la chimica, si cerca di studiare il comportamento delle molecole in una vista microscopica e il comportamento generale del soggetto in una vista macroscopica. Quindi in questo caso l'agente rappresenterebbe la molecola mentre il MAS rappresenterebbe l'intero sistema, l'oggetto reale preso in considerazione.

Area	Example
Industrial	Manufacturing, Process Control, Telecommunications, Air Traffic Control, and Transport System
Commercial	Information Management, E-Commerce, and Business Process Management
Entertainment	Games, Interactive Theatre, and Cinema
Medical	Patient Monitoring, Health Care

Figura 3.3: MAS: Applications

Allo stesso tempo, l'ingegneria del software si è evoluta nella direzione di componenti sempre più autonomi. I sistemi ad agenti multipli possono essere visti come un raccordo tra l'ingegneria del software e l'intelligenza artificiale, con l'apporto rilevante dei sistemi distribuiti.
In rapporto ad un oggetto, un agente può prendere iniziative, può rifiutarsi di obbedire ad una richiesta, può spostarsi ed altro. Consentendo al progettista di concentrarsi sul lato umanamente comprensibile del software.

3.1.2 MAS: Organizzazione

L'organizzazione dei sistemi MAS viene suddivisa in quattro principali raggruppamenti, in relazione alla ridondanza e specializzazione delle azioni svolte. [22]

Organizzazione non-ridondante generalizzata: ogni agente può eseguire molte azioni ed ogni azione è eseguita solo da pochi agenti.
Organizzazione ridondante specializzata: ogni agente può eseguire solo poche azioni ed ogni azione è svolta da molti agenti.
Organizzazione ridondante generalizzata: ogni agente può eseguire molte azioni ed ogni azione può essere eseguita da molti agenti.
Organizzazione non ridondante specializzata: ogni agente può eseguire solo poche azioni ed ogni azione è eseguita da pochi agenti.

La tabella sottostante permette di identificare la corretta categoria in relazione all'organizzazione di ogni sistema.

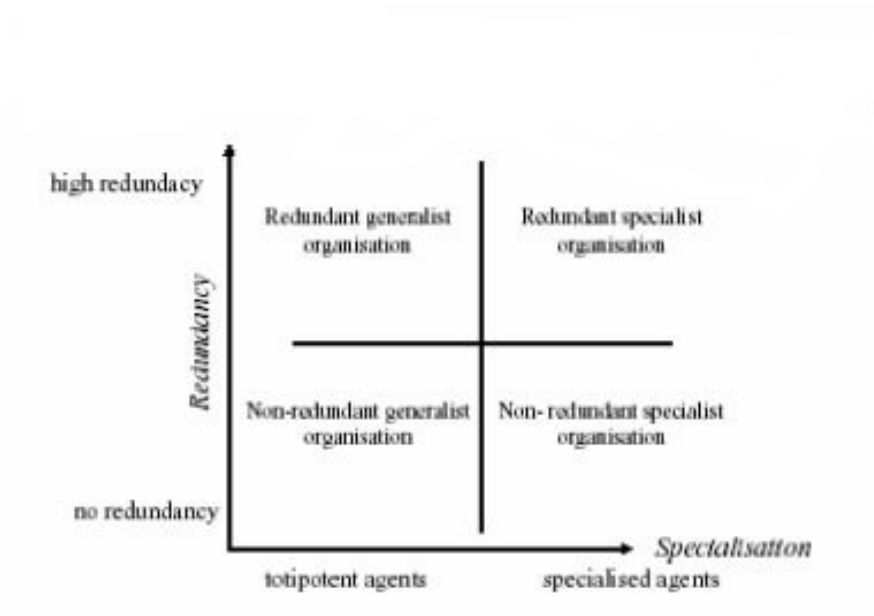

Figura 3.4: MAS: Organization

3.1.3 MAS: Modelli fisici

- *Blackboard* è un DKS nel quale si hanno varie *Knowledge Sources* (KS) indipendenti e asincrone che comunicano mediante un database condiviso, denominato *blackboard*, dove le KS possono leggere o scrivere in determinati livelli.

- *Contract-Net* esiste un'unità centrale che costituisce il piano e poi lo distribuisce ad agenti organizzati orizzontalmente. La distribuzione deve essere uniforme e si usa per questo un sistema di richieste/offerte tra agenti *manager* e agenti *contractor*.

- *Centralized Multiagent Planning* esiste uno schema centrale che, per essere eseguito, richiede che ogni agente svolga i suoi piani individuali. Ogni piano individuale può essere in opposizione ad altri, dovuto al fatto che un singolo agente ha una vista parziale, locale e non conosce completamente tutti i membri del sistema. Questi conflitti sono risolti tramite un agente privilegiato che riformula il piano globale per eliminarli. L'agente privilegiato può essere scelto in modo statico, inizializzazione, oppure direttamente durante l'esecuzione del piano.

- *Deductive Belief Model* questo modello divide gli agenti in due classi: gli osservatori e gli attori. I primi costruiscono dei piani in base alla conoscenza che hanno dei secondi.

- *Body-Head-Mouth* esistono tre componenti, uno funzionale, il corpo, orientato alla risoluzione di problemi, uno cooperativo, la testa, che possiede una conoscenza degli altri agenti e uno comunicativo, la bocca, che si occupa di mandare a destinazione le informazioni provenienti dalla testa e redigervi messaggi provenienti da altri agenti. Essi sono considerati egoisti e richiedono la collaborazione solo quando non sono in grado di risolvere un task. [33]

3.2 Architettura

È possibile delineare macroscopicamente gli elementi dell'architettura di un sistema ad agenti multipli come segue:

1. Gli agenti devono essere dotati di svariati sistemi di decisione e pianificazione. La ricerca operativa, o teoria delle decisioni, è una disciplina completamente dedita allo studio di questo soggetto. Nella categoria delle interazioni con l'ambiente.

2. Gli agenti necessitano inoltre di un modello cognitivo: esistono diversi modelli, tra cui uno dei più classici è il modello BDI, *Beliefs-Desires-Intentions*.

3. Gli agenti devono inoltre essere forniti di un sistema di comunicazione. Molti linguaggi specializzati sono stati sviluppati a questo scopo: il *Knowledge Query and Manipulation Language* (KQML), e più recentemente lo standard *FIPA-ACL* (ACL sta per *Agent Communication Language*) creato dalla *Foundation for Intelligent Physical Agents* FIPA.

4. Infine, l'implementazione effettiva di un sistema multi-agente che, pur non facendo parte dell'architettura del sistema, merita di essere menzionata. Attraverso i numerosi linguaggi di programmazione proposti, tra i quali AOP ed il più conosciuto OOP.

Questa sezione è tratta da [34].

3.3 Comunicazione e coordinazione

L'interazione è un aspetto cruciale nei sistemi multi-agenti e questo ne deriva in termini di comunicazione e di coordinazione nel momento in cui si desidera conseguire in maniera adeguata ad obiettivi comuni.
Il concetto di comunicazione in questo paragrafo viene solamente introdotto, essendo spiegato in maniera più esaustiva nei successivi capitoli con esempi e strutture di utilizzo. Nella sua definizione, la comunicazione è intesa come l'insieme dei fenomeni che comportano la distribuzione di informazioni.
Mentre il termine coordinazione sarà spiegato in maniera adeguata, per garantire al lettore una comprensione piena e dettagliata. Il problema principale nel lavoro cooperativo è la coordinazione espressa in termini di gestione delle dipendenze fra le attività di più agenti. Abbiamo bisogno essenzialmente di un meccanismo per poter gestire tali dipendenze quando esse si presentano. Tra i tipi principali di coordinazione multi-agente proposti, ne approfondiamo uno in maniera generica:
Parlando di *Partial Global Planning*, sviluppato inizialmente da *Durfee* nel 1988, gli agenti cooperano scambiandosi informazioni per raggiungere conclusioni sul *problem-solving*. Il planning è parziale poichè il sistema non può generare una soluzione per l'intero problema, ed è allo stesso tempo detto globale poichè gli agenti formano una soluzione non locale scambiandosi informazioni su soluzioni locali. Ogni agente quindi in base ai propri obiettivi

genera soluzioni parziali e locali, vengono poi scambiate informazioni per determinare come tali soluzioni possono interagire e in caso vengono modificate per coordinarsi. Una struttura dati detta PGP, *partial global plan*, generata cooperativamente, contiene informazioni sull'obiettivo globale del sistema.
In un sistema multi-agente possono istaurarsi forme di collaborazione o cooperazione. Una collaborazione viene stabilita quando un agente pur avendo tutte le informazioni e risorse per eseguire un obiettivo decide di accettarne e chiedere aiuto ad un altro agente, magari per velocizzare il compito.
Differente invece la cooperazione che avviene quando l'agente in questione non è in grado di portare a termine il lavoro per conto suo. A questo punto occorrono forme di coordinazione per prevenire il caos in cui ci si può facilmente trovare se non vi è un'entità di controllo centralizzato, oppure per incrementare l'efficienza del sistema, vedi l'esempio della collaborazione.
Oppure, avendo ogni agente informazioni incomplete al suo interno, lo scambio di informazioni è necessario per arrivare al conseguimento dell'obiettivo prescelto.

Capitolo 4

MODELLO DI STANDARDIZZAZIONE FIPA (Foundation for Intelligent Physical Agents)

4.1 Introduzione

Dopo aver introdotto l'importanza dell'Agent Software e specificato le sue caratteristiche è giunto il momento di conoscere come realizzarli.
Nella seconda metà degli anni '90 dei team di ricerca ed alcune aziende del settore hanno intrapreso un percorso di studio votato alla realizzazione di uno standard sulla tecnologia ad agenti.
Tra i risultati meglio noti vi è indubbiamente FIPA, che fornisce formati e protocolli generici per la comunicazione tra agenti e per la determinazione di contenuti e ontologie. Con il passare degli anni è divenuta un'organizzazione no-profit, leader nel settore della standardizzazione dei sistemi multi-agenti, con lo scopo di generare dei modelli di riferimento in grado di essere usati dai progettisti per implementare sistemi complessi, in qualsiasi area e settore, raggiungendo un elevato livello di interoperabilità.
Una buona parte degli ambienti di sviluppo per agenti adotta questi standard, nella conclusione di questo documento è presente un grafo che ne certifica la veridicità; mentre altri preferiscono adottare un approccio differente, ma comunque simile.
FIPA identifica un agente come l'elemento principale di ogni piattaforma ad agenti e unisce le funzionalità di uno o più servizi in un modello di esecuzione unico e integrato che può comprendere interazioni con programmi esterni,

utenti umani e strumenti di comunicazione. Un agente può svolgere anche il compito di mediatore per gli accessi software a risorse condivise. Ad ogni agente deve essere associato almeno un responsabile, per esempio in base all'appartenenza, ad organizzazioni o all'utente che lo esegue.
Grazie a queste forme di standardizzazione garantite da FIPA, *Fondation for Intelligent Physical Agent*, ed ai paradigmi di programmazione, *OOP* o *AOP*, è possibile progettare correttamente sistemi che implementano gli agenti e garantiscono scambio di informazioni, intra ed extra -sistema.

4.2 Agent Platform

Il modello standard di una piattaforma agente, come definita da FIPA, è rappresentato nella figura seguente. Esso comprende varie parti tra cui l'AMS e il DF, più un sistema di trasporto dei messaggi che permette la comunicazione tra loro. [26]
Occorre affermare inizialmente che, essendo una piattaforma software, non è detto che i confini di ogni entità siano delimitati in maniera definita. Anzi, può capitare che si sovrappongano.

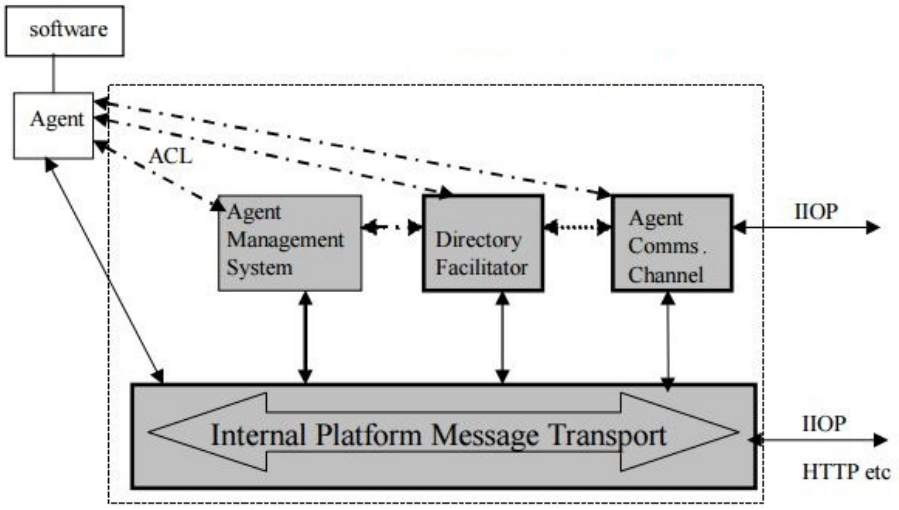

Figura 4.1: FIPA: AgentPlatform

Analizziamo ora le differenti parti che costituiscono tale piattaforma:

L'*Agent Management System*, AMS, corrisponde all'agente che esercita controlli di vigilanza e supervisione sull'accesso e sull'uso della piattaforma.
È inoltre responsabile per l'identificazione e la registrazione degli agenti residenti, compresi i metodi di creazione e cancellazione.
La piattaforma, *Agent Platform* AP, è un sistema software composto da uno o più contenitori di agenti, i quali devono appartenere, in ogni istante, ad una ed a una sola AP, con la facoltà, se necessario, di spostarsi sia internamente che esternamente.
Le regole della piattaforma implicano l'esistenza un solo AMS che prevede servizi come *white-pages* e *life-cycle*, mantenendo una directory di *agent identifiers*, dall'acronimo AID, e lo stato dell'agente stesso.
Una delle caratteristiche fondamentali dell'AMS è il fatto che può richiedere agli agenti presenti nella piattaforma, l'esecuzione di compiti specifici e, se necessario, forzarli.
Ad ogni agente, una volta registrato, viene assegnato un AID valido ed esistono due modi per fare ciò:

- L'*agent* viene creato sull'AP.
- L'*agent* è stato creato su un'altra AP ed è migrato in quella corrente. Questo principio vale solo per le piattaforme che supportano Agenti Mobili.

L'associazione ID-AGENT viene mantenuta dall'AMS, in modo di garantire un'immediata risposta allae richieste di quanti e quali agenti sono presenti nella piattaforma.
Trattandosi di un agente, anche l'AMS possiede un AID, riservato e rappresentato nella forma seguente:

agent-identifier
:name ams@hap
:address (sequence hapTransportAddress)

Se un agente viene deregistrato dall'AMS il suo ID viene liberato per essere riutilizzato in un possibile futuro. Generalmente il processo di deregistrazione avviene per rimozione o spostamento dell'agente dalla piattaforma originale.

Dato che esistono piattaforme con svariate capacità, l'AMS è in grado di generare una descrizione della propria; accessibile tramite il metodo *getDescription()*. Tale descrizione è comunque modificabile in una qualunque momento.

Mantiene inoltre in memoria l'intero ciclo di vita di ogni agente.

Il *Directory Facilitator*, DF, è l'agente che prevede il servizio di default *yellow-pages* nella piattaforma.
Esso, infatti, tiene traccia della descrizione, delle caratteristiche, delle funzionalità e della locazione di tutti gli agenti registrati, creando una lista dei servizi da essi offerti.
In ogni AP deve esistere almeno un Agent DF che è abilitato anche al mantenimento dei collegamenti con altri agenti situati su altre piattaforme.
Tra i metodi comuni presenti nell'AMS e DF individuiamo:

- *Register*: permette l'operazione di registrazione di un agente nella piattaforma.
- *Deregister*: operazione duale alla precedente, consente la deregistrazione dell'agente.
- *Search*: realizza l'operazione di ricerca di un agente, generalmente passando come parametro del metodo l'AID.
- *Modify*: abilita le operazioni di modifica dei criteri descrittivi.

Come detto precedentemente, ogni agente è caratterizzato da una collezione di coppie chiave/valore, chiamata *Agent Identifier*, AID.
Un AID comprende:

- Un nome.
- Altri termini tra cui: tutti gli indirizzi, IIOP, WAP, HTTP, che indicano dove è situato fisicamente l'agente, quindi su quale host, quale piattaforma e quale contenitore risiede, criteri necessari per la comunicazione e la migrazione. Inoltre vengono indicati anche i *resolvers*, cioè tutti quegli agenti speciali come l'AMS e il DF dove l'agente in questione è registrato.
  Possono, inoltre, essere supportati altri parametri a discrezione del progettista del sistema applicativo.

Solo il nome di un agente è invariabile mentre gli altri valori possono essere modificati durante tutto il corso della sua vita.

L'*Agent Communication Channel*, ACC, in FIPA2000 non risulta essere un agente ma si presenta come un'entità direttamente integrata nella piattaforma. Supporta il *Message Transport System*, MTS, come si evince dalla figura, cioè il servizio di comunicazione di default tra agenti sulla stessa e su

differenti piattaforme. Esso deve anche sostenere l'IIOP per garantire l'interoperabilità con le AP esterne.
Quando una piattaforma viene lanciata in esecuzione, l'AMS ed il DF sono immediatamente creati e il *Messaging Service* (implementazione del componente ACC) viene attivato per garantirne la comunicazione.

4.3 Agent LifeCycle

Un agente può essere, durante il suo ciclo di vita, descritto da uno dei differenti stati che lo comprendono, in accordo con l'*Agent Platform Life Cycle* delle specifiche FIPA, i differenti stati vengono rappresentanti e spiegati seguentemente:

- *Initiated*: l'oggetto agent è stato creato, ma non è ancora registrato nell'AMS, perciò non contiene né un nome né un indirizzo. Non è in grado quindi di comunicare con gli altri agenti.

- *Active*: l'agent è stato registrato nell'AMS, possiede a questo punto un nome regolare ed un indirizzo, può svolgere qualunque obiettivo desideri o gli sia imposto e può anche accedere a tutte le varie proprietà degli agenti, come la comunicazione e la cooperazione.

- *Suspended*: l'agent è momentaneamente sospeso, quindi il suo thread interno viene interrotto e nessun *Agent Behaviour* è in corso di esecuzione. Per entrare in tale stato viene invocato il metodo *doSuspend()*.

- *Waiting*: l'agent è bloccato, in attesa di qualcosa. Il suo thread interno sta dormendo e si risveglierà quando alcune condizioni, impostate e decise a priori dal programmatore, si verificheranno. Come per esempio l'arrivo di un messaggio specifico o di un input esterno. Per entrare in questo stato viene invocato il metodo *doWait()*.

- *Deleted*: l'Agent è definitivamente morto. Il thread interno ha terminato la sua esecuzione e l'Agent non è più registrato nell'AMS, dove avviene il processo di deregistrazione, l'identificatore associato viene rimosso e possibilmente assegnato ad un altro agente.
  Per entrare in questo stato viene invocato il metodo *destroy()*.

- *Transit*: questo stato è valido solo per gli *Agenti Mobili*, in grado perciò di spostarsi tra piattaforme differenti. Dove entra mentre sta migrando in una nuova locazione. La piattaforma continua a ricevere i messaggi, come se fosse ancora interno, fino a quando non trova la posizione definitiva e che saranno poi inviati alla nuova ubicazione dell'agente.

Ogni stato mette a disposizione dei metodi per permetterne il transito, questi metodi prendono il loro nome da idonee transazioni della *Macchina a Stati Finiti* mostrata nelle specifiche FIPA. Da notare che ad un agente è permesso di eseguire i suoi behaviours solo quando esso è in *active mode*.
Quindi, se alcuni behaviours chiamano il metodo *doWait()*, allora l'intero agente e tutte le sue attività sono bloccate, e non solo il behaviour che ha richiamato il metodo. A differenza del metodo *block()*, il quale permette la sospensione di un singolo behaviour all'interno dell'agente.
Lo stato di un agente è visibile e gestito dalla piattaforma.
La transazione di terminazione forzata di un agente, che non può essere ignorata, viene eseguita solo dall'AMS. L'invocazione del metodo *doSuspend()* può essere richiesta solo dall'AMS e lo stesso vale per il duale, cioè *resume()*. Compresa la riattivazione dell'agente dopo lo spostamento. Mentre lo stato di *waiting* con i rispettivi metodi sono interamente gestiti dall'agente stesso e l'entrata nello stato di transito dagli agenti mobili; come l'invocazione del metodo *move()* che può essere chiamata dall'agente stesso.

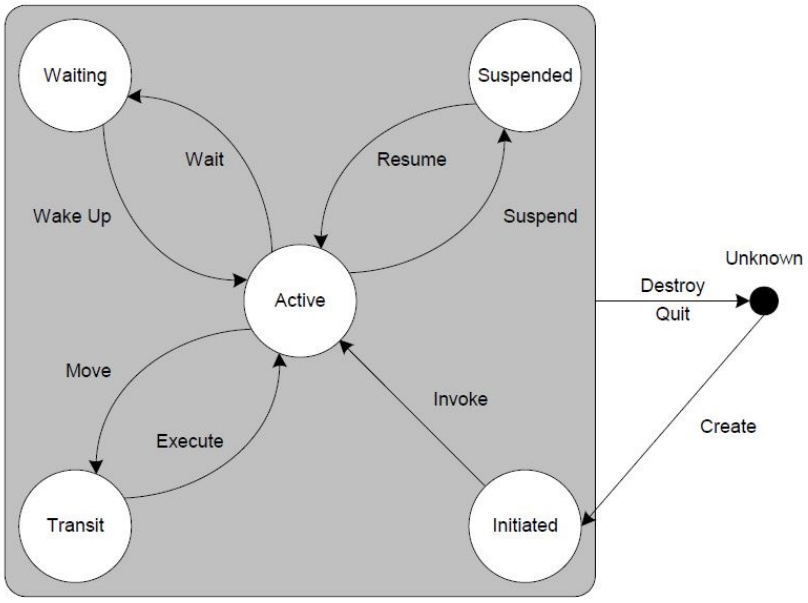

Figura 4.2: FIPA: AgentLifeCycle

4.4 ACL - Agent Communication Language

Agent Communication Language, ACL, è un linguaggio standard proposto per la comunicazione tra agenti, tra i più conosciuti e utilizzati vi sono:

- *Knowledge Query and Manipulation Language* KQML.

- FIPA-ACL.

Entrambi si basano sulla *teoria degli Atti Comunicativi* [31] sviluppata da *Searle* nel 1969, dall'idea che, tramite il dialogo, non solo si esprima un'affermazione ma, contemporaneamente, si compia una vera e propria azione. Esso definisce un *set di performatives*, frasi che non solo descrivono una data realtà, ma che sono in grado di cambiare la realtà sociale che stanno esprimendo, anche chiamati *Atti Comunicativi*.
Il contenuto dei *performatives* non è uno standard, ma variabile da sistema a sistema.

Questo modello di rappresentazione risulta particolarmente chiaro e permette di definire protocolli d'interazione ad alto livello, quali ad esempio:

Request Protocol, Query Protocol.

Per far sì che gli agenti si interpritino in maniera corretta gli uni con agli altri non solo devono parlare la stessa lingua ma anche avere un'ontologia comune. Un'ontologia è una parte base della conoscenza degli agenti che descrive che tipo di cose un agente può affrontare e come sono relazionate con gli altri agenti.
L'intento di FIPA, con la definizione di ACL, è quello di evidenziare un linguaggio comune fra gli agenti ed una semantica indipendente dalla struttura interna. FIPA non si occupa dei protocolli di rete e di trasporto a basso livello dell'architettura di comunicazione in un sistema distribuito, ma assume l'esistenza di tali servizi ed il loro utilizzo per la realizzazione degli *Atti Comunicativi*.

4.5 ACL Message

Una delle più importanti caratteristiche che gli agenti prevedono è l'abilità di comunicare. Il paradigma di comunicazione adottato è l'*asynchronous message passing*.
Ogni Agent ha una sorta di mailbox, definita *Agent Message Queue*, dove vengono pubblicati i messaggi inviati dagli altri agenti. Ogni qualvolta che un messaggio è pubblicato nella message queue l'agente ricevente viene avvisato. Il se ed il quando preleva il messaggio dipendere interamente dalla programmazione da parte dello sviluppatore. [12]
I messaggi scambiati tra agenti hanno un formato specifico definito dall'ACL, per garantire l'interoperabilità tra gli agenti.
La sintassi seguentemente esposta è corrispondente a FIPA-ACL, ma non vi è grande differenza con KMQL. Questo formato comprende un numero di campi tra i quali:

- Il *sender* del messaggio, cioè colui che invia l'informazione.
- La lista di *receivers*, cioè la lista degli Agent che possono ricevere il messaggio.
- Il *comunicative intention* che indica la tipologia di informazione contenuta nel messaggio. Ogni messaggio, infatti, deve dichiarare che tipo di dato intende comunicare. Se si sta inviando un'informazione (messaggio di tipo *INFORM*) il messaggio avrà una particolare struttura, se invece si sta effettuando una richiesta (messaggio di tipo *REQUEST*) la struttura sarà un'altra, inoltre l'agent che ha inviato il messaggio si aspetterà sempre una risposta dal destinatario.
- Il *content*, cioè il vero contenuto del messaggio.
- Il *content language*, cioè la sintassi utilizzata per esprimere il contenuto.
- L'*ontology*, esprime il vocabolario dei simboli utilizzati nel contenuto ed il loro significato.
- Ed alcuni campi utilizzati per controllare le diverse conversazioni correnti.

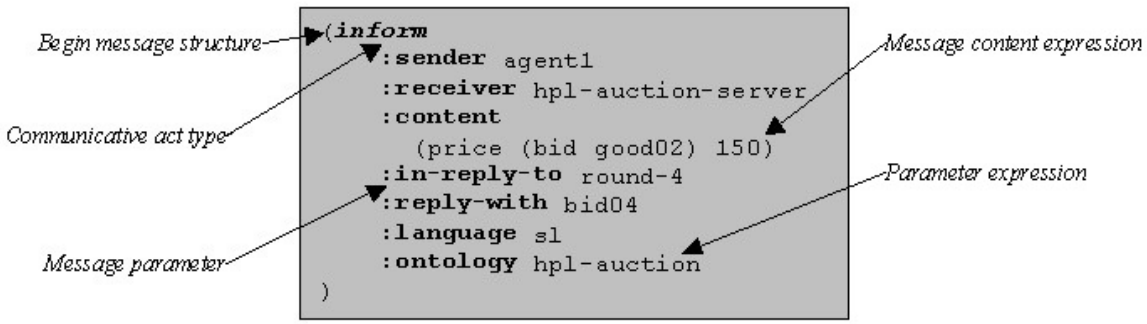

Figura 4.3: FIPA: ACL Message

Capitolo 5

MODELLO BDI (Belief, Desire, Intention)

5.1 Definizione

Tra i modelli più conosciuti e utilizzati, vedi da *AgentSpeak*, prende sempre più riconoscimento il **modello BDI**, proposto da *Michael Bratman* nel 1987, professore di filosofia presso la Stanford University. [6] Esso si ispira ai processi mentali che coinvolgono la logica umana e le implicazioni filosofiche che la determinana.

Ragionamento pratico significa valutare considerazioni contrastanti, a favore o contro opzioni alternative, ove le considerazioni sono date da ciò che l'agente desidera/valuta/preferisce e cosa crede. [36]

Una delle principali conseguenze è il fatto che un agente possa essere concepito come se avesse uno stato mentale. Programmarli dunque modificando quello che è il loro stato e non fornendo loro un elenco di istruzioni specifiche su compiti assegnati. Esso individua in *Belief, Desire, Intention* le tre componenti del pensiero umano.
Più specificatamente:

- *Belief*: insieme delle informazioni riguardanti l'ambiente, il contesto. Esso può contenere regole d'inferenza utili per la deduzione di nuove, ma tali informazioni possono essere anche parziali, incomplete o errate. Possono derivare da proprie percezioni oppure dalla comunicazione con altri agenti.
  Sono quindi una rappresentazione di quello che un agente crede dell'ambiente in cui è immerso.

- *Desire*: rappresenta lo stato motivazionale dell'agente ossia l'obiettivo che vuole eseguire e raggiungere. Influenzano le prossime azioni dell'agente in maniera diretta, offrendogli inoltre opzioni selezionabili per future mosse pratiche. Attenzione, più *desires* possono essere in conflitto tra loro, così come avviene negli esseri umani, nella quale però si specifica un indice di priorità che permette la realizzazione di obiettivi assieme irrealizzabili e/o che si escludono a vicenda.

- *Intention*: rappresenta lo stato deliberativo dell'agente ossia i *desires* per i quali ha pianificato delle azioni da eseguire e portare a termine. Un agente può perseguire obiettivi perché gli sono stati delegati da altri, oppure a seguito di una sua scelta.

I *beliefs* sono essenziali dato che il mondo è dinamico e il sistema ha solo una vista parziale e limitata di esso. Tuttavia è importante memorizzare alcune informazioni piuttosto che ricalcolarle a partire dai dati a disposizione, in modo da garantire un notevole incremento delle performance e risparmio di tempo. [20]
I *desires*, più comunemente conosciuti come *goals*, costituiscono un altro elemento fondamentale dello stato del sistema. In termini computazionali un goal può essere il valore di una variabile, un record, un'espressione simbolica. L'aspetto importante è che esso rappresenta uno stato finale desiderato.
I piani completati sono detti *intentions* e rappresentano il terzo componente dello stato di un sistema progettato secondo il modello BDI. Per intentions si intendono un insieme di threads eseguiti in un processo che può essere interrotto appropriatamente a seguito della ricezione di un cambiamento del contesto.

Risulta mostrato, in figura, un esempio del ciclo interno del modello BDI, dove vengono espressi i vari stati mentali, beliefs, desires e intentions.
Il modello BDI riconosce l'importanza delle intenzioni come elemento fondamentale nel guidare il ragionamento pratico. La loro caratteristica più importante risulta essere la pro-attività, ossia la tendenza a compiere azioni. Tuttavia avere un'intenzione non porta necessariamente alla produzione di azioni.
Una seconda qualità non trascurabile è la persistenza delle intenzioni. Se fallisce il tentativo di conseguire ad un obiettivo precisato in un'intenzione si esegue una nuova manovra, ma se le ragioni perdono di valore, allora l'intenzione può essere abbandonata.
Una terza proprietà è data dai vincoli che, l'aver adottato un'intenzione, pone sul ragionamento pratico futuro. Questa caratteristica è desiderabile

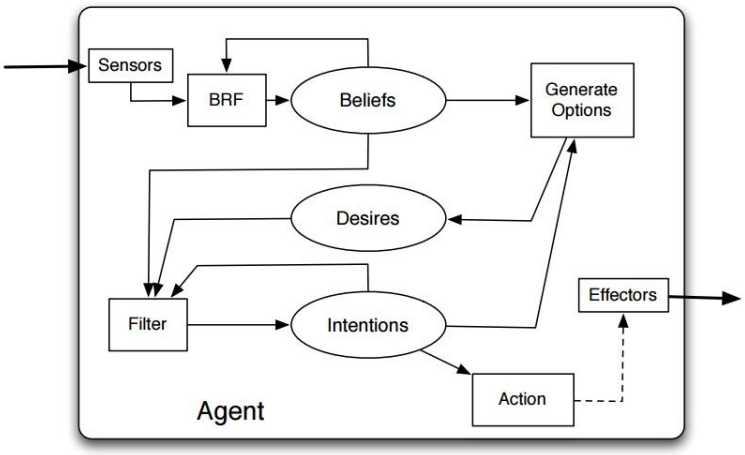

Figura 5.1: BDI Model

dal punto di vista dell'implementazione di agenti razionali poichè consente l'impiego di un filtro di ammissibilità sulle intenzioni prossime di un agente.

5.2 Practical Reasoning

Nella dinamicità del pensiero umano, ci si trova di fronte continuamente a problemi di vario genere e si deve scegliere il percorso di azioni da intraprendere per risolverli. Per far fronte quindi ad un problema, il ragionamento umano cerca di delineare tutti gli aspetti, cercare le possibili soluzioni e, sulla base delle proprie abilità, iniziare una serie di step per la risoluzione finale. Questo corrisponde esattamente al modello decisionale adottato da BDI, chiamato *Practical Reasoning*. Argomento parzialmente introdotto nel secondo capitolo di questo elaborato quando si accennava ai diversi tipi di agenti in circolazione.
Esso si divide in una fase di *deliberation* e una di *means-end reasonign*:
Quando si parla di deliberation, è necessario considerare le intentions prese in considerazione dall'agente. In questo stadio si determina quali tra quest'ultime possono essere perseguite e verificate, sulla base delle proprie abitudini, garantendo una forma di fattibilità. Questo non implica che, una volta designata una intention, essa venga automaticamente conclusa, ma che l'agente cercherà in ogni modo di eseguirla e completarla, provando a portarla a compimento. Nel caso in cui esso non riesca, per fallimenti o altre motivazioni, l'intention potrebbe essere rieseguita o abbandonata definitivamente.

Successivamente, nell'attività di *means-end reasoning*, viene scelta la modalità con la quale si raggiungere l'obiettivo prefissato nello stato precedente.
Per concludere possiamo semplicemente sostenere che le due fasi corrispondono al **cosa** si vuole svolgere ed al **come** si desidera procedere.
Possiamo affermare che si tratta di una vera e propria forma di planning, ovvero la costruzione di un algoritmo che permette, in base al nostro obiettivo, alle conoscienze acquisite dall'agente in quel rispettivo ambiente esterno e alle azioni possibili, di produrre un piano che verrà eseguito nell'immediato futuro.
Ultimi sviluppi hanno portato ad una modernizzazione di tale modello. Infatti è possibile controllare, modificare e prendere decisioni a riguardo anche quando l'esecuzione è in corso d'opera. Verificare quindi se l'azione scelta è ancora la miglior cosa da fare. Ciò rende il flusso incredibilmente flessibile e sempre più simile al comportamento umano, dato che, giornalmente, modifichiamo le nostre idee ed i corsi d'azione intrapresi, per cedere gli interessi ad altro.

5.3 Pregi e difetti

I pregi del modello BDI:

- Capacità di costruire piani d'azione che reagiscano a specifiche situazioni e che possano essere invocati sulla base degli scopi stabiliti. Questi piani sono sensibili al contesto in cui sono invocati e semplificano lo sviluppo modulare e incrementale.

- Equilibrio tra la componente reattiva ed il comportamento goal-oriented del sistema.

- Il linguaggio di programmazione e la rappresentazione ad alto livello consentono all'utente finale di codificare la conoscenza direttamente in termini di attitudini mentali senza dover ricorrere a costrutti di programmazione a basso livello.

I limiti [27]:

- Gli agenti BDI mancano di meccanismi per apprendere da eventi passati che gli permettano di adattarsi alle nuove situazioni.

- Gli studiosi della teoria decisionale classica si chiedono se sia necessario disporre delle tre attitudini mentali mentre i ricercatori dell'Intelligenza Artificiale distribuita si chiedono se siano sufficienti le tre attitudini.

- La logica multi-modale proposta dal modello BDI ha poca rilevanza in pratica.
- Il modello non descrive esplicitamente i meccanismi di interazione tra gli agenti e la loro integrazione in sistemi multi-agente.

Capitolo 6

Agent-Oriented Programming AOP

6.1 Overview AOP

È in questo capitolo che si fa riferimento ad un nuovo paradigma della programmazione, esso rappresenta una forma di evoluzione rispetto a quello ad oggetti e prende il nome di *Agent-oriented programming*, o *programmazione orientata agli agenti*, AOP. Permette lo sviluppo di applicazioni complesse che prevedono caratteristiche di alto livello. Principale scopo del paradigma, come è lecito immaginarsi, è di rappresentare e realizzare correttamente le entità e le proprietà degli agenti.
Storicamente il concetto di programmazione orientata ad agenti è stato presentato dal professore della Standford University, *Yoav Shaoham* nel 1993. Negli anni successivi molti *Agent Programming Languages*, APL, e linguaggi per la programmazione multi-agente sono stati proposti in letteratura, ma il nucleo è sempre rimasto il concetto di agente e di stato mentale a cui appartiene.

6.1.1 Conclusioni

Molti linguaggi di programmazione sono stati proposti per facilitare l'implementazione di MAS basati su agenti intelligenti. I linguaggi di programmazione sono necessari anche per colmare la distanza tra le fasi di analisi e di progettazione, da cui si ricava il prospetto di un sistema agent-oriented e l'implementazione di un MAS. Lo scopo dei linguaggi di programmazione ad agenti è di fornire il supporto per un'implementazione piuttosto diretta dei concetti chiave che fanno parte della mentalità agent-oriented.

La comunità informatica è stata particolarmente produttiva nell'area dei *programming frameworks* per sistemi ad agenti. La figura mostra una panoramica dei linguaggi suddividendo tra programmazione *Java-based* e *Logic-based*.

Figura 6.1: Families of Agent Programming Languages

I linguaggi Java-based rimango più vicini al ben conosciuto e familiare paradigma di programmazione ad oggetti mentre i linguaggi Logic-based prevedono motori di ragionamento più potenti per ragionare su beliefs e goals.
Un esempio di Java-based è il linguaggio JACK, visto come implementazione del modello BDI per agenti razionali Java con estensioni per supportare la progettazione e l'esecuzione di sistemi ad agenti. Un ulteriore è Jadex, introdotto e costruito sulla base di Java, incrementando ed estendendo Jade con agenti BDI.
Diversi studi e sondaggi dimostrano questa tendenza, dove circa il 75% di tutte le applicazioni agent-based utilizzate e immesse nel mercato corrispondono ai più popolari linguaggi di programmazione, come Java, C e PHP.
Un importante contributo al lavoro svolto è stata l'introduzione di moduli che supportano la progettazione modulare di questi programmi. I concetti di modulo e capacità sono strettamente correlati e permettono la strutturazione di agenti BDI in raggruppamenti funzionali che supportano un elevato grado di riusabiltà. Un linguaggi di programmazione che prevede estensioni con moduli è Jason. Non vi sono solo la modularità e la capacità tra gli approcci principali per la costruzione di sistemi multi-agente, infatti altri aspetti base potrebbero essere l'organizzazione ed il debugging. Tutte queste estensioni permettono un miglior utilizzo lato pratico.
Un'importante caratteristica dei sistemi ad agenti riguarda il fatto che essi sono immersi e interagiscono con un ambiente. Per questo motivo sono stati proposti diversi modelli che supportano l'interazione tra gli agenti e

l'ambiente che li circonda. Un esempio potrebbe essere Agent and Artifacts, A&A,basato sull'idea che un ambiente è composto da differenti specie di artefatti che vengono condivise e utilizzate da agenti per supportare le loro attività. [13]

6.2 Overview OOP

La programmazione orientata agli oggetti o *Object Oriented Programming*, OOP, è un paradigma di programmazione in cui un programma viene visto come un insieme di oggetti che interagiscono tra loro.
Nei linguaggi OOP esiste un nuovo tipo di dato, la classe che serve a modellare un insieme di oggetti dello stesso genere.
In generale, un oggetto è caratterizzato da un insieme di attributi e funzionalità.
I concetti base su cui si fonda questa programmazione sono la classe, l'incapsulamento, l'ereditarietà, il polimorfismo e l'oggetto stesso.
Un linguaggi di programmazione ad oggetti quindi offre costrutti espliciti per la definizione di entità, in questo caso oggetti, che incapsulano una struttura dati e le operazioni possibili da eseguire su di essa.

6.3 Confronto tra AOP e OOP

La AOP presenta costrutti mentali per la progettazione e l'analisi del sistema computazionale e per la rappresentazione dello stato, compito che nella OOP viene svolto dalle variabili di stato. [38]
Mentre la OOP propone la visione di un sistema composto da moduli in grado di comunicare tra loro e con modalità individuali di gestione dei messaggi in arrivo, la AOP specializza questo modello stabilendo lo stato dei moduli, ovvero lo stato mentale degli agenti, in modo che sia composto da opinioni, capacità e decisioni, ognuna delle quali prevede una sintassi ben definita.
Sono posti vari vincoli nello stato mentale degli agenti che sono approssimativamente in corrispondenza con le loro controparti umane. Una computazione vede gli agenti informarsi, inoltrare richieste, fare offerte, accettarle, rifiutarle e assistersi tra loro. Il comportamento di un agente estende quello di un oggetto, perché gli agenti hanno la libertà di controllare e di cambiare i propri comportamenti, inoltre non richiedono stimoli esterni per completare i loro compiti: per questo vengono definiti elementi attivi, mentre gli oggetti passivi.
Il comportamento degli agenti presenta un certo grado di imprevedibilità, a

causa delle interazioni che si verificano, della loro autonomia e per il fatto che gli effetti delle loro azioni non sono certi prima che queste vengano compiute. Hanno l'abilità di comunicare con l'ambiente e con altre entità, anche ingaggiando transazioni multiple concorrentemente.
Nella OOP è presente il concetto di classe per la definizione del tipo e della funzione di un oggetto; invece nella AOP questa viene sostituita con il termine di ruolo; analogamente i metodi lasciano il posto ai messaggi, i quali sono indipendenti dall'applicazione in quanto seguono la sintassi standard dell'ACL in uso, a differenza dei metodi per un oggetto che devono essere specifici ad-hoc per quella classe. [29]
Ulteriori concetti fonte di diversificazione che l'AOP adotta sono:

- *Decentralizzazione*, cioè nella OOP gli oggetti sono organizzati centralmente, perché i metodi vengono invocati sotto il controllo di altri componenti del sistema, mentre per gli agenti la computazione può avvenire sia in maniera centralizzata che decentralizzata, internamente o esternamente all'agente.

- *Classificazione multipla e dinamica*, ovvero gli agenti presentano un meccanismo più flessibile rispetto a quello degli oggetti, in quanto quest'ultimo, una volta istanziato da una classe, non potrà mai cambiarla.

- *Impatto minore*, se in un sistema di agenti ne viene perso uno durante un'elaborazione, il sistema può far fronte a questa evenienza grazie al supporto della collettività di quelli rimasti nell'ambiente; nella OOP al mancare di un oggetto viene lanciata normalmente un'eccezione.

- *Emergenza*, ovvero in un gruppo di agenti che collaborano vi saranno caratteristiche emergenti, non proprie del singolo agente, ma dovute al fatto che le entità formano una società grazie all'interazione; gli oggetti, invece, non interagiscono per propria natura fra loro, o meglio non senza un meccanismo a livello superiore che lo permette.

OOP versus AOP

	OOP	AOP
Basic unit	object	agent
Parameters defining state of basic unit	unconstrained	beliefs, commitments, capabilities, choices, ...
Process of computation	message passing and response methods	message passing and response methods
Types of message	unconstrained	inform, request, offer, promise, decline, ...
Constraints on methods	none	honesty, consistency, ...

Figura 6.2: AOP: Agent-Oriented and Object-Oriented Programming

Capitolo 7

RUOLO DEI LINGUAGGI E DELLE TECNOLOGIE AD AGENTI NELL'INGEGNERIA DEL SOFTWARE

7.1 Introduzione

L'ingegneria del software, *Software Engineering*, è una disciplina metodologica volta allo studio di criteri di produzione, di teorie alla base di essi e di strumenti di sviluppo e misura della qualità dei sistemi. Ponendosi una serie di obiettivi legati all'evoluzione dello sviluppo del software sia dal punto di vista tecnologico che metodologico.
È inoltre anche una disciplina empirica, cioè basata sull'esperienza e sulla storia dei progetti passati. [8]

7.1.1 Ing. del Software ad Agenti

Con l'avvento della computazione autonoma e distribuita la necessità di tecnologie basate sugli agenti è divenuta essenziale, ma non era possibile progettare senza un approccio disciplinato ed ingegneristico. Lo sviluppo di un sistema multi-agente dovrebbe sfruttare fruttuosamente astrazioni coerenti con le caratteristiche analizzate nei primi capitoli di questo documento, cioè:

- Gli agenti, come entità autonome, indipendenti e situate in un ambiente. Argomento trattato nel capitolo 2.

- L'ambiente, il mondo e le risorse che gli agenti percepiscono. Argomento trattato nella sezione 2.5.

- Protocolli di interazione, come atti di interazione tra gli agenti.

Tali astrazioni sono utilizzate per tradurre in concreto le entità dei sistemi software. E, grazie alle metodologie, viene garantita la codifica dei criteri di risoluzione dei problemi, centrate attorno alle specifiche astrazioni agent-oriented.
Ogni metodologia potrebbe aggiungere ulteriori astrazioni per modellare il software e organizzare i processi, come ruoli, organizzazioni, responsabilità, beliefs, desires e intentions. Non tutti direttamente traducibili in entità concrete del sistema, infatti il concetto di ruolo è un aspetto dell'agente e non un agente stesso.
L'intero compito spetta all'*Ingegneria del Software Orientata agli Agenti*, AOSE, nata in risposta alle esigenze di disporre di approcci originali e innovativi per l'ingegnerizzazione del software. Essa rappresenta un nuovo paradigma adottato per applicare una miglior pratica nello sviluppo di sistemi multi-agenti complessi e a tutto ciò che ne concerne.
AOSE richiede specifici tools agent-oriented, infatti UML non garantisce la modellazione di sistemi multi-agenti e delle loro interazioni, questo è dovuto al fatto che le astrazioni agent-oriented sono differenti da quelle di tipo object-oriented.

Trattando quindi le conclusioni possiamo affermare che l'ingegneria del software definisce la mentalità del sistema, un set di astrazione per essere utilizzate nello sviluppo del software e conseguentemente delle metodologie e tools.
Mentre l'ingegneria del software orientata agli agenti definisce le astrazioni degli agenti, l'ambiente, i protocolli di interazione e il contesto in cui agiscono. Specifica inoltre metodologie, affrontate nei paragrafi a venire, e tools opportuni e adatti. Studi recenti indicano che quest'ultimo paradigma appare essere applicabile ad un elevato raggio di applicazioni distribuite. [39]

7.1.2 Prenderà campo in larga scala? Perché? Vantaggi e svantaggi

In questo paragrafo cercheremo di individuare i motivi per cui le tecniche basate sugli agenti potrebbero avere successo nell'immediato futuro, consentendone una distribuzione di massa e rendendolo partecipe alla principale corrente dell'ingegneria del software. Quindi, in poche parole, determinare se avrà successo come un tradizionale paradigma.
Pur sembrando un argomento banale esso riscontra molte più difficoltà di quel che sembra, infatti la storia passata dimostra che, pur avendo sviluppa-

to apparentemente tecniche promettenti, esse non hanno mai preso campo in larga scala.
Fortunatamente ci sono due interessanti aspetti che ci permettono di credere nella grande distribuzione di questa tecnologia:
Come prima ragione, l'approccio basato sugli agenti può essere visto come un naturale e successivo passaggio nell'evoluzione dell'ingegneria del software.
Mentre come secondo aspetto, le tecniche basate sugli agenti sono l'ideale modello computazionale per lo sviluppo di software come sistemi aperti e distribuiti.
La vista del mondo tramite la concezione agent-oriented è forse il modo più naturale di rispondere a molti tipi di problemi. Il mondo reale è popolato da oggetti passivi che svolgono operazioni su loro stessi, e così equamente anche da attivi, intenzionali agenti che interagiscono per svolgere e archiviare i loro obiettivi.

Noi vediamo il mondo come un set di agenti autonomi che collaborano per portare a termine alcune funzioni di alto livello. [5]

I blocchi di costruzione base dei modelli di programmazione esibiscono gradi incrementali di posizionamento e incapsulamento. [24] Gli agenti seguono questo trend implementando il concetto di obiettivo all'interno di ognuno, dando all'agente il suo thread di controllo e selezionando le azioni di incapsulamento. Inoltre gli agenti cercano di porta i meccanismi di riuso ad un livello superiore, abilitandolo nei sottosistemi e nelle interazioni flessibili.

Purtroppo non è tutto oro ciò che luccica e, dopo aver introdotto i vantaggi dell'approccio agent-oriented, analizziamo i possibili inconvenienti che potrebbero presentarsi. L'obiettivo è di identificare quegli aspetti che, durante lo sviluppo di sistemi complessi, si sono fatti più complicati adottando un criterio basato sugli agenti.
Sono presenti due principali svantaggi associati alla vera e propria essenza dell'approccio ad agenti:

- I patterns ed i risultati delle interazioni sono intrinsecamente imprevedibili.

- Predire il comportamento dell'intero sistema, basandosi sui componenti che lo costituiscono, è estremamente difficile, alcune volte impossibile, a causa delle elevate possibilità che emergano altri componenti.

Sebbene la flessibilità delle interazioni di un agente abbia molti vantaggi, il rovescio della medaglia è che essa conduce all'imprevedibilità del sistema a run-time. Gli agenti decidono a tempo di esecuzione quale dei loro

obiettivi richiede l'interazione in uno specifico contesto, con quali altri agenti interagiranno in maniera da realizzare i loro compiti e quando avverrà la comunicazione. Perciò le decisioni sul numero, i pattern e la tempistica di interazione dipendono da una complessa influenza reciproca dello stato interno dell'agente, della percezione sull'ambiente, magari includendo lo stato degli altri agenti presenti nella stessa piattaforma, e del contesto di organizzazione che esiste quando la decisione viene presa.
Combinando questi molteplici fattori si sottolinea la circostanza che è difficile fare previsione sulle interazioni del sistema. Dal momento in cui gli agenti ritengono l'autonomia sopra ogni loro stessa scelta, una richiesta potrebbe essere immediatamente onorata, completamente rifiutata o potrebbe essere modificata attraverso alcune variabili di scambio sociale.
In breve, nel caso generale, sia la natura, una semplice richiesta oppure una lunga trattativa, che il risultato di un'interazione, potrebbero non essere determinati al momento della prima comparsa. Tutto il mondo è non-deterministico!
La seconda fonte di imprevedibilità nei sistemi agent-oriented riguarda la nozione di *Emergent Behaviour* o comportamento emergente. Per *emergent behaviour* si intendono quei comportamenti del sistema che non dipendono dalle sue parti individuali, ma dalla relazione che le lega. Questo implica che, tale comportamento, non può essere predetto dall'osservazione e studio dei singoli enti ma può essere intuito, gestito e controllato solo studiandone le parti e le relazioni che lo corrispondono. Perciò il concetto del tutto, spesso se non sempre, è più grande della sola somma della parti. [18]

Entrambi gli inconvenienti si applicano al caso generale di utilizzo di un approccio basato su agenti. Quindi la domanda sorge spontanea, conviene o meno adottare tale metodologia?
In specifici sistemi e applicazioni, i programmatori sono in grado di eludere queste difficoltà utilizzando protocolli di interazione le quali proprietà possono essere formalmente analizzate, a volte prendendo in prestito tecniche da ambiti differenti, adottando rigide e prestabilite strutture organizzative e/o limitando la natura e lo scopo dell'interazione tra le entità. In tutti questi casi, l'obiettivo è ridurre l'imprevedibilità del sistema limitando il potere che garantisce l'approccio agent-oriented. Pertanto, per soddisfare le caratteristiche del sistema complesso desiderato **si possono adottare tecniche agent-oriented** riducendo i suoi gradi di imprevedibilità.

7.2 Metodologie

Una metodologia è un set di linee guida per coprire l'intero ciclo di vita di un sistema, dallo sviluppo alla diffusione e manutenzione, sia dal punto di vista tecnologico che gestionale.

Le metodologie AOSE promuovono un approccio disciplinato per analizzare, progettare e sviluppare i sistemi multi-agente, adottando metafore e tecniche specifiche. A tale scopo, queste metodologie identificano le astrazioni di base che saranno utilizzate nello sviluppo, tipicamente agenti, ruoli, risorse e strutture organizzative, avvalendosi di un meta-modello, strumento fondamentale per studiare le metodologie e poterle confrontare tra loro. [2]
A tali astrazioni di base vengono aggiunte quelle che si potrebbero generare utilizzando una metodologia designata.
Esse sono in genere organizzate e suddivise in diversi modelli e/o fasi, strettamente correlati tra loro, in modo da fornire linee guida su come procedere nell'analisi, nella progettazione e nello sviluppo, specificando con chiarezza i risultati attesi da ogni fase. Producendo modelli, come rappresentazioni astratte di alcuni aspetti di interesse del software, e artefatti, visti come documenti che descrivono le caratteristiche del software. [39]
La formazione di diverse metodologie riguarda il contenuto di astrazioni che le hanno rese più adatte a scenari rispetto che altre.

7.2.1 GAIA

GAIA è una metodologia general-purpose, proposta per la prima volta da *Jennings* e *Wooldridge* nel 1999 poi modificata ed estesa grazie alla collaborazione di *Zambonelli* nel 2003. Essa mira a guidare il progettista nello sviluppo di un sistema multi-agente attraverso l'identificazione di una sequenza di modelli organizzativi e di relazioni che sussistono tra loro.
Le caratteristiche principali di GAIA riguardano le astrazioni organizzative, il concepire un sistema multi-agente come un'organizzazione di individui, ognuno dei quali gioca ruoli specifici nell'organizzazione ed interagisce in accordo con essi.
La progettazione mediante GAIA inizia con la fase di analisi, il cui scopo è organizzare le specifiche e sviluppare una comprensione del sistema e della struttura tramite regole e interazioni. Ciò rappresenta il prerequisito per la progettazione, senza far alcun riferimento ai dettagli dell'implementazione.
L'analisi porta alla composizione di quattro modelli:

- Il *modello dell'ambiente* evidenzia le caratteristiche dell'ambiente in cui il sistema andrà ad operare unitamente alle risorse che lo compongono.

- Il *modello preliminare dei ruoli* identifica le capacità che devono essere presenti nel sistema indipendentemente dalla struttura organizzativa che verrà scelta nelle fasi successive. In termini di risultati otteniamo permessi, protocolli di interazione e attività.

- Il *modello preliminare delle interazioni* cattura le dipendenze e le relazioni tra i vari ruoli identificati nel modello precedente e le formalizza mediante l'ausilio di protocolli di interazione. In termini di risultati otteniamo protocolli per i dati scambiati ed i partner coinvolti.

- La specifica di un insieme di regole organizzative che devono valere per tutta l'organizzazione.

Ogni modello rappresenta un output utile per le fasi successive.
Si procede poi con la fase di progettazione, occorre affermare fin da subito che le fasi sono ben separate tra loro, aspetto che non appartiene a tutte le metodologie.
Essa ha lo scopo di definire la struttura del sistema dell'agente, in termini di classi e istanze dei servizi che vengono forniti.
Questa fase è divisa in due parti: la progettazione architetturale e la progettazione di dettaglio.
La *progettazioni architetturale* si avvale dei risultati prodotti dalla fase di analisi per identificare un modo efficiente e affidabile per strutturare l'organizzazione del sistema multi-agente. La scelta della struttura organizzativa è la fase più critica perché impatta su tutte le fasi successive. GAIA propone come linea guida di scegliere sempre l'organizzazione topologica più semplice, che permetta di gestire la complessità sia computazionale sia di coordinazione delle entità.
Una volta stabilita la struttura organizzativa si conoscono sia i ruoli necessari, sia come essi debbano interagire, sia quali protocolli debbano essere associati, ciò permette di completare il modello dei ruoli e quello delle interazioni.
La progettazione di dettaglio si compone a sua volta di due modelli:

- Il *modello degli agenti* identifica quali ruoli ogni classe di agente sia autorizzata ad interpretare e quante istanze della stessa debbano essere definite nel sistema.

- Il *modello dei servizi* identifica i servizi associati a ogni ruolo. È opportuno sottolineare che il concetto di servizio in GAIA è diverso da quello tipico del mondo object-oriented: un servizio infatti può essere pensato come un singolo blocco di attività che potrebbe o meno essere

invocato dall'esterno, poichè un agente è autonomo e quindi è lui stesso a decidere di eseguire un particolare servizio senza che questo venga stimolato dall'esterno.

I limiti della metodologia sono diversi. In primo luogo, la neutralità di GAIA rispetto alla tecnologia di implementazione, che di per sé è un punto di forza, ma che può potenzialmente determinare problemi nel caso in cui la tecnologia di sviluppo introduca vincoli non considerati nella fase di progettazione.
Inoltre, GAIA non prevede la fase di analisi preliminare dei requisiti, né supporta il processo iterativo, costringendo così il progettista a dover riformulare tutti i modelli nel caso di modifica di un requisito. [2]

7.2.2 SODA

SODA, *Societies in Open and Distribuited Agent spaces*, è una metodologia per l'analisi e la progettazione di sistemi ad agenti che si concentra sugli aspetti inter-agente. In particolare SODA si focalizza sulla modellazione delle società di agenti e sulla descrizione/percezione dell'ambiente che circonda il sistema.
A tale fine SODA introduce il meta-modello ontologico *Agents and Artifacts*, A&A, che si basa sull'idea di modellare i sistemi non più solo tramite il concetto di agente, ma sfruttando anche la nozione di artefatto per rappresentare esplicitamente l'ambiente. Così, mentre gli agenti sono usati per modellare le attività individuali, gli artefatti definiscono l'ambiente del sistema e costituiscono sia la colla che mantiene uniti gli agenti sia la società che fornisce tutti i servizi richiesti per governare il loro comportamento.
In particolare, gli artefatti sono usati per mediare tra il singolo agente e il sistema (artefatti individuali), per costruire le istituzioni degli agenti (artefatti sociali) e per elevare le risorse del sistema al livello cognitivo degli agenti (artefatti risorsa).
In aggiunta a ciò, SODA mette a disposizione un meccanismo di zooming per modellare un sistema multi-agente a diversi livelli di astrazione: con l'operazione di *in-zooming* si passa da un livello più astratto ad uno più dettagliato, mentre con *out-zooming* si compie il passaggio inverso. A tale funzionamento si accompagna un meccanismo di proiezione che proietta da un livello al successivo le entità non soggette a zoom, in modo da preservare la consistenza interna di ogni livello.
Nel concreto, una modellazione SODA parte da un livello *core di base*, che viene completamente specificato, mentre gli altri strati ospitano solo le entità che hanno subito un processo di zoom e che sono state proiettate.
SODA è suddivisa in tre distinte fasi: la fase di analisi dei requisiti, la fase

di analisi e la fase di progettazione.
La *fase di analisi dei requisiti* si occupa della raccolta dei requisiti del sistema mediante l'impiego di tre astrazioni: i task, le funzioni e le dipendenze.
Un task è un'attività che richiede un certo numero di competenze per essere portata a termine, una funzione è un servizio a supporto dei task, mentre una dipendenza è un qualsiasi tipo di relazione che sussiste tra task, tra funzioni, o tra task e funzioni.
La *fase di analisi* si occupa dell'assegnamento a opportuni ruoli delle responsabilità delineate nella fase precedente.
Più precisamente, i task vengono assegnati a ruoli che saranno poi organizzati in gruppi, i quali andranno a comporre l'infrastruttura sociale del sistema.
Le funzioni vengono associate a risorse, che saranno poi assegnate a luoghi concettuali, definiti *place*, che compongono l'ambiente.
Le dipendenze danno origine a interazioni, modellate tramite opportuni protocolli, e vincoli: questi ultimi devono essere rispettati da tutte le entità che compongono il sistema, pena l'applicazione di una sanzione. Tali vincoli si distinguono in organizzativi se governano la struttura organizzativa, topologici se governano l'ambiente e di sicurezza se governano le politiche di sicurezza adottate nel sistema.
La *fase di progettazione* è una fase di sintesi del sistema dove le astrazioni chiave sono gli agenti, le società e gli artefatti. In particolare ad ogni agente vengono assegnati uno o più ruoli, nonchè l'artefatto individuale per la gestione di tutti i protocolli di interazione di cui l'agente può avvalersi. Un artefatto individuale opportunamente configurato permette di far rispettare i vincoli di sicurezza e impedisce che gli agenti eseguano azioni illecite. Ogni società di agenti viene quindi organizzata attorno ad uno o più artefatti sociali che inglobano le regole sociali derivate dai vincoli espressi nella precedente fase. Infine, ogni risorsa viene associata a uno o più artefatti-risorsa la cui interfaccia d'uso è stabilita dai protocolli di interazione; la collocazione di questi artefatti nel sistema è soggetta ai vincoli topologici specificati nella fase precedente.
Diversamente da altri approcci, SODA intenzionalmente non si interessa degli aspetti relativi alla struttura dei singoli agenti. Ciò sia perché SODA è concentrata sugli aspetti inter-agente, sia perché per tale scopo non è possibile avvalersi dell'ausilio di altre tecniche di progettazione come ad esempio l'impiego di strutture atte alla modellazione interna degli agenti di altre metodologie.

Capitolo 8

ANALISI AGENT DEVELOPMENT ENVIRONMENTS

8.1 Tecnologie esistenti

8.1.1 Jade

JADE, *Java Agent DEvelopment framework*, presentato da *Bellifemine* nel 2005, è un framework interamente implementato nel linguaggio JAVA. Esso semplifica lo sviluppo di sistemi multi-agenti attraverso un middleware che è conforme con le specifiche FIPA ed attraverso un set di graphical tools che supportano il debugging a la fase di distribuzione.
Un sistema basato su JADE può, infatti, essere distribuito su più macchine, le quali non hanno bisogno di condividere lo stesso OS e la cui configurazione può essere controllata tramite interfaccia remota GUI. La configurazione può anche essere cambiata a run-time tramite lo spostamento degli agenti tra una macchina e l'altra, come e quando richiesto.
Oltre all'agent abstraction, JADE prevede una semplice ma potente esecuzione dei compiti, denominati task, e un modello di composizione per realizzarli. Fornisce anche un'architettura di comunicazione *peer-to-peer* basata sul trasferimento di messaggi asincroni e molte altre caratteristiche vantaggiose. [15]
Esso supporta inoltre i cicli di vita dell'agente, la mobilità, la sicurezza e prevede servizi come *white and yellow pages* che possono essere usate dagli agenti per registrare e cercare i loro servizi. Tutti argomenti visti ed approfonditi nel capitolo 4.
Un agente viene creato estendendo una predefinita classe denominata sem-

plicemente Agent e ridefinendo i metodi di setup. Una volta creato, a tale agente viene assegnato un identificatore e viene registrato in un *Agent Management System*. A questo punto è inserito nello stato attivo e il suo metodo di setup viene eseguito. Quest'ultimo metodo è dunque il punto dove ogni attività degli agenti inizia.
Tutte le proprietà menzionate fanno parte del vasto sistema proposto da FIPA, è per questo motivo che JADE viene definito *FIPA-compliant*.
Gli agenti vengono eseguiti concorrentemente come differenti *pre-emptive Java threads*.
Inoltre è possibile definire dei comportamenti, tramite l'implementazione del modello BDI, per garantire agli agenti maggiori proprietà.
Il *JADE framework* viene sviluppato per applicazioni pratiche e industriali e viene rilasciato con dei set di graphical tools che possono essere utilizzati per controllare e monitorare l'esecuzione dei programmi multi-agenti.

8.1.2 Jason

Jason, proposto da *Bondini* nel 2007, viene introdotto come un interprete basato su *Java* per un'estensione di *AgentSpeak*, il quale è stato originariamente presentato da *Rao* nel 1996.
Jason distingue differentemente i concetti dei singoli agenti rispetto a quelli dei sistemi multi-agenti. Un *Individual Agent* in Jason è caratterizzato dai propri beliefs, plans and events ricevuti dall'ambiente o generati internamente.
Un piano in Jason è designato per uno specifico evento e contesto. L'esecuzione degli agenti individuali in Jason è controllata tramite un ciclo di operazioni, codificato nelle sue semantiche operazionali. In ogni ciclo vengono raccolti diversi eventi dall'ambiente e, dopo esserne stato selezionato uno, un piano viene generato e aggiunto all'*Intention Base* che, per concludere, ne consente l'esecuzione. [16]
Il *plan rule* in Jason indica che un piano dovrebbe essere generato da un agente se un evento è ricevuto/generato e l'agente ha determinati beliefs.
Jason è basato su una rappresentazione *first-order* per beliefs, events and plans.
Jason non ha costrutti di programmazione espliciti per implementare goals dichiarativi, sebbene questi ultimi possono essere simulati indirettamente tramite un pattern di piani.
Inoltre, beliefs e plans in Jason vengono annotati con informazioni addizionali che possono essere utilizzate in *belief queries* e *plan selection process*. [9]
L'ambiente di sviluppo di Jason prevede tools per monitorare l'esecuzione dei programmi multi-agenti.

Le tre componenti dell'architettura BDI vengono realizzate in *AgentSpeak* attraverso belief, goal and plan; vi è quindi un leggero scostamento da quelli che sono i pilastri originali della teoria BDI, infatti il significato è lievemente diverso, ma molto più pratico.

8.1.3 CArtAgO

CArtAgO, *Common ARTifact infrastructure for AGents Open environments*, è un framework/infrastruttura multiuso che rende possibile la programmazione e l'esecuzione di ambienti virtuali o software per la rappresentazione di sistemi multi-agenti.
CArtAgO è basato su *Agents and Artifacts*, A&A, un meta-modello per la modellazione ed la progettazione di sistemi multi-agenti. A&A introduce metafore di alto livello prese dalle cooperazioni umane presenti nell'ambiente: *Agents* sono entità computazionali che eseguono attività task/goal-oriented, in analogia al lavoro umano, e *Artifacts* che vengono visti come risorse di un ambiente dinamicamente costruito; utilizzate e manipolate dagli agenti per supportare/realizzare le loro individuali e collettive attività.
Attualmente A&A si basa su studi interdisciplinari che coinvolgono *Activity theory* e *Distributed Cognition* come concetti principali.
CArtAgO rende possibile lo sviluppo e l'esecuzione di ambienti basati sugli artefatti, strutturati in *workspaces* aperti e probabilmente distribuiti sulla rete, dove gli agenti di differenti piattaforme possono unirsi in modo da lavorare insieme dentro tale ambiente. A tal scopo gli sviluppatori di sistemi multi-agenti hanno finalmente un semplice modello di programmazione per progettare e programmare ambienti computazionali, composti da sets dinamici di artefatti. [7]
CArtAgO non è vincolato ad alcun *agent-model* specifico o ad alcuna piattaforma ma è decisamente utile ed effettivo quando integrato con un *Agent Programming Languages* basato su una forte nozione di agency - intelligent agents - in particolare quelli basati su BDI-architecture. Infatti l'ultima distribuzione di Cartago include un ponte per il linguaggio di programmazione Jason.
Visualizziamo ora brevemente alcuni punti principali che rendono interessante questo framework:

Workspaces - Un ambiente CArtAgO è dato da uno o numerosi workspaces, probabilmente diffusi su più nodi della rete. Molteplici workspaces possono essere eseguiti sullo stesso nodo. Per default ogni nodo possiede un default workspace. Per fare in modo che un agente lavori all'interno di un workspace, esso deve *unirsi* con tale. Una volta avviato, un agente si unisce automatica-

mente al workspace di default. Successivamente lo stesso agente può unirsi e lavorare simultaneamente in molti workspace.

Agents'action repertoire - Lavorando internamente ad un ambiente CArtAgO, il repertorio delle azioni di un agente viene determinato da un set di artifacts disponibili/utilizzabili nel workspace, in particolare dalle operazioni previste da ogni artifacts. Vi è una corrispondenza one-to-one tra le azioni e le operazioni: se vi è un artifatto che fornisce un'operazione myOp allora ogni agente interno al workspace può compiere un'azione esterna, chiamata myOp. In accordo, compiendo un'attività esterna, l'azione è completata con successo in corrispondenza al fatto che l'operazione sia completata con successo o meno. Dato che il set di artifacts può essere cambiato dinamicamente dagli agenti, creando nuovi artifacts o depositandone uno esistente, il repertorio delle azioni si può considerare anch'esso dinamico.

Default artifacts - Per default, ogni workspace contiene un set di base di *artifacts* predefiniti che prevedono il nucleo delle funzionalità degli agenti. In particolare:

- *Workspace artifact* [cartago.WorkspaceArtifact]: fornisce funzionalità per creare, smaltire, ricercare, collegare manufatti del workspace. Prevede anche operazioni per impostare ruoli e politiche relative al modello di sicurezza RBAC.
- *Node artifact* [cartago.NodeArtifact]: fornisce funzionalità per creare nuovi workspaces e per unirne locali o remoti.
- *Blackboard artifact* [cartago.tools.TupleSpace]: fornisce uno spazio di tuple che gli agenti possono sfruttare per comunicare e coordinare.
- *Console artifact* [cartago.toolls.Console]: prevede funzionalità per stampare messaggi su standard output.

8.1.4 Jack

Le applicazioni JACK consistono di una collezione di agenti autonomi che prendono input dall'ambiente e comunicano con gli altri. La costruzione di questo sistema prevede una vera e potente forma di incapsulamento. Ogni agente è definito in termini di propri goals, conoscienze e capacità sociali e viene successivamente lasciato all'esecuzione dei suoi automatismi funzionali nell'ambiente in cui risiede. Questo rappresenta una modalità reale ed effettiva che permette la costruzioni di applicazioni localizzate in ambienti dinamici e complessi - ogni agente è responsabile del raggiungimento dei

propri obiettivi, reagendo agli eventi e comunicando con le altre entità del sistema.
È bene evidenziare che l'abilità di ragionamento con beliefs e goals garantisce agli agenti di essere maggiormente flessibili nell'ottenere i loro compiti, nel senso che sono in grado di raggiungere obiettivi parzialmente e gradualmente. [9]
Non vi è la necessità di programmare esplicitamente le interazione dell'applicazione, infatti esse emergono come prodotto degli obiettivi individuali e delle capacità che possiedono gli agenti. [4]
JACK è un maturo ambiente cross-platform, presentato da *Winikoff* nel 2005, per costruire, eseguire e integrare sistemi multi-agenti commercial-grade. Esso è progettato su un modello BDI. In JACK, gli agenti sono definiti in termini dei loro beliefs (che cosa loro conoscono e cosa loro sanno fare), i loro desideri (quali obiettivi vogliono raggiungere) ed le loro interazioni (per gli obiettivi che sono attualmente impegnati a raggiungere).
Il *JACK AgentLanguage* è costruito su Java, esso fa più di estendere tali funzionalità, esso prevede un framework per supportare un intero nuovo paradigma di programmazione. Il JACK AgentLanguage è un linguaggio Agent-Oriented e viene utilizzato per implementare i sistemi software orientati agli agenti. [19]
Tra gli aspetti più interessanti del paradigma di JACK vi sono sei costrutti principali, quali:

- *Agent*: Il costrutto Agent viene utilizzato per definire i behaviours degli agenti software. Questo include le capacità che un agente possiede, a quale tipo di messaggi ed eventi risponde e quali piani esegue per raggiungere i suoi obiettivi.

- *Capability*: Questo costrutto permette ai componenti funzionali che costituiscono un agente di essere aggregati e riutilizzati. Una capacità può essere composta da piani, eventi, beliefsets e altre capacità che assieme permettono di definire delle abilità complesse. Un agente a turno può sfruttare le capacità e gli attributi che ne derivano.

- *Beliefset*: Il costrutto beliefset contiene i beliefs degli agenti, rappresentati con un modello relazionale generico. Esso è stato specificatamente progettato per svolgere delle query usando *Logical Members*. Logical Member sono come normali *Data Members* ad eccezione del fatto che seguono regole di programmazione logica.

- *View*: Il costrutto view permette *queries general-purpose* che possono essere svolte su un *underlying data model*. Tale modello potrebbe essere implementato utilizzando diversi beliefsets o altre strutture dati Java.

- *Event*: Il costrutto event descrive un'occorrenza in risposta a quali agenti devono prendere parte ad un'azione.

- *Plan*: I piani di un agente sono analoghi alle funzioni, rappresentano le istruzioni che l'agente deve provare ad eseguire in relazione ai suoi obiettivi e permettere la gestione degli eventi designati.

Ogni events, plans e beliefsets sono implementati come normali classi Java. Essi ereditano alcune proprietà fondamentali di una classe base e ne estendono le capacità per conseguire in maniera opportuna ai loro compiti. Le classi base sono definite internamente nel kernel e costituiscono una colla che permette di tenere assieme un programma agent-oriented JACK.
Il JACK Agent Language è più che una specifica organizzazione di oggetti Java e strutture di ereditarietà, esso prevede una sua propria sintassi estesa che non ha rappresentazione analoga in Java.
JACK inoltre viene integrato con altri ambienti di sviluppo e prevede anche il *monitoring and logging* dei vari sistemi.

8.1.5 Orleans

Orleans è un modello di programmazione a run-time per la costruzione di servizi *cloud-native*, cioè nati per la distribuzione sul cloud. Viene introdotto da Microsoft per migliorare l'utilizzo del servizio di clouding previsto per i giochi online; tra i quali anche *Halo* (molto conosciuto per la console Xbox). [28]
Accademicamente viene visto come un modello di attori virtuali distribuiti. I motivi di utilizzo di tale scelta sono principalmente due:

- Produttività dal punto di vista dello sviluppatore:

    - Concorrenza, distribuzione, tolleranza ai guasti, gestione delle risorse, dominio di sistemi distribuiti, minore scrittura di codice, aiuto agli sviluppatori desktop e altro.

- Scalabilità per default:

    - I fallimenti nella scalabilità potrebbero essere fatali per il proprio business, il codice dovrebbe essere *scale-proof* e garantire la scalabilità fuori dal *rewriting*.

Ogni classe ha una chiave, i quali valori identificano le istanze game, player, phone, device o altro. Per invocare un attore A, il chiamante passa la chiave alla sua classe locale e riceve come risultato la referenza all'attore richiesto.

Invoca poi un metodo di tale referenza; ogni invocazione è asincrona. [25]
Le istanze degli attori esistono sempre, al massimo vengono aggiunte virtualmente, le applicazioni non li creano e neanche li rimuovono, essi sono sempre presenti e non mancano in nessun arco del loro ciclo di vita; inoltre il codice può sempre invocare metodi su un attore, indipendentemente da quale sia stato chiamato.

L'adozione di questo modello porta diversi vantaggi dal punto di vista applicativo, quali il recovery trasparente da fallimenti, ad esempio server; il ciclo di vita viene gestito a run-time; a tempo si esecuzione si possono creare attivazioni di attori senza stato. Trasparenza alla posizione, gli attori possono passare le referenze di un altro o le loro persistenze. Le referenze sono logiche, virtuali e sempre valide ma non legate ad un'attivazione specifica.

8.1.6 Cougaar

Cougaar, COgnitive Agent ARchitecture, è un'architettura Java per la costruzione in larga scala di applicazioni distribuite basate sugli agenti. È il prodotto di molti anni di programmi di ricerca DARPA, denominate *Ultra-Log*, riguardanti sistemi ad agenti distribuiti in larga-scala. I programmi vengono suddivisi in due sezioni:
Il primo, una volta concluso, dimostra la fattibilità nell'utilizzo della tecnologia basata sugli agenti per condurre in maniera rapida e su larga scala, forme di pianificazione e di ripianificazione di logiche distribuite. [37]
Il secondo programma sviluppa tecnologie d'informazione per migliorare le possibilità di sopravvivenza di questi sistemi distribuiti che operano in ambienti estremamente caotici. L'architettura risultante, prevede sviluppi con framework per implementare applicazioni agente distribuiti in larga scala ma con minime considerazioni per l'architettura e l'infrastruttura sottostante. L'architettura Cougaar utilizza la più recente progettazione basata su componenti agent-oriented e garantisce una lunga lista di caratteristiche. [32]
L'astrazione degli agenti Cougaar include diversi servizi avanzati, come:

- *Blackboard publish/subscribe* per la comunicazione interna ed esterna all'agente stesso e tra gli altri agenti.

- *Http servlet* engine per UIs based.

- *Knowledge representation system*: logistics assets, FrameSets.

- Coordinazione tra agenti attraverso meccanismi di coordination-slate, assignments.

Agenti Cougaar vengono eseguiti su un nodo Cougaar, in Java Virtual Machine, il quale viene eseguito a sua volta su un host. Tale agente concorda con uno o più plugins, che definiscono il suo behaviour; come si può immaginare, un agent con zero plugin non fa nulla.
Diversamente da altre architetture basate interamente sui messaggi, i cougaar blackboard plugins sono basati su principali data managers. I plugins reagiscono alle informazioni sulla blackboard aggiungendo, cambiando, rimuovendo notifiche. La piattaforma Cougaar trasforma le informazioni dalla blackboard in operazioni inter-dealer, ma questo viene nascosto dalle API dello sviluppatore. Tutti i plugins di coordinazione sono implementati attraverso asynchronous data subscriptions. L'infrastruttura della blackboard aggiunge, cambia, elimina le notifiche batch, fornendo una maggiore robustezza e scalabilità al sistema. L'intero stato è memorizzato nella blackboard, la quale prevede meccanismi di supporto per i recovery falliti. Il nucleo dei servizi sono le implementazioni Cougaar component-based, mobilità degli agenti tra nodi, la persistenza degli stati agente ed il successivo recovery dopo un problema di crash o un supporto di trasporto alla messaggistica che prevede differenti protocolli. [1]

8.1.7 TuCSoN

TuCSoN, *Tuple Centres Spread over the Network*, è una libreria Java che prevede la coordinazione come un servizio di agenti Java e *tuProlog*. TuCSoN è un modello per la coordinazione di processi distribuiti, autonomi, intelligenti e mobili da qui il nome che, tradotto correttamente indica: "centri di tuple che si espandono in rete".
Sfrutta i centri di tuple come suo media di coordinazione, il quale spazio di tuple accresce le nozioni di comportamento grazie al linguaggio Respect. La tecnologia di coordinazione implementa il modello come un middleware distribuito, disponibile sotto GNU LGPL. [30]
Arrivando alla conclusione stretta che Tucson è un modello per la coordinazione di processi distribuiti, con agenti autonomi e mobili.
Comunemente parlando si potrebbe affermare:

Un sistema TuCSoN è una collezione di agenti e centri di tuple che lavorano assieme in un set di nodi possibilmente distribuiti.

Visualizzando le entità di base si può affermare che la piattaforma consente di coordinare le richieste degli agenti in maniera efficiente senza creare intralci al sistema. Essa è composta da:

- *TuCSoN Agents*: i cosidetti *coordinables*, cioè entità coordinabili, in grado di cooperare, sincronizzarsi e competere all'interno di uno spazio di coordinazione.

- *Centri di tuple Respect*: di default il *coordination media*.

- *TuCSoN Nodes*: rappresentano la *topological abstraction* di base, la quale ospita i centri di tuple e gli agenti. Concettualmente corrispondono a server collegati in rete che permettono di definire la topologia della rete stessa. Ogni nodo è identificato tramite una coppia di valori.

Gli agenti, i nodi e i centri di tupla rappresentano un sistema unico. Dal momento in cui agenti sono entità pro-active, ed i centri di tuple sono entità reactive, la coordinazione necessita di operazioni di coordinazione in modo da agire sul coordination media: tutte le operazioni sono costruite tramite il *TuCSoN coordination language*.
Agenti interagiscono scambiandosi tuples attraverso centri di tuple e usando le TuCSoN coordination primitives (operazioni primitive di coordinazioni), completamente definite dal linguaggio. Centri di tuple prevedono la condivisione dello spazio per la comunicazione basata sulle tuple (tuple space), insieme con behaviour space per la coordinazione basata su tuple (specification space).
Aspetto interessante riguarda il concetto di mobilità, gli agenti potrebbero in principio muoversi indipendentemente dal device dove sono eseguiti.

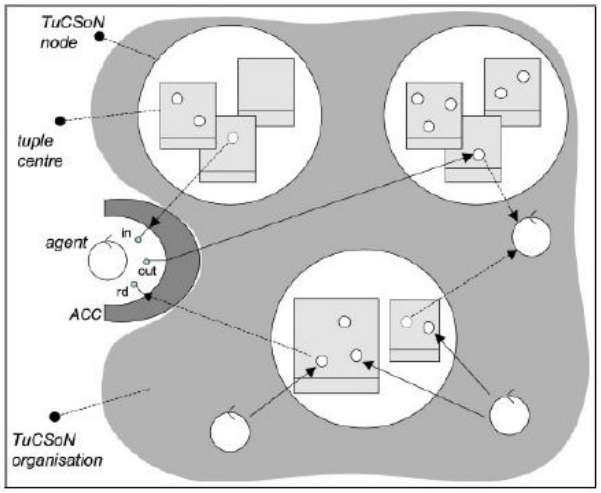

Figura 8.1: Tucson Architecture

I centri di tuple sono essenzialmente associati ad un device, possibilmente mobile, così la mobilità del centro di tuple è dipendente sul loro device ospitante. [3]
Continuiamo adesso l'analisi del modello TuCSoN incentrandoci sull'architettura del sistema. Un sistema TuCSoN è prima di tutto caratterizzato da una collezione, possibilmente distribuita, di nodi ospitanti un servizio. Un nodo è caratterizzato da un device che lavora sul network ospitandone il servizio e da una porta network dove il servizio ascolta la richiesta in ingresso. Molti nodi TuCSoN possono in principio correre nello stesso device sulla rete, ognuno in ascolto su una porta differente. La porta di default è la 20504, così un agente può invocare l'operazione:

```
tname @ netid ? op
```

Figura 8.2: Tucson Invocation

senza specificare il nome della porta si sta a significare che gli agenti intendono invocare l'operazione *op* sul centro di tuple *tname* del nodo di *default netid: 20504* ospitato sul device in rete netid.

Lo spazio di coordinazione di un nodo TuCSoN è definito come una collezione di tutti i centri di tuple ammissibili, ognuno denominato con un nome univoco. Qualunque nodo TuCSoN prevede agenti con un completo spazio di coordinazione, così che in principio qualsiasi operazione di coordination può essere invocata su un qualsiasi centro di tuple ammissibile presente in un qualsiasi nodo.
Il centro di tuple di default di ogni nodo TuCSoN è chiamato *default*. Ogni nodo TuCSoN definisce un *default tuple centre*, il quale risponde a qualsiasi operazione ricevuta dal nodo che non specifica il target del centro di tuple.

8.1.8 Astra

Astra è un linguaggio di programmazione ad agenti, costruito ed integrato con Java. Astra rappresenta un'implementazione dell'AgentSpeak(TR) - un linguaggio di programmazione logica basata sugli agenti che combina AgentSpeak(L), implementazione di Jason, con funzioni *Teleo-Reactive*. Gli obiettivi chiave nella progettazione sono:

- Minimizzare il gap tra Astra e Java.

- Fornire un linguaggio agente con una curva di apprendimento ridotta.

- Incentivare l'integrazione con altre tecnologie.

- Abilitare l'effettivo riutilizzo del codice.

Come specificato precedentemente Astra è un'implementazione di AgentSpeak(TR), linguaggio proposto da *Keith Clark*, per integrare l'unione di specifiche che caratterizzano i due differenti linguaggi.
AgentSpeak(L) ha dimostrato di essere un buon linguaggio per specifici *goal-oriented behaviours*, mentre la programmazione in Teleo-Reactive si è dimostrata efficiente nella realizzazione di *reactive behaviours* in ambienti molto dinamici. Astra viene vista quindi come un'implementazione alternativa di AgentSpeak(L) che introduce un range di funzionalità aggiuntive adatte ad una traduzione più pratica degli *Agent Programming Language*.
In termini di beliefs, goals ed eventi la principale caratteristica che differenzia Astra riguarda il fatto che i termini e le variabili sono basati sul concetto di tipo. Il sistema dei tipi in Astra si accosta a quello di Java, migliorando la transazione, rendendola più semplice e trasparente. [11]
Questo significa che indicando il belief che un agente è vivo, esso potrebbe essere rappresentato come:

state("alive")

da notare l'alto livello del codice e l'argomento trascritto come stringa.
In similitudine, gli obiettivi (goals) sono gli stessi di AgentSpeak con la solita eccezione dei tipi. Infatti un agente che gioca a calcio, nel momento in cui attacca, per esempio, dovrebbe avere l'obiettivo di realizzare un goal, che potrebbe essere rappresentato da:

!score("goal")

Per concludere si indicano gli eventi con un'implementazione uguale ad AgentSpeak(L), per esempio l'evento che corrisponde all'adozione del goal. Dal momento in cui tutti i termini sono basati sui tipi, anche le variabili lo devono essere.

Aspetto molto interessante in Astra riguarda la pianificazione di regole, definita da una sintassi differente rispetto a quella utilizzata in AgentSpeak(L). Specificatamente, questa sintassi cerca di esser più vicina a quella utilizzata nei linguaggi Java e C. La parte di *plan body* di una regola viene implementata come un blocco di codice, formato da una seguenza di istruzioni primitive, dal controllo degli stati del flusso e da sottoblocchi.
Astra si differenzia da AgentSpeak(L) nei seguenti modi:

- programmi Agent sono organizzati in classi Agent che possono essere estesi su modelli di interazioni multiple.

- le azioni primitive utilizzate devono essere dichiarate esplicitamente nel programma Agent.

- programmi Astra possono referenziare classi Java.

- in aggiunta di *plan rules*, i programmi Astra includono piani parziali, chiamati *plan bodies*, in maniera di migliorare la riusabilità delle classi e del codice.

- gli obiettivi di Astra sono la familiarizzazione per coloro che conoscono *Mainstream Programming Languages*, in maniera parlicolare Java.

Il risultato è che i programmi sono leggermente più strutturati di AgentSpeak(L). In contrasto con altri linguaggi di programmazione, gli sviluppatori di Astra non hanno intenzione di prevedere un debugger per Astra stesso. Essi non la considerano una svista ma una vera e propria caratteristica, perché l'esecuzione dei debugger potrebbero effettivamente ostacolare la comprensione del codice (in particolare per gli sviluppatori alle prime armi) ed aumentare la complessità percepita. Promuovendo perciò gli approcci più tradizionali, come stampe da console e logging tools.

8.2 Confronto sullo stato

A questo punto, dopo aver svolto una panoramica generale sugli ambienti di sviluppo, possiamo immergerci nel mondo reale, per osservare come sono utilizzate queste tecnologie e in quali settori di lavoro. Per fare ciò ci avvaliamo dell'ausilio di alcuni sondaggi eseguiti da esperti e rilasciati in diversi articoli.
Nel grafo a torta presentato viene mostrata la distribuzione delle applicazioni agent-based in relazione al settore di appartenenza. Come si evince dall'os-

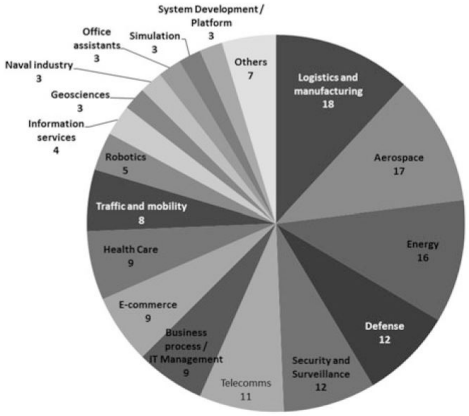

Figura 8.3: Number of applications in different sectors

servazione, undici settori lavorativi rivestono l'86% di tutte le applicazioni multi-agenti. Dove ben i migliori sei, come la logistica, l'energia, la difesa, la sicurezza, la telecomunicazione e la sorveglianza ne ricoprono il 60% circa. Questo indica che solo una piccolissima parte di tutte le industrie al mondo, nei differenti settori, adottano tecnologie agent-based.
Osservando ancora il grafo si può notare che la logistica, la produzione e la telecomunicazione sono i domini che globalmente hanno il più alto numero di programmi sviluppati sugli agenti, mentre l'energia, la sicurezza e la sorveglianza o difesa, appaiono ancora con applicazioni emergenti e con un piccolo impatto nel mondo.

Vediamo allora una breve caratterizzazione di questi settori, andando ad analizzare solo i principali ed osservando per quali scopi vengono utilizzate le tecnologie ad agenti.

Manufacturing & Logistics
È un'interessante area di applicazione per le ricerche agent-oriented fin dai primi inizi degli anni '80, che prevede un raggio abbastanza ampio di produzione. Partendo da piani e controlli, a compiti di allocazione, fino ad arrivare alla negoziazione e simulazione di mercato dei prodotti. Commercialmente parlando una delle principali aziende in questo settore è la *Whitestein Tecnologies*.

Security & Surveillance
La sicurezza è un concetto basilare per ogni dominio di applicazione. Internet richiede necessariamente fondamenti di sicurezza, in modo chiaro e trasparente a tutti i membri che ne fanno uso. La ricerca ad agenti offre interessanti impostazioni nel quale le soluzioni teoriche possono essere implementate e dimostrare. La sorveglianza incrementa l'idea di sicurezza con un ulteriore livello. Nella società moderna, dove le reti wireless si distribuiscono in maniera sempre più veloce, la sorveglianza prende ampiamente campo in tutti i casi in cui non occorre aumentare la sicurezza, ma risolvere problemi per quanto riguarda la privacy. Ed essendo il dominio di applicazione naturalmente distribuito, esso è un settore ideale per le tecnologie ad agenti.

Telecomunications
Le società di telecomunicazioni sono state interessate e coinvolte nella ricerca ad agenti fin dalle fasi iniziali. Al giorno d'oggi, gli smartphone sono diventati dei veri sistemi complessi, in cui possono essere eseguite applicazioni MAS e possono essere adottate forme di interazione con applicazioni simili su altri dispositivi. È quindi molto probabile una spinta di tali programmi nel prossimo futuro, per migliorarne il servizio e facilitarne l'utilizzo.

Ci interessiamo adesso ad osservare come sono composte queste applicazioni adottate nei differenti ambiti lavorativi, cioè quale linguaggio di programmazione viene impiegato e se vengono utilizzate delle *Agent Platform* per migliorare la progettazione e lo sviluppo.
Java è di gran lunga il più popolare linguaggio di programmazione impiegato, seguito da C, C++ e C# per poi concludere con PHP e Python. Questi quattro gruppi rappresentano il 75% delle applicazioni. Inoltre alcuni programmi utilizzano più di un linguaggio. Nel grafo a torta seguente viene illustrata la copertura delle piattaforme ad agenti utilizzate nelle applicazioni.
Possiamo fare a questo punto un paio di osservazioni; prima di tutto una grande maggioranza non utilizza alcuna piattaforma dedicata o strumenti di sviluppo. In secondo luogo, quelle più comuni sono Jade, AOS's Jack, CoJack and C-BDI. Considerando che WADE è un'estensione del più famoso

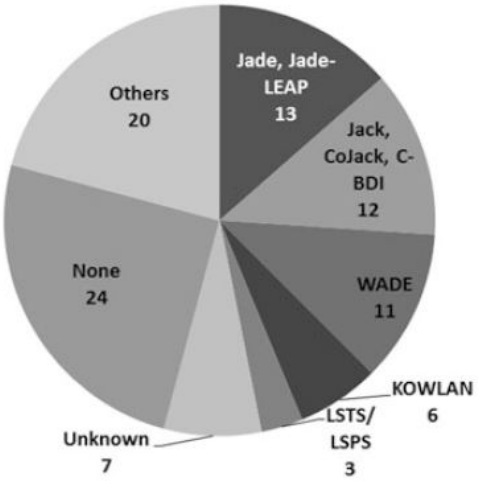

Figura 8.4: Usage of Agent Platforms

ambiente JADE, possiamo concludere che complessivamente JADE è il più utilizzato.
Come ultimo argomento possiamo affermare che, pur non essendo visibile dal grafo, delle applicazioni che non utilizzano le piattaforme ad agenti per sviluppare le tecnologie, solo una piccola parte sono considerate con un livello di *maturità*, cioè egregiamente strutturate e progettate, elevato.
Mentre per le restanti, che adottano quindi, seppur con differenti ambienti, delle piattaforme ad agenti, la maggior parte delle applicazioni sono considerate con livelli di maturità eccellenti. Questo viene indicato per riflettere l'elevata importanza e ruolo che adottano le aziende come Telecom Italia, Telefonica e altre. Tutte quelle perciò che stanno applicando e sviluppando le tecnologie basate sugli agenti per la costruzione di applicazioni distribuite e che adottano opportuni ambienti di lavoro per migliorarne la qualità. Concludendo che, le piattaforme ad agenti dedicati possono fare la differenza per quanto riguarda il successo del proprio business.

Una maggiore indicazione dell'impatto di una specifica tecnologia è il numero di imprese che costruiscono business di successo dalla vendita dei servizi o prodotti basati sulle tecniche ad agenti. Nel seguito sono presentate compagnie dove il loro business è riconosciuto per la costruzione di questo tipo di tecnologie. Evidenziamo il fatto che tra le aziende non sono indicate le principali di telecomunicazione, come Telecom Italia o Telefonica, perché non basano il loro business sulle tecnologie ad agenti, pur comunque adottandole.

Whitestein Technologies: offrono soluzioni agent-based per la gestione ed esecuzione dei processi nelle aree di finanza, produzione, telecomunicazione o logistica. Addizionalmente anche servizi di ottimizzazione del controllo e gestione.

Agent-oriented Software (AOS): sostiene di essere l'azienda leader per la fornitura di sistemi autonomi o semi-autonomi. AOS prevede piattaforme e strumenti di sviluppo per la progettazione di sistemi basati sugli agenti, tra i più conosciuti citiamo Jack e CoJack, per domini di applicazioni dedicati come l'assistenza alla sorveglianza ed al controllo in industrie di estrazioni di gas o olio, dove l'imprevedibilità dell'ambiente è un aspetto da non sottovalutare.

Agentis Software: startup fondata dai membri dell'*Australian Artificial Intelligence Institute*, AAII. L'obiettivo principale è l'applicazione dei concetti BDI per la gestione ed esecuzione di processi business.

Concludiamo questo paragrafo con un'incetivazione verso il futuro. Pur avendo presentato diverse aziende, applicazioni e tecnologie, il mondo dei sistemi basati sugli agenti è ancora molto acerbo.
Nell'esempio le applicazioni vengono suddivise in accordo con le caratteristiche del settore in cui sono state trattate, cioè facendo distinzioni tra quelle sviluppate e progettate da industrie o organizzazioni governative, applicazioni sviluppate in università o facoltà specializzate e per concludere applicazioni date dalla cooperazione tra entrambi i settori, industriali e scolastici.
Circa il 31% dei programmi sono stati esclusivamente progettati dalle industrie di competenza dei vari ambiti, mentre il 28% da università e il 41% dalla cooperazione di entrambe. Questo ci permette di arrivare alla conclusione che ancora oggi le università hanno un alto livello di partecipazione, arrivando al circa 70%. Pur sembrando un notevole vantaggio, riguardante il fatto che gli studenti si interessano a tecnologie ancora in totale espansione, da un ulteriore lato è un problema rilevante. Le industrie non adottano tecnologie emergenti, preferiscono avere sicurezze in quelle passate che possibili migliorie nelle future. Occorre quindi incentivare la produzione e lo sviluppo da parte del settore industriale, non solo per migliorare questa stessa tecnologia ma anche per garantire vantaggi alla sua intera economia. [21]

Capitolo 9

CONCLUSIONI

9.1 Conclusioni

Grazie ad alcune interessanti caratteristiche della programmazione orientata agli agenti, negli ultimi anni si è presentato un incremento sempre maggiore per quanto riguarda l'attività di ricerca su questa tecnologia, e questo fatto è testimoniato dal crescente numero di linguaggi, tools e piattaforme per lo sviluppo di MAS; non solo negli argomenti trattati in questo documento.
Attualmente in ambito industriale la tecnologia ad agenti ha poco margine di manovra, ma in un futuro non molto remoto essa darà sicuramente il suo contributo in maniera massiva.
Come si è potuto apprendere dalla lettura di questo elaborato la tecnologia ad agenti offre innumerevoli possibilità di progettazione per sistemi complessi, utilizzando un livello di astrazione sempre più elevato. Si utilizza infatti una descrizione del sistema in termini di obiettivi e intenzioni nell'ambiente piuttosto che in termini di operazioni e proprietà. Ritenendo, quindi, questa tecnologia come una potente soluzione futura per le prossime generazioni di sistemi autonomi.
Occorre continuare a promuovere e contribuire alla mentalità agent-oriented, lavorando su tecniche e strumenti che supportano tutte le fasi di progettazioni, garantendone la facilità d'uso, la scalabilità e adeguate performance.
In conclusione, sembra particolarmente utile fare un maggiore sforzo sull'integrazione di metodologie e linguaggi di programmazione agent-based.
Come ritengono molti esperti, appunto, gli agenti sono la giusta astrazione per re-integrare varie sub-discipline AI, *Artificial Intelligence*. Per la modellazione di sistemi molto complessi come i robots. [17]
Infatti la futura applicazione di questa tecnologia andrà a coprire tutti quei sistemi che richiedono un certo grado di automazione e intelligenza.

Auspico che chiunque abbia letto e cercato di assimilare questo documento sia riuscito a comprenderne i concetti cardine, nonché la sua importanza nell'informatica del futuro. Mi auguro inoltre che sia stato di piacevole lettura e ringrazio per l'interesse mostrato.

9.2 Ringraziamenti

Al termine di questo percorso desidero innanzittuto ringraziare il mio relatore, il professore Andrea Omicini per la professionalità e la simpatia, per i messaggi a tarda notte e per l'aiuto. Al correlatore, l'ingegnere Stefano Mariani, per il supporto e la disponibilità.

Alla mia famiglia, mia madre che mi ha sostenuto e mio padre che mi ha incentivato. Senza di loro, ora di certo non sarei qui!

Un immenso ringraziamento è rivolto alla mia ragazza, Beatrice, per essere stata al mio fianco, per avermi sollevato il morale nei momenti di difficoltà e per l'infinita pazienza. Grazie!

Elenco delle figure

Bibliografia

[1] Wikipedia - cougaar [https://en.wikipedia.org/wiki/cougaar], 2016.

[2] Andrea Omicini Ambra Molesini, Enrico Denti. Metodologie per l'ingegneria del software: approccio ad agenti.

[3] Stefano Mariani Andrea Omicini. The tucson coordination model and technology. 2015.

[4] Jack Developers [aosgrp.com/products/jack]. Jack - autonomous decision-making software, 2015.

[5] Grady Booch. *Object oriented analysis and design with application*. Pearson Education India, p.17, 2006.

[6] Michael Bratman. Intention, plans, and practical reason. 1987.

[7] CArtAgO Developers [cartago.sourceforge.net]. About cartago, 2010.

[8] Paolo Ciancarini. Ingegneria del software.

[9] Mehdi Dastani. Programming multi-agent systems. 2015.

[10] Daniel Clement Dennett. *The intentional stance*. MIT press, 1989.

[11] Astra Developers. Astra: Agentspeak(tr) agents [astralanguage.com], 2016.

[12] ACL Fipa. Fipa - acl message structure specification. *Foundation for Intelligent Physical Agents*, 2002.

[13] Koen V Hindriks. The shaping of the agent-oriented mindset. 2014.

[14] John Horberg. Talk to my agent: software agents in virtual reality. *Computer–Mediated Communication Magazine*, 1995.

[15] Jade Developers [jade.tilab.com]. Java agent development framework, 2016.

[16] Jason Developers [jason.sourceforge.net/wp/description]. Jason description.

[17] Nicholas R Jennings. Agent-oriented software engineering. 1999.

[18] Nicholas R Jennings. On agent-based software engineering. *Artificial intelligence*, 2000.

[19] Agent Oriented Software Pty. Ltd. *JACK Intelligent Agents - Agent Manual*. 2009.

[20] Jorg Muller, Munindar P Singh, and Anand S Rao. *Intelligent Agents V. Agents Theories, Architectures, and Languages*. Springer, 2007.

[21] Jorg P Muller and Klaus Fischer. Application impact of multi-agent systems and technologies: a survey. 2014.

[22] Greg MP O'Hare and Nick Jennings. *Foundations of distributed artificial intelligence*, volume 9. John Wiley & Sons, 1996.

[23] Andrea Omicini. On autonomy definitions and acceptations, 2015/2016.

[24] H.V.D. Parunak. *Industrial and practical applications of DAI, in: G.Weiss(Ed.), Multi-Agent System*. MIT Press, 1999.

[25] Alan Geller Philip A. Bernstein, Sergey Bykov. Project orleans - distributed virtual actors for programmability and scalability. 2014.

[26] Stefan Poslad, Phil Buckle, and Rob Hadingham. The fipa-os agent platform: Open source for open standards. In *proceedings of the 5th international conference and exhibition on the practical application of intelligent agents and multi-agents*, volume 355, page 368, 2000.

[27] Anand S Rao and Michael P Georgeff. Modeling rational agents within a bdi-architecture. 1991.

[28] Microsoft Research [research.microsoft.com/en us/projects/orleans/]. Orleans: a platform for cloud computing.

[29] Aladdin Ayesh Rula K.Al-Azawi. Comparing agent-oriented programming and object-oriented programming. 2013.

[30] A. Omicini S. Mariani. Tucson home [https://apice.unibo.it/xwiki/bin/view/tucson/].

[31] John R Searle. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press, 1969.

[32] Raytheon BBN Technologies. Cougaar documentation [cougaar.org], 2013.

[33] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[34] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[35] Michael Wooldridge, Nicholas R Jennings, et al. Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152, 1995.

[36] Michael J Wooldridge. *Reasoning about rational agents*. MIT press, 2000.

[37] Todd Wright. *CougaarOverview*. 2007.

[38] Y.Shoham. *Agent-oriented Programming - Artificial Intelligence*. 1993.

[39] Franco Zambonelli. Agent-oriented software engineering. 2010.