UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Un Sistema Interattivo basato sui Design Pattern per la Progettazione di Applicazioni Multi Agente

RELATORE
Chiar.mo Prof. Carlo Ferrari

CANDIDATO
Cognolato Elisabetta
Matricola 602114

ANNO ACCADEMICO 2010-2011

Indice

Introduzione

Questa tesi tratta la progettazione e l'implementazione di un sistema interattivo di appoggio per la realizzazione di applicazioni basate su sistemi multi agente.

Il sistema realizzato è propriamente un framework che mira a fornire al programmatore uno strumento di sviluppo. Per strumento di sviluppo si intende un programma, dotato di interfaccia grafica, che assiste l'utente nella fase di analisi e progettazione di un'applicazione; ne esistono di diversi tipi, dai tool più elementari fino a veri e propri strumenti CASE. Questo sistema in particolare fornisce in modo automatico parti di codice suggerite e/o necessarie per l'implementazione e un modulo grafico per schematizzare le interazioni che dovranno avere luogo nel progetto finale.

Un framework di norma fornisce, in aggiunta agli strumenti di sviluppo, anche una libreria di partenza per l'implementazione e degli strumenti di runtime finalizzati alla fase di deployment. Il sistema descritto in questa tesi si limita a fornire le API ovvero le librerie di classi, per le funzionalità specifiche degli agenti, ma non sviluppa un ambiente per l'esecuzione degli stessi e, di conseguenza, non fornisce strumenti di monitoraggio, controllo e debugging. Questo perché l'obiettivo principale è quello di supportare il programmatore nelle fasi iniziali di modellizzazione e progettazione, aspetti che non sono stati ancora implementati da nessuna piattaforma ad agenti. A tal proposito al sistema interattivo viene in appoggio la piattaforma JADE, utilizzata come tool di ausilio per il deployment.

Le applicazioni che saranno oggetto del sistema interattivo si baseranno su un'architettura che utilizza come strategia la tecnologia ad agenti mobili. In questa tesi verrà trattato il connubio tra il paradigma ad agenti, come evoluzione attiva del paradigma ad oggetti, e la mobilità, caratteristica delle reti Mobile Ad-hoc NETworks (MANET) oggetto di discussioni e studi in questi ultimi anni di sviluppo tecnologico.

La tesi è strutturata, oltre a questa sezione introduttiva in sette capitoli.

Di seguito viene riportata una breve descrizione dei capitoli redatti:

- *Il capitolo 1 presenta le caratteristiche delle mobile ad-hoc networks e dei dispositivi che le compongono per offrire una panoramica della letteratura e degli obiettivi che questa tesi si propone;*
- *Il capitolo 2 riporta le caratteristiche principali dei sistemi multi agente, in particolare della tecnologia ad agenti mobili e le piattaforme di sviluppo che supportano la mobilità;*
- *Il capitolo 3 presenta la categorizzazione semantica degli agenti che vengono ontologicamente definiti a seconda delle loro capacità e funzionalità;*
- *Il capitolo 4 presenta i design pattern propri della tecnologia ad agenti mobili applicata alle reti mobili ad hoc;*
- *Il capitolo 5 espone la fase di modellizazione del sistema creato, ed in particolare vengono definiti i prototipi realizzati e i pattern creati;*
- *Il capitolo 6 tratta del sistema nel dettaglio dal punto di vista dell'implementazione e delle funzionalità fornite per il supporto alla progettazione;*
- *Il capitolo 7 espone le conclusioni del lavoro di tesi e vengono presentati spunti e sviluppi futuri per quanto riguarda gli agenti mobili, il framework sviluppato e le reti mobili stesse.*

Durante gli studi della tecnologia ad agenti mobili e la progettazione del sistema interattivo si è riscontrato l'effettivo bisogno di un framework con le funzionalità proposte, data la complessità degli stessi e dalla mancanza di strumenti completi per il loro sviluppo. Si è potuto verificare in questo modo la bontà del progetto nel suo complesso e del lavoro di ricerca e analisi iniziale che trova riscontro nei prototipi di cui il framework è dotato.

1 Mobile Ad-hoc Networks

In questo capitolo viene fornita un' introduzione alle reti mobili ad-hoc e alle nuove problematiche che hanno portato alla luce.

In particolare vengono evidenziati i requisiti che queste reti devono rispettare e la caratteristiche dei dispositivi che vi accedono.

1.1 Introduzione

La necessità e la voglia di aumentare le prestazioni delle tecnologie informatiche si sta concentrando notevolmente nello sviluppo di applicazioni SMART. Si tratta di applicazioni costituite da componenti software intelligenti in grado di svolgere autonomamente compiti specifici delegati dall'utente, e di acquisire un proprio potere decisionale facendo esperienza nell'ambiente in cui si trovano. Per '*fare esperienza*' s'intende la raccolta d'informazioni che un componente software sperimenta durante il suo ciclo di vita; il tracciamento di queste informazioni e l'elaborazione di dati statistici inerenti favoriscono il processo di conoscenza di un applicativo che lo porta ad assumere competenze specifiche in relazione alle condizioni in cui deve operare. A seconda di ciò che si vuole conoscere e del tipo di feedback disponibile, l'apprendimento può assumere forme diverse. Per esempio, si può apprendere l'utilità di un comportamento in base agli effetti che produce, acquisire la capacità di categorizzare ontologicamente un ambiente a partire da una serie di condizioni, o imparare a riconoscere pattern o schemi in assenza di informazioni sull'output.

Questo ha portato alla rapida crescita di sistemi operativi multitasking, di ambienti distribuiti e tecnologie wireless associate allo sviluppo di dispositivi di taglia sempre più ridotta, che hanno favorito la diffusione del cosiddetto mobile computing.

Ogni utente, dotato di un dispositivo portatile, è in grado di accedere ad applicazioni e a servizi distribuiti in qualsiasi modo, in qualsiasi momento e posizione, grazie alla connettività, alimentata da moderne tecnologie di rete.

La maturità delle reti wireless e la popolarità di questi dispositivi hanno permesso lo sviluppo di queste reti wireless ad-hoc. Una rete di questo tipo consiste in un

insieme di hosts mobili che operano senza l'aiuto di una infrastruttura stabile e statica gestita da un dominio di amministrazione centrale. Le comunicazioni tra i dispositivi sono realizzate dai link wireless attraverso trasmissioni radio captate dalle antenne dei dispositivi stessi. Le reti assumono capacità di riconfigurazione a seconda dei dispositivi ivi connessi che dinamicamente accedono alla rete per poi disconnettersi e accedere nuovamente in tempi successivi in un'altra posizione. Pertanto si parla principalmente di Mobile Ad hoc NETwork, identificate con l'acronimo MANET

Queste nuove e importanti caratteristiche hanno sollevato nuovi dibattiti e nuove problematiche in merito, mai affrontate precedentemente per le reti fisse.

Uno dei problemi principali è la tecnologia wireless stessa: la larghezza di banda fornita è di ordini di grandezza inferiore rispetto a quella delle reti cablate causando frequenti perdite di segnale e un alto livello di rumore introdotto nella trasmissione dei segnali, la cui influenza delle condizioni dell'ambiente esterno è notevolmente amplificata nello scenario mobile.

Un secondo aspetto è legato all'infrastruttura dei dispositivi mobili: essi sono caratterizzati da scarse risorse in termini di CPU, RAM, display e storage. In particolare, sono dotati di batterie che, seppur intelligenti, limitano l'autonomia del dispositivo (in termini di potenza e consumo) influenzando la comunicazione wireless e di conseguenza l'accesso ai servizi che richiede un elevato carico computazionale.

Infine, un terzo aspetto da non sottovalutare è la mobilità degli utenti, che causa problemi di perdita del segnale durante il movimento in una nuova cella[1]. A causa della limitata potenza delle trasmissioni radio e dell'utilizzazione dei canali stessi, un host mobile potrebbe non essere in grado di comunicare direttamente con gli altri hosts in comunicazioni single-hop. Di conseguenza lo scenario che si presenta per le reti mobili è uno scenario multi-hop, nel quale i pacchetti inviati dal nodo sorgente devono essere consegnati a più nodi intermediari prima di raggiungere la destinazione effettiva.

Questa problematica si riscontra anche nella gestione degli indirizzi di rete, che cambiano continuamente in quanto nella navigazione vengono percorsi più e diversi domini amministrativi di rete, creando la necessità di adeguare i servizi alla posizione dell'utente.

Soluzioni middleware tradizionali non sono in grado di gestire adeguatamente questi problemi. Questi sono stati progettati per la gestione host di rete fissa in un contesto statico, con l'obiettivo di nascondere dettagli di basso livello della rete per fornire un elevato livello di trasparenza alle applicazioni. L'ambiente statico è difatti caratterizzato da:

- Banda larga, bassa latenza e comunicazioni affidabili;

[1] Unità di area in cui è suddivisa una subnet

- Capacità di storage persistente, buone capacità computazionali, nessun problema di consumo energetico;
- No mobilità.

Il contesto quindi muta notevolmente quando si considera un ambiente mobile. Si parla di sistemi totalmente dinamici che non possono essere gestiti a priori, ma che devono essere supportati da efficienti tecniche di riconfigurazione in grado di reagire ai cambiamenti del contesto operativo.

Caratteristiche	Ambiente Distribuito	Ambiente Mobile
Banda	Alta	Bassa
Contesto	Statico	Dinamico
Tipo di connessione	Stabile	Instabile
Mobilità	No	Si
Comunicazione	Sincrona	Asincrona
Disponibilità di risorse	Alta	Bassa

Tabella 1.1 - Ambiente distribuito e mobile a confronto

Obiettivo principale del mobile middleware è di ricercare nuove tecnologie di rete in grado di gestire i continui cambiamenti dell'ambiente e di fornire meccanismi di cooperazione per sopperire alla mancanza di risorse dei dispositivi portatili. Altro obiettivo perseguito è la trasparenza. Un tecnologia è trasparente se nasconde la complessità delle operazioni sottostanti all'utente. Questo aspetto è molto importante nelle reti mobili per nascondere proprio la complessità indotta dal sistema distribuito e dai dispositivi eterogenei che vi operano.

Con questa tesi, si mostra come l'utilizzo della tecnologia ad agenti mobili fornisca una efficace strategia nello sviluppo di un'architettura adeguata che raggiunge gli obiettivi del mobile middleware.

1.2 Requisiti delle reti MANET

Le reti mobili ad hoc sono networks che vengono create dinamicamente (*on-the-fly*) per soddisfare un bisogno tipicamente temporaneo dei dispositivi che vi accedono. Usualmente tali reti coinvolgono nodi mobili, ad eccezione delle reti wireless di sensori (WSN). Sono strutturate similmente alle reti di natura cablata, ovvero partizionate in celle adiacenti per ottimizzare le funzioni di gestione e supporto alla rete stessa. Una cella è la più piccola infrastruttura che compone la rete nel suo complesso, è definita per questo *Basic Service Area* (BSA).

I caratteri che contraddistinguono questa tipologia di sistema sono i seguenti:

- **Mobilità**. Con i termine mobilità si rappresenta sia la mobilità degli utenti sia la mobilità dei dispositivi. L'obiettivo di una MANET ottimizzata è quello di mantenere la capacità di comunicazione in presenza della mobilità, anche se questa induce guasti e fallimenti o la variazione delle risorse nei nodi nel tempo.
- **Dispositivi che accedono**. I dispositivi mobili sono dotati di prestazioni e funzionalità differenziate, caratterizzando la rete da eterogeneità. Le operazioni hanno vincoli di bandwidth, di energia, di sicurezza anche se un sistema distribuito fornisce robustezza contro i fallimenti dei singoli nodi.
- **Alta volatilità.** È una caratteristica trasversale che coinvolge sia la rete sia la memoria dei dispositivi. Indica il contenuto altamente dinamico, in grado di cancellarsi spontaneamente, di variare e di aggiornarsi in tempi brevissimi.
- **Deployment**. Il deployment in queste reti deve essere immediato e altamente riconfigurabile a causa della dinamicità e dei cambiamenti repentini.
- **Capacità asimmetriche.** L'asimmetria è data dalla diversità dei raggi di copertura, delle tecniche di trasmissione wireless e dalle diverse caratteristiche dei dispositivi.
  Di seguito vengono riportate le grandezze relative alle diverse bande di cui i dispositivi possono usufruire a seconda della propria infrastruttura:
  - $10^0 \div 10^1$ Kbps (dati su GSM)
  - $10^1 \div 10^2$ Kbps (GPRS, UMTS)
  - $10^0 \div 10^2$ Kbps (satellite)
  - $10^0 \div 10^1$ Mbps (WLAN)

  I canali wireless su cui poggiano queste reti sono basati su radiofrequenze o infrarossi: le prime sono caratterizzate da una maggior portata e dalla capacità di attraversare ostacoli fisici come i muri; le seconde invece hanno una portata molto inferiore, raggio molto limitato e non attraversano oggetti fisici.

  Le reti wireless non necessariamente supportano la mobilità, si considerino ad esempio le reti fisse terrestri e satellitari, ma si può dire che permettono la mobilità garantendo maggiori funzioni.

  Queste capacità definite asimmetriche possono avere delle conseguenze di un certo interesse come perdite di pacchetti dovute a errori di trasmissione, partizionamenti di rete potenzialmente frequenti o problematiche legate alla sicurezza come il fenomeno dello *snooping*[2] delle trasmissioni wireless.
- **Responsabilità asimmetriche.** Si parla di responsabilità asimmetriche come di capacità in possesso a un dispositivo di assumere ruoli diversi, con particolare attenzione al ruolo di router per l'inoltro dei dati. In una rete mobile ogni nodo può svolgere il ruolo potenziale di router generando

[2] Letteralmente *curiosare*, lo snooping è un'operazione di scrittura non autorizzata

comunicazioni multi-hop, ma in realtà solo alcuni nodi possono eseguire il routing dei pacchetti. Esiste comunque la possibilità all'interno di un cluster di nodi, di avere un device che assume ruolo leader per i nodi vicini, definito quindi *cluster head*.

- **Trasparenza.** Una tecnologia è trasparente se nasconde agli utenti la complessità delle operazioni svolte in background per soddisfare le loro richieste.

La dinamicità di questo ambiente pone di fronte alla necessità di sfruttare nuovi criteri per la valutazione delle performance e per la personalizzazione dei servizi usufruiti. Tra i fattori che influenzano le prestazioni di una mobile ad-hoc network sono da considerare:

- Le dimensioni della rete;
- La connettività della rete intesa come il numero medio di collegamenti per nodo;
- Il traffico;
- Il tasso di cambio della topologia e la velocità della variazione;
- La capacità dei link ovvero il tasso effettivo (bps) depurato degli effetti di perdita pacchetti, codifica, overhead per accessi multipli,ecc. .

Ad esempio, come indici di prestazione per gli algoritmi sviluppati ad-hoc si valutano la quantità traffico dati generato, il consumo di energia del sistema computazionale considerato durante l'esecuzione e l'efficienza del meccanismo di localizzazione adottato che può causare ritardi nelle comunicazioni.

Vengono identificati per il mobile computing principalmente 5 requisiti non funzionali, ovvero proprietà che l'infrastruttura implementativa deve avere e vincoli che le funzioni associate devono rispettare.

Si parla di:

- **Raggiungibilità e accessibilità delle risorse fornite dai dispositivi**. Queste caratteristiche non devono essere legate alla locazione corrente degli utenti e delle risorse stesse ma devono essere gestite esplicitamente dai progettisti, in quanto ci si trova ad affrontare ambienti estremamente dinamici ed eterogenei che richiedono soluzioni ad hoc.
- **Capacità di adattamento**. Rappresenta l'abilità dei dispositivi mobili tramite i loro componenti di monitorare i continui cambiamenti nell'ambiente, e di reagire a questi riconfigurando la rete in modo autonomo o assistito dall'utente. Il processo di sviluppo e successivamente l'accesso ai servizi mobili devono pertanto essere affiancati da tecniche di predizione dei cambiamenti. Si parla di *context-aware*, ovvero si pone particolare attenzione al contesto, inteso come ambiente circostante. Caratteri come il tempo, la locazione, gli interessi personali, le previsioni per le navigazioni future, …, devono essere utilizzati come bagaglio di conoscenza per adattare i servizi offerti e regolare l'accesso alle risorse ottenendo una visibilità personalizzata anche a livello di interfaccia.

- **Affidabilità**. Deve essere gestito un appropriato livello di trust fra le entità interagenti in modo tale che operazioni siano svolte in accordo ad aspettative. La fiducia è un concetto difficile per sistemi distribuiti in quanto non è più un meccanismo centralizzato ma diventa anch'esso distribuito. Al concetto di fiducia si associano i concetti di contratto e autorizzazione: come avviene la gestione decentralizzata di quest'ultimi?
- **Universalità**. Questo concetto è legato ai dati e alla possibilità di accedervi in maniera universale come per i dati sul web. Le risorse su Internet sono infatti accessibili da qualsiasi luogo e in qualsiasi momento, utilizzando protocolli e formati standard. Pertanto, l'universalità è anche un requisito fondamentale per il mobile computing. La tendenza attuale è quella di utilizzare protocolli già sfruttati per le reti statiche anche in ambiente mobile.
- **Scalabilità**. La scalabilità è implicita nel concetto di universalità. Questo concetto può essere percepito in molti modi, per esempio il numero di nodi supportati dalla rete e il massimo numero di messaggi che possono passare attraverso un sistema. Attualmente ci sono oltre tre miliardi di telefoni cellulari sul mercato e l'aspettativa è che il valore raggiungerà la quota di cinque miliardi negli prossimi anni.

Gli obiettivi del mobile middleware si possono pertanto riassumere in: condivisione di risorse, adattatività, supporto eterogeneità, scalabilità e fault-tolerance.

Per raggiungere queste caratteristiche nelle reti MANET si richiede agli sviluppatori un approccio a livelli multipli, per riuscire a soddisfare i requisiti richiesti con competenze specifiche e adeguate. I livelli possono essere contraddistinti in:

- Livello dei dispositivi embedded a cui si affiancano problematiche di miniaturizzazione e di gestione delle risorse;
- Livello delle comunicazioni wireless (IEEE 802.11a/b/g/s, Bluetooth, WiMAX, etc.)
- Livello delle piattaforme di supporto software (Android, iPhoneSDK, SymbianOS, …)

1.3 Dispositivi mobili

Attualmente siamo giunti alla quarta generazione di dispositivi portatili. Lo sviluppo smodato e continuo ci ha portato a soli vent'anni di distanza dalla prima generazione, risalente agli anni 90, caratterizzata da forme limitate di accesso mobile ai dati e banda massima accessibile attorno alle decine di Kbps. La generazione odierna offre tecnologie ad alte prestazioni supportate da servizi, interfacce e protocolli adattativi, connettività continua e banda massima fino a centinaia di Mbps.

La caratteristica principale che distingue i devices oggigiorno è l'eterogeneità. Di conseguenza, il problema rappresentato dalle entità mobili che accedono alle reti ad hoc è legato alla grande varianza di risorse e dalle loro differenziate capacità prestazionali. Si sottolinea che la variazione nella capacità di risorse offerte varia di unità di grandezza. Ad esempio la potenza di calcolo ha prestazioni che variano dai 10^0 Mhz dei sensori ai 10^2 Mhz per i telefoni SMART, per poi passare a grandezze nettamente superiori offerte da PDA, laptop e sistemi desktop. Per quanto riguarda la memoria RAM si parla similmente di grandezze dell'ordine di 10^0 Kb per i sensori, 10^1 Kb per i telefoni mobili, etc.

Ma non sono le uniche grandezze fisiche che possono portare all'incompatibilità, basta considerare:

- la disponibilità di energia limitata nei dispositivi che può durare in media dalle 10^0, 10^1, 10^2 … ore a seconda dell'uso,
- la modalità di interazione,
- la risoluzione video,
- il colore: si passa dai toni del bianco e nero a 256 colori e più,
- l'audio, che può essere anch'esso di bassa o alta qualità,
- i dispositivi input adottati: tastiere delle più svariate grandezze, penne digitali, mouse, ...,
- la variazione di indirizzo IP che è fornito dalla rete che offre la connettività in quel momento e luogo. Infatti quando l'utente si connette ad un'altra rete, l'indirizzo IP tende a cambiare e i pacchetti che appartengono a connessioni correntemente attive devono essere consegnati verso il nuovo indirizzo IP.

Più precisamente i dispositivi vengono differenziati dalla sigla G, 2G, 3G, 4G a simboleggiare i cambiamenti di generazione ovvero le nuove funzionalità acquisite con il passare degli anni e l'aumento delle risorse:

- Sistemi 2G: sono sistemi rappresentati da una comunicazione limitata al vocale:
  - IS-136 TDMA: FDMA/TDMA combinati
  - Global System for Mobile communications (GSM): FDMA/TDMA combinati
  - IS-95 Code Division Multiple Access (CDMA)
- Sistemi 2.5G: canali per voce e dati
  - Estensioni a 2G prima della diffusione di 3G
  - General Packet Radio Service (GPRS)
  - Evoluzione di GSM
  - Data inviati su canali multipli, quando disponibili
  - Enhanced Data rates for Global Evolution (EDGE)
  - Data rate fino a 384Kbps
- Sistemi 3G: canali voce e dati
  - Universal Mobile Telecommunications Service (UMTS)
  - Ulteriore passo evolutivo di GSM, ma con utilizzo di CDMA
  - CDMA-2000

    - WCDMA
- Sistemi 4G: canali voce e dati
    - Data rate fino a 20Mbps
    - Indicati spesso con termine LTE o LTE-Advanced (Long Term Evolution)
    - Già disponibile in Giappone dal 2006

Di seguito riportiamo alcuni dati più precisi rispetto alle tecnologie oggigiorno più in espansione.

Sensore:

- cpu: 16 bit RISC
- memoria: 10 KB RAM + 48 KB (ROM) flash + 1 MB external (serial)R/W flash
- tecnologia di comunicazione wireless: IEEE 802.15.4/ZigBee compliant (250 kbps)
- interfaccia utente: porta USB
- fornitura di energia: 2 batterie AA

Smartphone:

- cpu: 32 bit
- memoria: 16 MB SDRAM + 64 MB / 1 GB flash
- tecnologia di comunicazione wireless: IEEE 802.11b/g (11-54 Mbps), GSM/GPRS/EDGE, IEEE 802.15
- interfaccia utente: 3.25¡¨ LCD display, 480x360 pixels, 65000 colors; touch screen
- fornitura di energia: batteria ricaricabile al litio

Laptop:

- cpu: dual-core 32/64 bit
- memoria: 4 GB RAM + 250 / 500 GB HD
- tecnologia di comunicazione wireless: IEEE 802.11b/g (11-54 Mbps), GSM/GPRS/EDGE, IEEE 802.15
- interfaccia utente: 15” LCD display, 1440x900 pixels, 106 colors; regular keyboard; trackpad
- fornitura di energia: batteria ricaricabile al litio

2 Mobile Agents

In questo capitolo viene presentata l'evoluzione dalla programmazione ad oggetti al nuovo concetto di paradigma ad agenti. Nello specifico vengono elencate le caratteristiche degli agenti mobili e vengono chiariti i concetti di mobilità e di migrazione applicati agli agenti. Nell'ultimo paragrafo del capitolo si presentano le piattaforme diffuse attualmente che supportano questa nuova tecnologia.

2.1 Evoluzione del paradigma ad oggetti

La programmazione orientata agli oggetti (OOP, *Object Oriented Programming*) ha rappresentato nel decennio scorso una notevole svolta nell'approccio assunto dal programmatore nella creazione di software. L'idea di base è l'interazione biunivoca tra oggetti, che si scambiano messaggi mantenendo ognuno il proprio stato ed i propri dati.

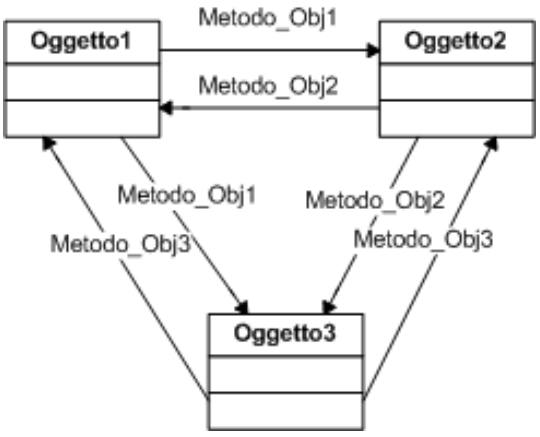


Figura 2.1 - Colloquio tra oggetti

In OOP, le caratteristiche di un oggetto vengono denominate *proprietà* e le azioni sono dette *metodi***.** I vantaggi che questa tipologia di programmazione offre sono:

- la possibilità di creare programmi modulari, ovvero suddivisi in sezioni tali per cui una classe può essere interamente riprogrammata, rispettando alcuni vincoli, senza andare a modificare il resto del programma;
- la possibilità di riutilizzare una classe in più programmi senza alcuna modifica, sfruttando metodi già strutturati.

Con il termine *Classe* viene quindi rappresentato un oggetto statico, composto da codice passivo, specializzato e implementato per risolvere problemi certi, ovvero problemi specifici che se modificati nel tempo richiederanno una nuova ristrutturazione della classe.

La programmazione ad agenti amplia e rivoluziona il concetto di modularità. La crescente diffusione di macchine sempre più performanti, di sistemi operativi multitasking, e soprattutto di ambienti distribuiti, ha permesso la crescita di un approccio distribuito alla formazione delle applicazioni. Sotto quest'ottica i moduli statici, che nella programmazione ad oggetti risiedono nella medesima macchina, ora incorrono in un processo di dislocazione: le componenti specializzate risiedono su macchine diverse e possono essere richiamate dall'applicazione vera e propria tramite lo scambio degli atti comunicativi.

Gli agenti sono nati per supportare questo approccio distributivo, sono componenti che non solo comunicano fra loro, ma possono anche essere in grado di spostarsi da un computer a un altro con i dati richiesti. Un agente, pertanto, non è solo un oggetto intelligente, composto da codice attivo, ma un più complesso sistema computazionale che assume veri e propri comportamenti. Esso è in grado di:

- **Interagire con l'ambiente circostante.** Si parla di reattività dell'agente, ovvero la capacità di percepire le condizioni in cui versa l'ambiente e di agire in conseguenza agli stimoli ottenuti. La percezione dell'ambiente viene interpretata dall'agente con l'assunzione di un proprio stato interno, che solo lui è in grado di modificare;
- **Offrire autonomia nelle azioni.** L'agente prende decisioni su quale azioni intraprendere in base allo stato interno assunto in quel momento. In una rete dinamica la mobilità rafforza l'autonomia, l'agente infatti decide in base alle politiche adottate e alle risorse richieste verso quale macchina migrare. L'autonomia espressa è direttamente proporzionale al ciclo di vita dell'agente, cresce infatti con la quantità di informazioni che vengono acquisite dallo studio dell'environment. Si parla quindi di proattività dell'agente;
- **Comunicare**, e quindi esprimere un'interazione sociale con altri agenti e/o con esseri umani. Per comunicare s'intende la capacità di coordinarsi, cooperare e negoziare con altri soggetti;

La programmazione ad agenti è basata strettamente su quella ad oggetti: un agente può essere visto come una classe che incapsula le funzionalità implementate

dall'agente stesso, i cui comportamenti riportano metodi e proprietà pubbliche che definiscono i servizi messi a disposizione.

Si parla di scambio di messaggi sia nel contesto degli oggetti che in quello degli agenti, ma i due concetti sono diversi:

- Secondo la filosofia ad oggetti, lo scambio di messaggi significa invocare dei metodi, ossia eseguire delle chiamate di procedure, generalmente a temporizzazione sincrona.
- Secondo la filosofia ad agenti, lo scambio di messaggi significa inviare e ricevere atti comunicativi, ovvero veri e propri messaggi che trasportano dati già elaborati, generalmente a temporizzazione asincrona. Gli agenti devono scambiarsi messaggi facendo riferimento ad un nome, o comunque ad un identificativo (ad esempio l'*Agent Identificator*, AID, definito da FIPA), ma mai conservare riferimenti espliciti ad altri agenti né invocare direttamente i metodi.

Per far fronte a questa nuova logica, i linguaggi di programmazione tradizionali, che consentivano una certa flessibilità, si sono evoluti, estendendo nuove funzionalità, Come ad esempio il linguaggio C++ che rappresenta un'estensione del C. Alcuni linguaggi, come Java, sono nati ex-novo, impostati totalmente sulla filosofia ad oggetti.

Per un sistema multi agente, creato per far fronte alle peculiarità di una MANET si preferisce sfruttare il linguaggio Java, contraddistinto da caratteristiche che meglio si adattano all'ambiente dinamico di cui si sta discutendo.

Si tratta di un linguaggio indipendente dalla piattaforma, la cui architettura è studiata in modo da rendere completamente portabili sia il codice sorgente che il codice compilato, ovvero il bytecode. La sintassi di Java deriva dal C++, uno dei linguaggi di programmazione più popolari, ma a differenza del suo predecessore può essere definito un linguaggio puramente object-oriented, in quanto non utilizza la logica della programmazione procedurale. Altre funzionalità interessanti sono la gestione semplificata del multi-threading e della comunicazione attraverso la rete, il trattamento delle eccezioni, e il meccanismo di *serialization* e quello di *reflection*.

Per quanto riguarda la gestione delle applicazione in rete, Java adotta diversi meccanismi, ma il più utilizzato è l' RMI (*Remote Method Invocation*), una tecnica di comunicazione tra oggetti che estende la classica chiamata a procedura remota (RPC). Questa procedura viene sfruttata per l'invocazione di metodi e di oggetti che si trovano fisicamente su un computer remoto come se fossero oggetti locali. Nel caso dei sistemi multi agenti viene sfruttata in molti tool di sviluppo che utilizzano solo agenti statici e quindi sostituisce la migrazione degli agenti nell'host di destinazione per le computazioni locali.

La serializzazione è un meccanismo che permette di rendere persistenti gli oggetti, salvandone lo stato. Mentre la riflessione è una funzionalità che pochi ambienti di programmazione possiedono e fornisce la capacità di accedere dinamicamente a classi, oggetti e metodi, a runtime. In Java questa è una caratteristica intrinseca del

linguaggio e questi ultimi due meccanismi verranno specificati ulteriormente al paragrafo 2.2.2.1.

Di seguito vengono riassunti i requisiti del codice propri della programmazione mobile e del linguaggio Java:

- Portabilità
- Sicurezza
- Integrità
- Disponibilità
- Efficienza

2.2 La mobilità

Gli agenti vengono differenziati non solo dal servizio che offrono ma anche dalla loro capacità di esprimere la mobilità, ovvero la caratteristica di migrare da un dispositivo all'altro.

Si definiscono agenti stazionari, agenti che comunicano con l'ambiente attraverso procedure tradizionali, come ad esempio il meccanismo di chiamata di procedura remota, definito nel paragrafo precedente, e rimangono attivi solo nel sistema in cui inizia la loro esecuzione. Al contrario, si definiscono agenti mobili, agenti non legati al dispositivo di attivazione e che possono muoversi tra gli hosts della rete alla ricerca dei servizi richiesti e/o di risorse remote. La migrazione può avvenire anche solo per comunicare con altri agenti, senza la necessità di installare supporti specifici. Il supporto fisico per questa nuova tecnologia si riduce infatti ad essere una generica piattaforma ad agenti che supporta la mobilità.

Si parla quindi di codice mobile come una tecnica in cui il codice è trasferito dal sistema che contiene le cartelle e i file annessi ad un sistema dove il codice viene eseguito effettivamente, e di mobile agent come una speciale tipologia di codice mobile.

Da un lato, questa caratteristica permette operazioni decentrate, dall'altro, vi è un'interazione più radicata tra l'agente e l'ambiente in cui opera. Grazie a questa capacità si facilita in modo notevole l'aumento della scalabilità delle networks, ed l'utilizzo di meccanismi di context e location-aware.

Esistono due tipologie principali di mobilità, la mobilità proattiva e quella reattiva:

- La mobilità proattiva viene attivata, in modo controllato e rigoroso, periodicamente, e/o per cambi di configurazione della rete, e/o per il movimento del dispositivo. Viene definita *proattiva* in quanto costruisce in anticipo, rispetto alla fase di mobilità vera e propria, le informazioni per la localizzazione dei dispositivi in cui migrare. Il vantaggio principale è relativo alla riduzione del ritardo di localizzazione, ma questo favorisce la scalabilità dei database che supportano le informazioni della localizzazione, portando a

possibili ritardi negli aggiornamenti, a overhead per il controllo del traffico e a consumo di capacità della rete e di energia.

- La mobilità reattiva si basa, al contrario, su algoritmi guidati dalla presenza effettiva di traffico. La localizzazione viene attivata solo se il livello di traffico per quella destinazione supera un certo valore soglia e il database di informazioni viene popolato al momento. I vantaggi che si riscontrano sono la potenziale riduzione del traffico di controllo e la riduzione dell' utilizzazione dei link e del consumo di energia. Al contrario viene fornito un maggiore ritardo di localizzazione.

2.2.1 I vantaggi

Gli agenti mobili sembrano essere una scelta ottima per trattare le questioni relative alla fornitura di servizi avanzati in ambienti mobili. La programmazione ad agente si sta sviluppando come un modo flessibile e complementare di gestire le risorse nei sistemi distribuiti grazie alla maggiore flessibilità nell'adattarsi alle esigenze che cambiano dinamicamente. L'interesse dimostrato nello studio degli agenti non è derivato dal loro innovativo concetto di programmazione quanto dai benefici che forniscono nella creazione e nella gestione di un sistema distribuito.

Il primo vantaggio deriva dai concetti di autonomia e di esecuzione asincrona propri della definizione stessa di agente. I dispositivi mobili si affidano spesso a costose e fragili connessioni di rete e per questo le funzioni vengono incorporate nel codice degli agenti mobili. Queste entità migrano nell'host di destinazione per effettuare le computazioni localmente. Dopo essere stati spediti, gli agenti diventano indipendenti dal processo che li ha creati e possono operare in modo asincrono e autonomo. Il dispositivo mobile potrà riconnettersi in un secondo momento per 'raccogliere' i risultati elaborati dall'agente. Questa soluzione riduce notevolmente il consumo di energia e di cpu all'interno del dispositivo sorgente in quanto l'elaborazione coinvolge solo due atti comunicativi verso l'esterno: uno per inviare l'agente, l'altro per riceverlo. Nei mobile devices lo spreco di energia è legato infatti alla quantità di messaggi che il dispositivo scambia con i nodi circostanti. In aggiunta questo spostamento delle operazioni presso il nodo che contiene i dati o le funzionalità richieste permette operazioni di tipo off-line.

Il loro utilizzo riduce inoltre il carico della rete. In un sistema distribuito, composto da dispositivi eterogenei, è molto importante l'interazione e quindi la comunicazione per portare a termine un dato task. Questo porta ad avere un notevole traffico di rete dovuto ai messaggi che tali dispositivi si scambiano. Gli agenti mobili permettono all'utente di 'impachettare' la conversazione e di effettuarne il *dispatch* presso un'altra destinazione, dove le interazioni possono avvenire localmente. Questo riduce inoltre il flusso di raw data nelle networks, che ha un notevole impatto positivo soprattutto quando i dati richiesti sono grandi volumi di dati, contenuti in host remoti.

Gli agenti mobili incapsulano nei loro behaviour i protocolli di comunicazione. Questo facilita l'esecuzione di codice esterno, proveniente da una diversa piattaforma. In questo modo il codice viene interpretato correttamente senza il bisogno di verifiche per i possibili upgrade del protocollo stesso. Gli agenti mobili vengono per questo descritti come computer and transport-layer indipendent, e per questo forniscono le condizioni ottimali per l'integrazione tra dispositivi composti da piattaforme non uniformi.

Un altro aspetto importante è l'azzeramento della latenza di rete, i cui valori superiori sono da considerare inaccettabili in questi sistemi real-time, dove le risposte agli stimoli ricevuti dall'ambiente devono essere immediate. Grazie al dispatch degli agenti nell'host interessato, il dispositivo può far eseguire localmente le sue direttive e ottenere una risposta in tempi quasi impercettibili. L'agente infatti si adatta dinamicamente ai cambiamenti e può portare a termine operazioni complesse operando in modo tradizionale, in tempi brevi e con il minor dispendio di risorse possibile. Da sottolineare che esistono agenti mobili specializzati a mantenere la configurazione ottimale della rete e contengono procedure di risoluzione dei fallimenti per il ripristino della stessa. In generale i mobile agents sono robusti e fault-tolerant, integrano infatti nei loro comportamenti dei metodi per reagire dinamicamente a situazioni sfavorevoli, come lo spegnimento o la disconnessione del dispositivo in cui stanno lavorando, o la perdita del segnale che causa dei crash alle connessioni di rete. L'utilizzo, quindi, di agenti mobili permette di superare problemi tipici e le restrizioni dovute alle comunicazione wireless. Si evidenziano, in aggiunta a quelle già menzionate, l'alto tasso di errore di bit, la bassa potenza di elaborazione e la piccola area disponibile per l'interfaccia utente.

2.2.2 Motivazioni alla migrazione

Esistono diverse cause che portano un agente a migrare verso host esterni.

Si distinguono motivazioni funzionali, motivazioni legate alle prestazioni del sistema e motivazioni legate al raggiungimento della fault-tolerance.

Motivazioni funzionali.

Con il termine funzionale si evidenzia come causa dello spostamento la ricerca di nuove funzionalità necessarie alla computazione dell'agente o del dispositivo che delega l'agente. Si hanno ricerche per l'acquisizione di:

- Autorizzazioni/certificati per accedere a nuovi servizi con accesso limitato;
- Nuovi comportamenti da aggiungere al corpo eseguibile dell'agente, necessari alla computazione richiesta;
- Risultati elaborati da altri agenti che forniscono servizi complementari necessari all'elaborazione.

Motivazioni prestazionali.

Le cause della mobilità appartenenti a questa categoria sono da imputare alle prestazioni offerte dal sistema di calcolo in cui l'agente viene eseguito, e in particolare:

- Alla scarsità di risorse in possesso del dispositivo, le quali caratterizzano un sistema di bassa tecnologia;
- Alla diminuzione delle capacità dovute a:
  - rallentamenti per sovraccarichi del sistema;
  - congestioni del traffico dovute a molteplici operazioni di I/O contemporanee;
  - applicazioni attive in sottofondo che fanno scaricare più velocemente la batteria;
  - installazione di applicazioni *pesanti* che pregiudicano le performance relativamente alla stabilità, alla velocità e, soprattutto, alla durata di batteria.

Motivazioni dovute ai fallimenti.

Le reti MANET adottano algoritmi mirati per la predizione dei fallimenti. Quando la perdita del segnale di rete, il livello di batteria è quasi al limite o altri parametri di sistema superano un valore soglia di pericolo, viene trasmesso un alert in broadcast, ad ogni agente appartenente al dispositivo. A seconda della tipologia di agente, verrà innescato nel metodo `handleFailure()` il meccanismo di migrazione verso un altro container o la disattivazione dell'agente stesso. In questi casi, la mobilità rappresenta una risposta a causa di cambiamenti di stato del dispositivo che si sta preparando a disconnettersi dalla rete.

2.2.3 La migrazione in JAVA

Per migrazione dell'agente s'intende la capacità di trasportare il proprio stato e il proprio codice in un ambiente della rete diverso da quello di partenza, dove l'agente riprende l'esecuzione. Ci si aspetta che nel codice che implementa l'agente ci sia un metodo che faccia partire la migrazione (ad esempio il metodo `doMove(Location place)` in JADE) e che la piattaforma ad agenti utilizzata gestisca tutto il meccanismo, senza intervento esplicito del programmatore.

Con il termine *stato dell'agente* si identificano i valori degli attributi dell'agente stesso che lo aiutano a stabilire cosa fare quando riprende l'esecuzione al nodo di destinazione.

Un oggetto in Java è costituito principalmente da tre stati:

- Program state: byte code della classe dell'oggetto;
- Data state: contenuto delle variabili d'istanza dell'oggetto;
- Execution state: stato di esecuzione attuale dell'oggetto.

Esistono diversi tipi di migrazione, a seconda di quali stati vengono trasmessi. Si parla di:

- migrazione debole con invocazione di metodo fisso;
- migrazione debole con invocazione di metodo arbitrario;
- migrazione forte.

La prima trasmette solamente il data state e il program state, e di conseguenza la ripresa dell'esecuzione inizia con l'invocazione di un metodo ben definito in quanto non è accessibile lo stato di esecuzione dell'agente prima della migrazione. Questa tipologia viene utilizzata ad esempio dalle piattaforme Aglets, Grasshoppers e JADE e viene definita anche come migrazione non trasparente.

La seconda offre la possibilità, rispetto alla prima tipologia, di gestire in modo dinamico la scelta del metodo da invocare quando l'agente arriva a destinazione. Il nome del metodo viene infatti dato in input dal programmatore e trasmesso nella migrazione con il data e il program state. Il programmatore in questo caso deve impiegare uno sforzo aggiuntivo per implementare il marshalling e il demarshalling delle variabili locali, il cui valore viene registrato in variabili dell'oggetto ad hoc, utilizzate solo per la migrazione. Questa tecnica è utilizzata, ad esempio, dalla piattaforma Voyager.

In entrambe le migrazioni deboli il comando di migrazione deve essere l'ultima istruzione del metodo in quanto nella nuova agency[1] verrà invocato un nuovo metodo di partenza.

La migrazione forte a differenza delle altre, inserisce nel byte code da migrare anche tutte le variabili del metodo corrente, il program counter e il call stack, ovvero lo stato di esecuzione dell'oggetto. Più precisamente dallo schema riportato in basso si evidenzia cosa viene trasmesso rispetto ai tre stati:

[1] Una piattaforma ad agenti è definita *agency*, a sottolineare la capacità di queste strutture di registrare agenti e i loro servizi all'interno del dispositivo in cui sono installate. Un'agency verrà infatti interrogata da agenti esterni per le operazioni di ricerca, e per operazioni di registrazione da parte di agenti attivati dalla stessa piattaforma.

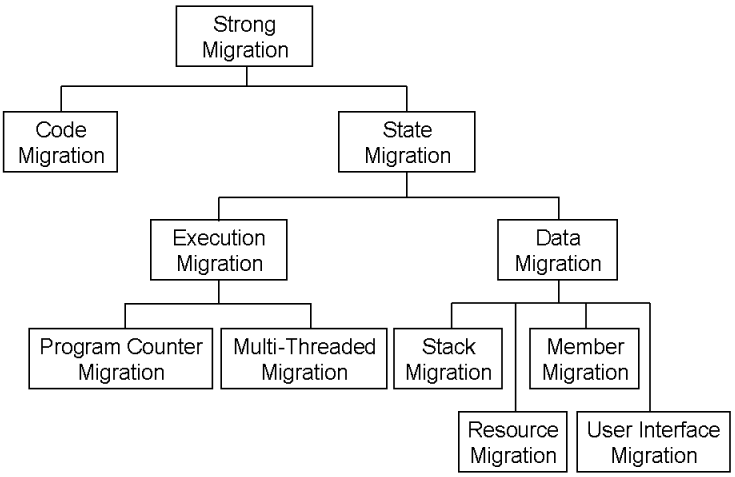


Figura 2.2 – Strong Migration

La migrazione dell'agente può essere implementata utilizzando i seguenti meccanismi standard di Java:

- Object serialization
- Reflection Technique
- Dynamic class loading

Nell'host ricevente viene infatti creata dapprima una copia dell'agente originario, che riporta nello specifico i dati serializzati. Successivamente l'agente nell'agency originaria viene eliminato o mantenuto a seconda se l'agente migra completamente o se solamente una percentuale del codice posseduto è stato dislocato.

Gli altri attributi (program counter, multi-threaded, stack, resource and user interface migration) non sono supportati da caratteristiche standard del linguaggio. Questo rappresenta l'effettivo svantaggio di JAVA rispetto alla mobilità, specificato dall'impossibilità di ottenere il corrente stato d'esecuzione di un thread.

Per questo, principalmente, i sistemi multi-agenti Java-based possono offrire solo una forma di migrazione debole: l'agente ricomincia la sua esecuzione all'agency ricevente invocando un metodo specifico, e non riprende dal metodo che stava eseguendo nel nodo precedente dopo la dislocazione.

La serializzazione.

La tecnica di serializzazione in Java consiste nel trasformare in uno stream di byte, e per la precisione in un byte array, i dati, lo stato di esecuzione e il codice di un agente. Costituisce l'insieme di dati tutte le variabili dell'agente e ricorsivamente tutti, o un insieme di, gli oggetti referenziati e le loro variabili. Si definisce per

questo il concetto di *object closure* ovvero il set di tutti gli oggetti che vengono serializzati. Costituisce, invece, il codice e più precisamente la *code closure*, la classe principale dell'agente e tutte le classi che l'agente potrebbe utilizzare a destinazione.

La serializzazione avviene invocando il metodo `writeObject(Object Obj)` della classe `ObjectOutputStream`, nel dispositivo di partenza; mentre la deserializzazione, ovvero il processo inverso che ripristina a destinazione l'agente e i suoi oggetti dall'array di byte, avviene invocando il metodo `readObject()` della classe `ObjectInputStream`.

Questa tecnica si adatta bene alla maggior parte delle situazioni in cui si deve trasmettere del codice da una macchina all'altra, ma bisogna considerare alcune limitazioni e caratteristiche aggiuntive per ottimizzare lo spostamento e limitare il carico della rete dovuto alla quantità di dati trasmessi:

- Il costo della migrazione;
- L'uso dell'attributo `transient` per limitare la taglia del codice serializzato;
- La ridefinizione dei processi di serializzazione e deserializzazione a seconda del contesto in cui il programmatore opera;
- La consistenza tra il ripristino degli agenti a destinazione e il nuovo ambiente;
- La gestione degli upgrades, nel caso in cui sussistano più versioni dell'agente e degli oggetti;
- Eventuali classi serializzabili che ereditano da classi non serializzabili.
- Il meccanismo dei tokens, utilizzati per identificare una classe serializzata;

Il costo della migrazione varia in base alla taglia dell'agente e degli oggetti referenziati da serializzare. Un quantità troppo elevata di dati intaserebbe la rete e le connessioni risulterebbero più predisposte a cadute di segnale con conseguenti rallentamenti.

Per garantire un'efficiente mobilità all'interno della rete, si tende a migrare solo una parte del codice necessario, eliminando le eventuali classi istanziate che si suppone potrebbero essere ridondanti nell'host ricevente. Il linguaggio Java non fornisce un tool per identificare tali classi, pertanto il programmatore dovrà gestire esplicitamente la scorporazione del codice limitando la migrazione dell'agente entro una certa taglia. Questo avviene attraverso l'uso dell'attributo `transient` nella dichiarazione di una classe o di un suo attributo. Il meccanismo di serializzazione quando esegue l'operazione di lettura e legge tale attributo, salta l'oggetto istanziato o la variabile a cui si riferisce e riprende la sua esecuzione alla dichiarazione successiva. Sono definite `transient`, variabili che modificano il loro stato in base alle condizioni in cui l'agente si trova. Il problema emerge nell'host di arrivo, dove avviene il meccanismo di deserializzazione che determina: come ricostruire o meglio linkare le classi di oggetti che volutamente si sono saltate durante il processo iniziale? Ecco il motivo per cui il programmatore deve

esplicitamente ridefinire i metodi di lettura e scrittura degli stream richiamando al loro interno gli attributi non serializzati, che muteranno in base al contesto. Da evidenziare che anche il riferimento alla piattaforma di appartenenza non viene serializzato ma viene ripristinato poi a destinazione, in cui viene aggiornato con il riferimento alla nuova piattaforma.

La gestione delle versioni degli upgrade del codice avviene attraverso la variabile `serialVersionUID`, un numero da 64 bit associato ad ogni classe che riporta il numero di versione e che viene utilizzato per gestire la compatibilità.

Nel caso in cui sussistano classi serializzabili che estendono classi non serializzabili vengono gestiti i metodi `writeExternal(Object out)` e `readExternal(Object in)`, della classe `Externalizable`, che definiscono le variabili d'istanza dell'oggetto non serializzabile. Tali variabili vengono poi aggiunte allo stream di dati in output.

Per evitare che uno stesso oggetto sia scritto più volte si usano dei tokens numerici. Un token è un identificatore di un oggetto serializzato per lo stream preso in considerazione. Il problema legato al loro utilizzo è associato al meccanismo di garbage collection. Infatti, anche se un oggetto serializzato perde tutti i suoi riferimenti, il riferimento dello stream rimane. La soluzione, pertanto si rifà all'invocazione del metodo `reset()` che elimina il contenuto della tabella dei riferimenti. Il suo utilizzo, però, può causare delle situazioni contrastanti: abusarne può portare alla duplicazione di oggetti ma non usarlo può portare alla perdita di informazioni importanti.

Reflection Tecnique e Dynamic Class Loading.

La tecnica di riflessione è utilizzata per riprendere l'esecuzione dell'agente nell'host di destinazione. Essa riporta informazioni a runtime delle classi relative alla porzione di codice che migra, delle loro variabili e dei loro metodi. Al termine dello spostamento, viene utilizzato un meccanismo dinamico di Java definito *Dynamic Class Loading* che permette il caricamento e il linking delle classi nella running agency ricevente. Questa tecnica, implementata dalla classe `ClassLoader`, permette alla JVM di trovare e definire le classi da caricare, in tempo reale, attraverso i metodi `findClass()` e `defineClass()`.

2.3 Piattaforme ad agenti mobili

Esistono numerosi strumenti software creati per semplificare la programmazione ad agenti: sono costituiti per lo più da librerie e tool che guidano l'utente durante le fasi di progetto, implementazione e collaudo di sistemi multi-agente. A seconda delle funzionalità fornite e delle capacità aggiunte da moduli esterni integrabili questo strumenti possono essere infrastrutture ad agenti, piattaforme e toolkit per la costruzione di sistemi multi-agente. In questo capitolo vengono affrontate le

piattaforme più diffuse attualmente per lo sviluppo di sistemi multi-agente e sono esposte le ragioni che hanno portato alla scelta di JADE per lo sviluppo degli agenti forniti nel sistema interattivo.

2.3.1 Aglets

Aglets è una delle prime piattaforme ad agenti sviluppate da IBM nel 1997, ma il lavoro attorno a questa piattaforma risulta fermo al 2002. Dal suo sviluppo partirono le prime idee innovative attorno allo sviluppo degli agenti: la prima era costruire questa struttura dati intelligente come un singolo thread, indipendente dagli altri in modo da ottenere una computazione parallela con più strutture attive; mentre la seconda riguardava la comunicazione basata sullo scambio di messaggi caratterizzata da trasmissioni sincrone e asincroniche. Con l'avanzamento delle attività e degli studi attorno a questa nuova tecnologie queste idee rappresentarono i principali svantaggi della piattaforma stessa: la struttura dell'agente basata su un modello single-threaded portava il programmatore a implementare l'agente con poche funzionalità, per evitare di perdere messaggi in arrivo che potevano anche non essere mai considerati durante l'esecuzione. Inoltre i proxies utilizzati per la comunicazione non sono dinamici ma lo sviluppatore deve esplicitamente gestire i loro aggiornamenti.

2.3.2 Voyager

Voyager è una piattaforma sviluppata anch'essa nel 1997 dall'azienda Object Space e portata avanti poi da Recursion Software. Voyager è interamente scritto in linguaggio di programmazione Java, integrato con il paradigma ad agenti. Un agente mobile viene considerato come un particolare tipo di oggetto, che ha semplicemente una proprietà aggiuntiva, ovvero la mobilità. Gli agenti mobili possono muoversi tra i server di Voyager e quando giungono a destinazione, il metodo passato come argomento al comando di spostamento viene richiamato e l'agente riprende la sua esecuzione. Viene quindi supportata una migrazione di tipo debole che delega al programmatore il compito di salvare esplicitamente lo stato interno dell'agente mediante l'utilizzo di variabili locali. Vengono utilizzati, inoltre, standard di serializzazione Java per il trasferimento di un agente e il suo stato, che vengono rigenerati al server di destinazione.

Questa piattaforma offre un servizio di archiviazione definito directory service, che viene eseguito localmente su ogni server, al quale si può accedere anche da remoto. Tuttavia, questa funzionalità non è fornita mediante l'uso di agenti.

Un aspetto che la contraddistingue è il fatto di non essere una piattaforma open source ma bensì commerciale, con qualche eccezione per periodi di prova concessi ad università o aziende per il suo utilizzo. L'ultima versione risale al 2005, ma costituisce comunque un buon prodotto per la realizzazione delle applicazioni multi agente.

2.3.3 SPRINGS

SPRINGS è una piattaforma sviluppata successivamente, e ancora in sviluppo, dal gruppo di laboratorio di Sistemi Distribuiti presso l'Università di Saragoza. Questo framework mira soprattutto alla gestione della scalabilità e all'affidabilità in ambiente mobile, supporta infatti le strutture ad agenti mobili che dislocano da un nodo all'altro per computare localmente le operazioni di cui necessitano. Viene fornita trasparenza a livello di locazione e i proxies utilizzati sono dinamici. Lo svantaggio principale nello sviluppo degli agenti è la comunicazione che non rispetta lo standard FIPA per lo scambio dei messaggi, inoltra non sviluppa ancora un buon meccanismo di sicurezza che può influenzare notevolmente nella scelta della piattaforma stesso nello sviluppo di questa nuova tecnologia mobile.

2.3.4 JADE

La piattaforma JADE è nata nel 1998, sviluppata dal Laboratorio di ricerca e sviluppo di Telecom Italia. Divenne poi open source nel 2000 ed è ancora attualmente in fase di aggiornamento, l'ultima versione risale infatti al 13 Febbraio dell'anno corrente. Questo nuovo framework rivede le idee introdotte dalla piattaforma Aglets e le cambia completamente. Il primo aspetto rivoluzionato è il modello di costruzione degli agenti che passa da single-threaded a multi-threaded: un agente è composto da più comportamenti che possono agire in concorrenza e essere aggiunti o eliminati dinamicamente. Il secondo aspetto importante riguarda la gestione delle comunicazioni che vede due importanti novità: da una parte il supporto dello standard FIPA per la formalizzazione dei messaggi, dall'altra l'utilizzo di un agente amministratore di sistema, detto AMS Agent, al posto dei proxies per ottenere i riferimenti agli agenti remoti.

A differenza delle altre piattaforme, questa introduce un potente supporto visivo al deployment degli agenti facilitando i processi di debug e monitoraggio degli agenti in esecuzione. La sua versatilità permette inoltre una forte integrazione con altri tool di sviluppo come ad esempio JESS per il supporto dell'ontologia, e la creazione di diverse versioni per permettere l'esecuzione del framework in un ampia gamma di dispositivi (vedi ad esempio JADE-Leap e JADE-ANDROID).

2.3.5 Caratteristiche a confronto

Nonostante la crescente popolarità dei servizi wireless e i vantaggi forniti dagli agenti mobili in questi ambienti, non ci sono molte applicazioni che utilizzano questa nuova tecnologia. Una possibile spiegazione potrebbe appoggiarsi alla difficoltà di sviluppare e mantenere queste applicazioni perché le esistenti piattaforme mancano di determinate caratteristiche che dovrebbero al contrario essere presenti. Si può parlare ad esempio di mancanza di funzioni legate alla sicurezza, a particolari topologie della rete, a tools di monitoraggio e debbunging.

Una piattaforma dovrebbe rappresentare una realizzazione concreta dell'architettura middleware.

Nonostante queste limitazioni le piattaforme attuali forniscono ugualmente una suite di funzionalità rilevanti per lo sviluppo di un sistema di questo tipo:

- **Localizzazione automatica dei nodi e dei servizi forniti.** Gli agenti sono forniti di determinati metodi per la localizzazione del nodo di arrivo dopo la migrazione. L'itinerario da intraprendere viene calcolato in maniera statica o tramite algoritmo on-demand: nella prima opzione l'agente conosce prima della migrazione tutti i nodi del path su cui spostarsi; nella seconda il calcolo avviene tramite un meccanismo di richiesta a seconda del contesto che viene ricalcolato ad ogni nodo visitato.
- **Localizzazione degli agenti mobili.** Il meccanismo di tracking si occupa di tracciare la posizione corrente di un agente mobile durante il processo di migrazione. Esistono diversi meccanismi utilizzati dalle piattaforme per fornire questo servizio. Ad esempio alcune piattaforme forniscono un sistema di *Name Services*, altre utilizzano un approccio basato su proxy, simili agli stubs nella RMI.
- **Java Virtual Machine.** Per garantire la portabilità del codice *mobile* nelle diverse piattaforme sono state sviluppate più versioni della macchina virtuale di Java, sviluppo incentivato dal costo per l'acquisto di queste piattaforme. Una delle JVM utilizzata maggiormente è la IBM Websphere J9.
- **Protocolli di comunicazione.** Esistono diversi protocolli di rete nati soprattutto dallo sviluppo del middleware che superano gli ostacoli dei protocolli tradizionali di alto e basso livello. Nelle MANET viene attualmente utilizzato il protocollo HTTP (*Hyper-Text Transfer Protocol*)
- **Deployment e tools grafici di supporto.** Il deployment (rilascio o distribuzione) è l'applicazione della soluzione sviluppata al problema reale in un dato dominio. Consiste nell'avviare il sistema ad agenti (tipicamente su una rete di computer) e quindi di collaudarlo, effettuarne la manutenzione e/o estenderne le funzionalità. L'unica piataforma che provvede a fornire un tool grafico per supportare questa funzionalità è la piattaforma JADE presentata nel prossimo paragrafo.

La scelta della piattaforma da utilizzare per lo sviluppo dei prototipi e dei sistemi demo contenuti nel sistema è ricaduta su JADE. JADE infatti non presenta strumenti di sviluppo, che saranno forniti dal framework realizzato, ma il supporto runtime è molto buono: dall'interfaccia grafica di Jade è possibile gestire e monitorare anche gli agenti che risiedono su altre piattaforme, visualizzarne le comunicazioni e gli spostamenti. Uno strumento estremamente utile per il collaudo di un sistema è l'agente Sniffer, che analizza graficamente le conversazioni che avvengono all'interno del sistema: l'utente sceglie gli agenti che desidera monitorare, e lo Sniffer traccia in tempo reale un diagramma di sequenza UML comprendente tutti i messaggi scambiati da quegli agenti.

Si riassumono in questa tabella le caratteristiche delle quattro piattaforme descritte – *rif. [9]*

Caratteristiche	Aglets	Voyager	SPRINGS	JADE
Componenti della piattaforma	Context, Agenti, Tahiti	Server, Agenti	Places, Regioni, Agenti	Containers, Piattaforme, Agenti, DF, AMS
Supporto di Proxy/Tipologia	Si/Stabili	Si/Dinamici	Si/Dinamici	No
Supporto di comunicazione sincrona/asincrona	Si/si	Si/si	Si/si	No (il supporto è fornito in modo esplicito dallo sviluppatore)/ si
Scambio di messaggi	Si	No	Si	Si (FIPA)
Supporto di RMI	No	Si	Si	No
Modello di struttura degli agenti	Ad eventi	Procedurale	Procedurale	Behaviours
Tools grafici forniti	Alcuni	No	No	Si
Licenze	IBM Public License	Not free (evaluation version)	Open source	LGPL
Sicurezza	Base	Si (security managers, etc.)	Base	Si (JAAS, etc.)
Altre caratteristiche	ATP, Itinerary	Multicast, Publish/subscribe, Dynamic aggregation	No livelock, Schedule, Reliable, efficient	FIPA, Jess, JADEX, Ontology support

Tabella 2.1 – Confronto tra piattaforme ad agenti

3 Categorizzazione degli Agenti

In questo paragrafo si è affrontata la categorizzazione semantica degli agenti per permettere una semplificazione nel processo progettuale. L'obiettivo principale è quello di delineare in maniera chiara ed esaustiva quali sono i ruoli e le principali funzioni associate, elencando anche alcuni esempi che poi saranno inseriti come prototipi nel sistema interattivo.

3.1 Terminologia Introduttiva

Di seguito si elencano alcune terminologie proprie di una rete mobile con tecnologia ad agenti:

- **Incoming Agent**: Agente in fase di migrazione in arrivo nel dispositivo considerato.
- **Outgoing Agent**: Agente che lascia il nodo in cui è situato in quel momento verso un'altra destinazione
- **Mobile Node (MN):** Dispositivo mobile che può cambiare il punto di connessione alla rete senza cambiare il suo indirizzo IP, definito anche come *home address.*
- **Home Agent (HA):** Agente nella home network del Mobile Node, che tipicamente ha le funzionalità di router. Mantiene aggiornata una struttura dati di indirizzi IP, registrando la locazione corrente dei nodi all'interno del raggio di copertura wireless della propria rete e reindirizza mediante un meccanismo definito *tunneling* gli agenti verso altre reti (COA[1]).
- **Foreign Agent (FA)**: Agente nella rete corrente di MN, tipicamente un router con funzionalità arricchite. Inoltra verso MN i pacchetti ricevuti, tipicamente è anche il default router di MN.
- **Correspondent Node (CN)**: Partner di una comunicazione con MN.

[1] Il *Care-of Address* (COA) è l' indirizzo IP che identifica l'attuale locazione di MN ovvero l'indirizzo temporaneo, *temporary address*.

3.2 Agenti Comportamentali

Gli agenti comportamentali sono agenti che esprimono la capacità di relazionarsi con altre entità, siano esse agenti, utenti, dispositivi e in grado di acquisire conoscenza studiando l'ambiente nel quale sono in esecuzione. In questo modo il concetto di comportamento in questi agenti è maggiormente radicato. A seconda dello stato interno assunto modificano dinamicamente e in maniera sempre più autonoma il comportamento da assumere in quel determinato frangente. Nelle sezioni successive vengono trattati i principali comportamenti che un agente può assumere.

3.2.1 Negotiator Agent

Un agente negoziatore è un agente che partecipa ad un accordo tra più agenti in conflitto. Si definisce conflitto una situazione in cui più dispositivi/agenti devono negoziare per raggiungere i propri obiettivi; obiettivi che possono essere in contrasto con quelli degli altri agenti, come ad esempio l'utilizzo completo di una risorsa, la massima efficienza di un servizio oppure ottenere un bene di qualità al minimo prezzo, etc. Un agente Negoziatore deve avere capacità di vendita se si tratta di un agente vendita e capacità di acquisto se si tratta di un agente compratore. Deve quindi essere in grado di gestire diverse tipologie d'asta che verranno scelte a seconda di una previa valutazione delle condizioni ambientali in cui l'agente si trova (per le tipologie di asta si rimanda al paragrafo successivo).

Agente Negoziatore e MANET.

L'atto della negoziazione avviene interamente nel dispositivo che fornisce il servizio/la risorsa/il bene, e pertanto dove risiede l'agente banditore che deve 'vendere' al migliore acquirente. Gli agenti offerenti (bidders) migrano dai propri nodi verso il nodo interessato e dopo aver raggiunto l'accordo tra le parti, ritornano al nodo sorgente per comunicare il risultato ottenuto. Questa gestione in toto del negoziato presso un singolo dispositivo permette al nodo stesso una gestione ottimizzata delle risorse e la riduzione degli sprechi di risorse di tutti i dispositivi in gioco.

Nel caso di conflitti che riguardano l'utilizzo di risorse offerte dal sistema si può ipotizzare l'esistenza di un resource management agent che gestisce le richieste degli agenti ed eventualmente avvia la negoziazione vera e propria.

In dispositivi con scarsa capacità di risorse nei quali si deve ospitare una negoziazione si presuppone che non venga utilizzato un agente intermediario al fine di evitare congestioni e sprechi inefficienti delle risorse interne. In caso contrario la negoziazione avrà luogo nel nodo più vicino con capacità prestazioni più efficienti.

Inoltre nel caso si verifichi un fallimento del nodo, l'asta viene annullata e agli agenti verrà comunicato di ritornare ai nodi sorgenti.

Ogni agente negoziatore conosce:

- il negotiation set, ovvero lo spazio di tutti i possibili proposals (proposte) che l'agente può richiedere;
- il protocollo di comunicazione che rappresenta quali sono i messaggi accettati dagli agenti durante il negoziato e come vengono formalizzate le proposte. Il protocollo definisce la sintassi legale per lo scambio di messaggi;
- un piano strategico dove sono elencate tutte le strategie da seguire in base allo stato interno dell'agente.

  Un piano strategico è composto da:

  - una funzione di utilità, detta anche benefit tale per cui, per ogni stato dell'ambiente, si quantifica un valore reale che definisce il livello di gradimento dell'agente considerato per quel particolare stato dell'ambiente. A partire dai risultati della funzione di utilità è possibile stabilire una lista basata sull' importanza degli stati dell'ambiente che l'agente deve raggiungere;
  - un insieme di strategie che vengono scelte in base al contesto. In alcuni casi, durante la negoziazione, la miglior strategia che un agente deve intraprendere può dipendere anche dalle strategie che seguono gli altri agenti. Esprime in questo caso una capacità di tipo conoscitivo derivata dall'esperienza avuta.

3.2.2 Broker Agent

Un agente mediatore è un agente che si pone nell'intermezzo di due o più agenti negoziatori che hanno lo scopo di raggiungere un accordo. Per agenti negoziatori s'intendono agenti compratori e venditori. Il mediatore non è un partecipante nel conflitto in atto ma gestisce le proposte e i rilanci scegliendo l'opportuna strategia a seconda delle volontà del venditore.

Agente mediatore e MANET.

In una rete mobile, i conflitti avvengono nel luogo in cui più agenti ricercano l'utilizzo della medesima risorsa o si contendono determinati servizi. La negoziazione pertanto si svolge interamente nel nodo conflittuale, il venditore ricerca attraverso il DF un agente mediatore, il quale, se presente, riceve dal venditore stesso l'indirizzo del dispositivo in cui è in esecuzione per la migrazione. Di norma un agente mediatore viene attivato dal 'venditore' del bene o dal gestore del servizio offerto.

Nei dispositivi con disponibilità di risorse maggiore si presuppone che sia già presente un agente mediatore specializzato; in questo modo quando viene

identificato un conflitto, l'agente entra in esecuzione velocemente, allocando dinamicamente una quantità di memoria già stabilita. Se richieste multiple della medesima risorsa si avvicendano in un dispositivo con scarsità di risorse, come già specificato per un agente negoziatore, la negoziazione può avvenire senza l'utilizzo di un agente mediatore, ma può essere risolta attraverso politiche di gestione già implementate nei dispositivi.

Dopo la risoluzione del conflitto gli agenti coinvolti continuano le loro operazioni nel perseguire l'obiettivo per cui sono stati creati mentre l'agente mediatore viene 'disattivato' e la memoria utilizzata viene deallocata.

Accenni sulle tipologie di asta.

Le aste costituiscono uno dei protocolli di negoziazione più semplici da gestire.

Un'asta consiste generalmente in un banditore (*auctioneer*), che vuole vendere un bene (di solito di sua proprietà), e un insieme di offerenti (*bidders*), che vogliono acquistare il bene messo all'asta dal banditore. Nella maggior parte dei casi il banditore è interessato a trarre il massimo profitto dalla vendita del bene, mentre gli offerenti sono interessati ad acquistarlo al minor prezzo possibile. Tutta la fase di contrattazione fra banditore e offerenti deve comunque seguire le regole imposte dal protocollo di negoziazione.

Nel caso di offerte pubbliche l'asta viene definita a pubblico annuncio (*open-cry*), altrimenti viene definita ad offerta segreta (*sealed-bid*).

Tipologie di asta:

- **Asta inglese.** First-price(higher price),open cry/sealed bid, ascending auction. L'asta inglese è probabilmente la tipologia d'asta più comune e consiste dei seguenti passaggi:
  1. il banditore mette all'asta un bene ad un prezzo base;
  2. i partecipanti all'asta possono fare la propria offerta, superiore al prezzo di partenza, e rilanciare con una nuova offerta nel caso in cui un altro offerente abbia fatto un'offerta superiore;
  3. l'asta termina quando nessun partecipante rilancia con una nuova offerta. Il bene viene venduto all'offerente che ha fatto l'offerta maggiore e il prezzo di vendita è quello offerto dal vincitore. Nel caso in cui nessun partecipante faccia un'offerta, il bene rimane al venditore.

  La strategia dominante per un agente consiste nel rilanciare le offerte degli altri partecipanti di una piccola somma finché il prezzo del bene non raggiunge il prezzo stimato dall'agente stesso.

- **Asta olandese.** L'asta tedesca è una tipologia di asta al ribasso e si sviluppa secondo i seguenti passaggi:
  1. il banditore mette all'asta il proprio bene ad un prezzo di partenza molto alto;

2. il banditore diminuisce continuamente il prezzo di vendita di una piccola somma, finché un agente fa un'offerta che eguaglia il prezzo attuale;
3. il vincitore è l'offerente che per primo fa un'offerta e il prezzo da lui pagato è pari all'offerta fatta.

In generale non esiste una strategia dominante per quest'asta, ma una buona strategia da seguire per un agente consiste nell'offrire meno del valore da lui stimato; quanto in meno dipende da quanto pensa offriranno gli altri agenti. Infatti, lo scopo di un agente è di ottenere il bene al minor prezzo possibile e quindi cercherà di realizzare l'offerta minima che vinca l'asta.

- **Asta di Vickrey.** La meno intuitiva fra le aste presentate in questo capitolo è sicuramente l'asta di Vickrey, un'asta classificabile come *second-price sealed-bid*. I passi di cui si compone quest'asta sono i seguenti:
  1. il banditore mette all'asta un proprio bene;
  2. ciascun agente fa la propria offerta;
  3. il vincitore è l'agente che ha presentato l'offerta più alta ma paga quanto offerto dal secondo miglior offerente.

Il punto importante per quest'asta è proprio il prezzo pagato dal vincitore. Il vincitore, infatti, non paga il prezzo da lui offerto ma paga il prezzo offerto dal secondo miglior offerente.

3.2.3 Decisional Agent

Un agente decisionale è un agente intelligente che comprende le condizioni del sistema in cui è coinvolto e prende decisioni sul comportamento da intraprendere a seconda dello stato del sistema. È quindi un agente che opera delle scelte valutando:

- le condizioni ambientali in cui si trova;
- le azioni già intraprese da altri agenti;
- le azioni che gli altri agenti svolgono in quel momento;
- le azioni future che gli altri agenti sceglieranno una volta compiute determinate operazioni.

Un agente che ha capacità decisionale è pertanto in possesso di informazioni asimmetriche, tali per cui necessita di un piano strategico di azione per raggiungere l'obiettivo preposto.

Un piano strategico è un piano completo che delinea il comportamento da intraprendere una volta che si manifestano determinate condizioni; presenta pertanto stati aggiuntivi che il dispositivo in cui risiede l'agente non può raggiungere.

Il problema chiave che quindi si presenta ad un agente e quello della scelta dell'azione migliore da intraprendere. la difficoltà nel prende-re una decisione può essere dovuta alle diverse caratteristiche dell'ambiente, quindi per capire meglio i comportamenti associati ad un agente è necessario prima capirne il suo dominio d'azione.

Il concetto di ambiente.

L'ambiente rappresenta l'infrastruttura computazionale all'interno della quale gli agenti si muovono, operano ed interagiscono tra loro.

Esso può essere classificato in base alle seguenti caratteristiche:

- accessibile vs. non accessibile: un ambiente accessibile permette all'agente di svolgere qualsiasi tipo di operazione, di ottenere informazioni complete e aggiornate. La maggior parte degli ambienti (quali il mondo reale o internet) non offre un tale livello d'accessibilità;
- deterministico vs. non deterministico: un'azione in un ambiente deterministico ha un effetto noto a priori. In un ambiente non deterministico la stessa azione, eseguita in istanti diversi, può avere effetti differenti;
- statico vs. dinamico: un ambiente statico rimane costantemente immutato durante l'inattività dell'agente. Di contro un ambiente dinamico e sottoposto all'azione di molti processi e quindi non e sotto il controllo del singolo agente. il mondo reale e internet sono esempi di realtà estremamente mutevoli.
- discreto vs. continuo: un ambiente è discreto se è possibile compiere su di esso un numero fisso di azioni. un maggiore livello d'accessibilità di un ambiente facilita la costruzione di agenti che operano su di esso. infatti, considerando che un buon agente deve essere in grado di fare le scelte più giuste e che la qualità delle decisioni dipende dalla qualità delle informazioni a lui accessibili, una maggiore fonte di informazioni aumenta le probabilità di produrre risultati accurati. un ambiente deterministico produce un risultato ben definito e univoco, mentre un ambiente non deterministico pone l'agente di fronte alla possibilità di fallimento nel conseguire i propri obiettivi e ne limita la sfera d'influenza. il non-determinismo e strettamente correlato col dinamismo.

la classe d'ambienti più complessa da gestire e costituita da ambienti inaccessibili, non deterministici, dinamici e continui. Gli ambienti con queste caratteristiche sono chiamati aperti.

Consideriamo l'agente all'interno di un sistema percettivo e definiamo due funzioni:

- see: rappresenta l'abilità dell'agente nell'analisi delle condizioni in cui si trova. L'output della funzione è una 'percezione' del sistema

- action: rappresenta la decisione di come l'agente stesso deve procedere

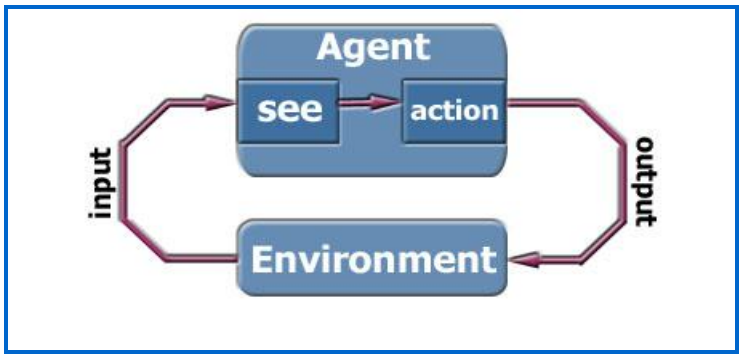


Figura 3.1 – Sistema di un agente decisionale reattivo

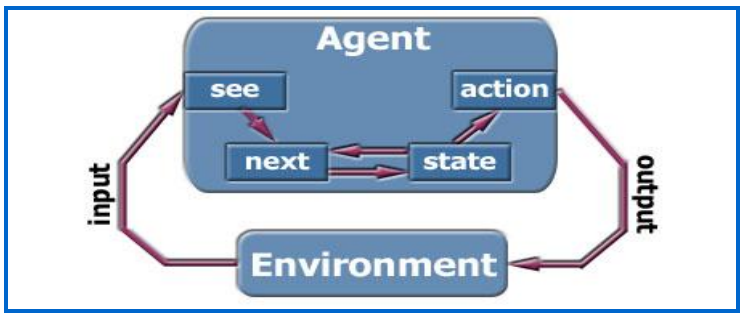


Figura 3.2 – Sistema di un agente decisionale che esamina le informazioni immagazzinate degli stati precedenti e della sua storia

In questo secondo sistema viene introdotta una nuova funzione `next()` che mappa uno stato interno dell'agente e una 'percezione' ottenuta dall'analisi dell'ambiente in un nuovo stato interno.

Un agente generico che acquisisce la capacità di tipo decisionale a seconda della taglia può assumere tutti i comportamenti associati ad un agente specifico. In alcuni casi quando la capacità decisionale dipende solo da informazioni sugli stati correnti un agente generico implementa comportamenti senza considerare le informazioni riguardanti la history degli agenti esterni con una riduzione notevole delle computazioni e dei probabili overhead delle comunicazioni. Se da un lato questo comporta all'elaborazione di una quantità molto inferiore di informazioni che l'agente ottiene dai nodi esterni, dall'altro la decisione che verrà poi presa dall'agente potrebbe non essere la soluzione ottimale. Il nodo potrebbe infatti ritrovarsi in condizioni già riscontrate e ripetere soluzioni non idonee

Agente decisionale e MANET.

Un agente decisionale all'interno di una rete mobile è un agente che associa alla capacità decisionale anche capacità funzionali. È un agente che a seconda

dell'obiettivo da raggiungere compie decisioni a seconda dello stato del dispositivo in cui si trova durante la migrazione. Decisioni che devono essere reattive a seconda dei cambiamenti delle condizioni.

3.3 Agenti Funzionali

Per agenti funzionali s'intendono agenti che hanno un ben determinato compito specifico che non muta a seconda del mutamento della rete. la dinamicità di questi comportamenti è data dall'aggiornamento continuo di eventuali strutture dati contenute e dalla gestione dei fallimenti che nella norma prevede lo spostamento degli agenti presso altri dispositivi.

Generic Service Agent e MANET.

Un agente che fornisce un servizio generico migra nel momento in cui richiede alcuni servizi esterni non presenti nel dispositivo. La mobilità viene gestita quindi nel momento della richiesta del servizio al router e nel momento in cui si verifica un fallimento come per tutti gli altri agenti.

Nelle sezioni successive vengono trattati alcuni tra le principali funzioni che un agente può assumere.

3.3.1 Locator Agent

Un agente localizzatore è un agente che individua i nodi all'interno di un certo raggio della rete e traccia in una struttura dati interna:

- l'identificativo dei container esistenti;
- l'identificativo dell'agente, ovvero l'AID in JADE. Questo è formato dall'indirizzo della piattaforma in cui l'agente si trova e dal nome dell'agente all'interno del dispositivo;
- il servizio che l'agente fornisce.

In questo modo la rete rilevata viene gestita attraverso una struttura dati che memorizza indirizzi e servizi dei nodi connessi. Questo agente facilita e velocizza le interrogazioni da parte del router per l'inoltro di agenti in subnet diverse.

Un agente generico acquisisce funzionalità di locator quando deve raggiungere un host e nel cammino esegue tutte le computazioni per trovare il nodo in cui dirigersi. Si tratta di un agente che on-demand ricerca il path più conveniente e che in casi più complessi può anche duplicarsi per il calcolo di paths alternativi da intraprendere in caso di fallimenti e guasti della rete. Un agente generico mantiene al suo interno una struttura di dimensioni ridotte che traccia al suo interno i dati relativi ai nodi toccati nella ricerca del servizio richiesto.

La differenza principale da un agente specializzato è dettata dal momento in cui l'agente trova il dispositivo che soddisfa la sua richiesta. Una volta eseguite le computazioni appropriate l'agente ritornerà al nodo sorgente seguendo la tabella delle connessioni senza esaminare i nodi restanti connessi al dispositivo.

Locator Agent e MANET.

Un agente localizzatore specializzato è abbastanza pesante i termini di memoria: a seconda della numerosità dei nodi che popolano il raggio della rete in cui è situato, la struttura dati che memorizza le informazioni sui servizi può essere di dimensioni elevate. Il vantaggio è legato all'efficienza fornita nel calcolo di cammini multipli per raggiungere la destinazione: all'agente computazionale vengono fornite informazioni sul cammino più efficiente in termini di tempo e spazio e i cammini alternativi da intraprendere in caso di fallimenti. Può essere infatti utilizzato per fornire l'itinerario prima della migrazione e i cammini alternativi in caso di fallimenti riscontrati durante il percorso.

I dispositivi mobili con risorse a capacità maggiore, ad esempio tablet pc, smart phones, utilizzano questa tipologia di agente per ottimizzare le prestazioni.

Al contrario, data l'elevata frequenza con cui la topologia della rete cambia, risulta conveniente, soprattutto per i dispositivi con scarsità di risorse, calcolare i cammini on-demand, sfruttando la capacità di clonazione per il calcolo dei paths alternativi.

3.3.2 Router Agent

Un agente di inoltro è un agente che consente di inoltrare agenti tra host di subnet diverse. Ciascun router deve essere in grado di riconoscere gli agenti e di inoltrarli correttamente. In JADE il meccanismo di inoltro si implementa comunicando all'agente *incoming* un oggetto di tipo Location, che identifica il container dove migrare, e l'identificativo o AID dell'agente che fornisce il servizio nel container stesso. Sarà poi l'agente richiedente a eseguire il metodo `doMove(Location c)`.

Router Agent e MANET.

Un Router Agent viene impiegato quando la numerosità dei dispositivi mobili all'interno della rete supera un certo valore limite. La sua gestione è affidata a un dispositivo avente risorse più performanti e a capacità maggiore. Tramite un algoritmo di selezione tra i dispositivi più 'forti', viene eletto il nodo che attiverà l'agente router per inoltrare gli agenti ai nodi successivi.

In caso di fallimenti del nodo in cui il router è sito, l'agente può migrare in un secondo dispositivo all'interno del raggio della subnet e aggiornare la propria routing table.

L'esistenza della routing table dipende dall'eventuale appoggio del Router Agent al Locator e da quali politiche di gestione vengono adottate per una maggiore

efficienza date le condizioni. Un agente generico acquisisce funzioni da router quando la rete non presenta un'elevata quantità di dispositivi mobili, ma per efficienza rispetto anche ad una scalabilità di tipo spaziale, deve essere suddivisa in piccole aree. Funzioni semplificate di inoltro e aggiornamento connessioni possono gestire subnet di entità minore, in quanto diminuisce la probabilità di fallimenti. I messaggi e i metodi sono i medesimi di un agente specializzato con la differenza sulle ridotte dimensioni della routing table e la diminuzione dei tempi di ricerca e di farwarding di un agente.

3.3.3 Searcher Agent

Un agente di ricerca è un agente che analizza dati e documenti appartenenti al nodo in cui si trova rispetto ad una ricerca specifica commissionata dal dispositivo di appartenenza.

Per documento s'intendono tutte le tipologie di file: testuale, multimediale, semi-strutturato e strutturato. L'agente in questo ambito ha il compito di ricercare un bisogno informativo dell'utente.

L'informazione è un dato che ha significato all'interno di un contesto, proveniente da fonti strutturate o semi-strutturate; ad esempio un database contiene i dati che vengono interrogati dall'agente per produrre informazione. S'introduce il concetto di catalogazione semantica, che si pone ad un livello superiore rispetto alla catalogazione letterale, creata con un qualsiasi DBMS. I campi che vengono analizzati per ottenere informazione devono quindi essere indicizzati sia letteralmente sia secondo una propria ontologia per raffinare la ricerca di base.

Uno dei passi da compiere per fornire questo tipo di servizio è l'integrazione di sorgenti eterogenee, rese disponibili da terze parti, come siti web, database aziendali o anagrafici e archivi di ogni tipo.

L'agente ricercatore deve quindi essere fornito di un supporto per ricerche a livello semantico in grandi archivi di testi non strutturati, con particolare riferimento ai dati provenienti dal web.

Nella realtà esistono tre principali categorie di sorgenti di dati soggette all'integrazione:

- sorgenti contenenti dati strutturati, come un database ad accesso pubblico;
- sorgenti contenenti dati non strutturati, come una pagina web contenente del testo;
- sorgenti contenenti dati semi-strutturati, come un documento XML.

Un agente ricercatore specifico è un agente che ottimizza la ricerca sotto gli aspetti di efficienza temporale e precisione dei risultati ottenuti. Può implementare tutti i metodi di ricerca attraverso dei comportamenti specifici che sceglierà a seconda dell'accuratezza della query e dell'intenzione dell'agente richiedente.

Un agente specializzato per questo scopo può essere anche attivato da un agente generico che sta effettuando altre operazioni all'interno del dispositivo e necessita di informazioni provenienti da fonti esterne. Un agente di ricerca general purpose implementa generalmente una singola classe di strategia di ricerca che dipende dall'informazione ricercata. Per ridurre il consumo di risorse e eliminare eventuali tempi di attesa prolungati , implementa principalmnete il modello booleano di ricerca o eventualmente un modello vettoriale ridotto. Il modello ridotto implica dimensioni dei vettori esigue e calcolo semplificato del grado di similarità tra query e documenti che si avvicina alla logica binaria del yes or not.

Accenni ai metodi di ricerca.

Di seguito vengono elencati i principali metodi di ricerca e le caratteristiche che li rappresentano:

- **Boolean Model**: La strategia di ricerca si basa su un criterio binario di decisione, senza alcuna nozione sul grado di rilevanza: un documento è considerato rilevante o non rilevante
- **Simple Indexing Model**: Viene introdotto il concetto di bag-of-words che rappresenta l'insieme dei documenti in cui viene effettuatala ricerca e delle query che si devono eseguire per ottenere i dati voluti. Questo modello ignora l'ordine delle parole nell'oggetto di ricerca (parole chiavi, frasi complete sintatticamente corrette), la morfologia e la sintassi (singolari/plurali, femminile/maschile, etc.). La ricerca prevede il mero conteggio del numero di match tra le parole nel documento e l'oggetto della query.
- **Vector Space Model**: La strategia è basata sulla rappresentazione vettoriale dei documenti e delle query ai quali viene assegnato un peso. La ricerca si appoggia ad una matrice di similarità nella quale ogni colonna rappresenta un documento e ogni record un termine di ricerca. La matrice viene popolata con il conteggio dei match di ogni termine per documento. La ricerca si svolge calcolando il 'grado di similarità' tra il vettore delle query e ogni vettore rappresentante un documento, secondo la seguente formula:

  $Sim_{Doc\_n,Query} = Doc\_n \cdot Query$

  $= ct_{1n} \times q_1 + ct_{2n} \times q_2 + \ldots + ctMn \times q_M = \Sigma_m ct_{mn} \times q_m$

  Dove $ctij$ rappresenta il conteggio dei match del termine all'indice ij della matrice di similarità, e qk la query k. Questa funzione viene poi normalizzata secondo la funzione coseno in quanto la lunghezza dei documenti è molto variabile.
- **Index term**: Questa tecnica si basa sull'importanza data ad un termine. Parole troppo frequenti producono meno informazione di altre, parole troppo poco ripetute sono troppo specifiche e non sarebbero un buon indice di ricerca. Viene introdotto il concetto di *Inverse Document Frequency* (IDF)

che stabilisce come i documenti siano caratterizzati da termini, relativamente rari in altri documenti. Se una parola si ripete frequentemente nella maggior parte dei documenti, la sua probabilità di essere utilizzata come index term scende a zero. IDF è rappresentata dalla formula:

$idf_i = log (|D| / |\{d:t_i \text{ appartiene } d\}|)$

dove $|D|$ è il numero dei documenti nella collezione, e $|\{d:ti \text{ appartiene } d\}|$ è il numero di documenti in cui appare il termine ti.

La frequenza inversa viene poi normalizzata con la lunghezza del documento attraverso un operazione di moltiplicazione. Se $l_d \times idf_i$ è un valore alto allora la frequenza del termine in quel documento è alta mentre nel resto dei documenti è minore.

Searcher Agent e MANET.

La migrazione di un Searcher Agent è basata sulla successione di siti da visitare che riceve in input. Tale lista viene spesso definita *itinerary* e può essere definita staticamente alla creazione dell'agente o dinamicamente, ovvero costruita dalle informazioni raccolte durante la migrazione e la ricerca. Questa tipologia di agente in una rete mobile ha anche particolari funzioni di clonazione: crea un nuovo agente per ogni link esterno trovato. Il nuovo agente deve a sua volta ricercare pagine interessanti a partire dal link esterno per cui è stato creato. Questi sub-searcher agent contengono al loro interno il path già percorso dall'agente master fino a quel nodo e i cammini alternativi per il ritorno al client di appartenenza.

3.3.4 Tracker Agent

Un agente di tracking viene impiegato nella gestione dei fallimenti rispetto ad un agente o più agenti entro un certo threshold. Gestisce al suo interno le registrazioni di tutti i cambiamenti di stato a cui incorre un agente mobile che attraversa più e più nodi della rete.

Per ottimizzare la taglia, ad ogni registrazione, l'agente verifica se effettivamente lo stato di quell'agente è variato, in caso contrario non registra i valori delle variabili. Se vengono gestiti più agenti i diversi tracciamenti possono essere memorizzati tramite tabella hash: attraverso questa struttura l'indicizzazione diventa notevolmente efficiente e di conseguenza migliora le prestazioni di ricerca e di inserimento.

Tracker Agent e MANET.

Con le recenti ed efficienti tecniche di predizione dei fallimenti il tracker segue l'agente nel suo itinerario, migrando in un momento successivo. Si trova così nello stesso dispositivo per la durata necessaria a trascrivere il nuovo stato. Può essere

dislocato in altri dispositivi all'interno del raggio di copertura in caso di alti livelli di sicurezza a cui gli agenti si devono attenere. Se vengono toccate più subnet il Tracking Agent lascia una copia sull'ultimo nodo visitato della rete e si replica. Viene eliminata la storia precedente e la memorizzazione ricomincia dall'ultimo nodo della subnet incontrato dove avviene la clonazione.

3.4 Agenti infrastrutturali

3.4.1 Rules-based Agent

Si parla di agenti i cui comportamenti sono gestiti mediante l'utilizzo di regole. Il comportamento assunto in quel momento può dipendere, infatti, dalla situazione passata, corrente e/o futura in cui l'agente si trova. All'analisi di una condizione corrisponde quindi una specifica azione:

`if` *`condizione`* `then` *`azione`*

Questa tipologia di comportamento si applica correttamente e risulta efficiente quando l'ambiente è completamente osservabile, ovvero lo si conosce in ogni suo aspetto. La percezione cognitiva è completa e permette di analizzare ogni situazione in cui l'agente può operare. Per ambienti parzialmente osservabili l'agente mantiene una struttura dati al suo interno nella quale viene registrata la struttura della rete, dell'ambiente non conosciuto, man mano che esso viene osservato.

3.4.2 Utility-based Agent

Un agente strutturato in base all'utilità è un caso particolare di agente basato sulle regole. Un behaviour strutturato in base all'utilità è previsto quando l'ambiente cicostante offre condizioni generiche. L'agente si trova di conseguenza di fronte alla scelta di più strade, per completare il proprio compito e computa una funzione di utilità. Questa funzione viene utilizzata per quantificare l'ottimalità di uno stato interno che l'agente può assumere, ovvero analizza quanto quello stato lo *soddisfa* e gli permetta di raggiungere l'obiettivo. Al termine delle computazioni si procede alla scelta dell'opzione che ha la funzione di utilità maggiore di un threshold stabilito ma non necessariamente con valore massimo. L'obiettivo, infatti, è quello di massimizzare il valore della funzione di utilità a seconda della funzione stessa utilizzata e del suo andamento.

3.5 Agenti e meta-funzionalità

3.5.1 Delegation

Un agente che delega è un agente che conferisce permessi e autorità ad un altro agente per svolgere determinate funzioni che possono essere assegnate dall'agente stesso o appartenere al sotto-agente. Si definiscono principal o master agent l'agente che delega, e slave agent o sub-agent l'agente delegato adibito al compito. L'agente master cerca quindi di raggiungere un obiettivo attraverso le azioni svolte da un altro agente o più agenti.

Un agente generico sviluppa principalmente la delegazione stretta, ovvero gestisce esplicitamente la richiesta di 'aiuto' da parte del principal.

Come nel caso specifico anche un agente generico funzionale si appoggia ad un locator agent o acquisisce le funzioni adeguate per identificare i nodi e gli agenti che forniscono il servizio a lui necessario. L'agente invia una richiesta di delega che verrà accettata o meno dall'agente. Si può incorrere ad un accordo se l'agente incaricato, analizzato il contesto e la situazione, percepisce dei possibili benefit o meno dall'esecuzione del compito.

Delegation agent e Manet.

La delegazione è una tecnica per rendere una rete mobile ulteriormente distribuita. Si parla di elastic processing ovvero la capacità di eseguire programmi, anche gli stessi, in più punti della rete ottenendo migliori prestazioni dal punto di vista temporale e di performance.

La delega viene inoltre utilizzata per il calcolo parallelo di più agenti che eseguono codice anche non corrispondente su più punti della rete. Questi agenti ad operazioni concluse comunicano al master agent i risultati ottenuti.

3.5.2 Cloning

La clonazione è una forma di replicazione e trova ampio utilizzo all'interno delle reti mobili. Questo meccanismo viene sfruttato in diversi ambiti, come ad esempio nel campo dell'information retrieval o in quello dedicato alla fault-tolerance.

Nel primo caso la clonazione viene intrapresa ogni qual volta viene recuperato un indirizzo che punta verso un documento esterno alla fonte dati visionata. L'agente incaricato del compito provvede quindi alla creazione di una sua copia per intraprendere una nuova ricerca a partire dal nuovo dominio. La clonazione in questo caso è l'implementazione, nell'ottica degli agenti, di una procedura ricorsiva che riunisce i risultati ottenuti da ogni ramo della ricorsione, ovvero dalle copie dell'agente, nella fase finale.

Nel secondo caso la clonazione permette la tolleranza ai fallimenti. Con la ridondanza dei dati o degli agenti, che hanno il compito di elaborare compiti specifici, le copie vengono registrate all'interno di siti differenti. Se l'agente in esecuzione viene corrotto o il suo stato incorre in fallimenti viene rimpiazzato in tempi impercettibili da una sua copia.

La capacità di clonazione quando viene espressa aggiungere circa un overhead del 30% rispetto al mantenimento di un'unica copia. Il sovraccarico raggiunge comunque una percentuale inferiore (18%) nel caso l'agente abbia un itinerario di almeno tre nodi, nei quali non si verifica nessun fallimento. Un agente replicato che è in grado di tollerare un guasto in uno stadio è da tre a quattro volte più costoso di un unico agente. Inoltre l'aumento del tempo di esecuzione dell' agente è causato principalmente dai costi supplementari di comunicazioni instaurate dalle copie e dalla migrazione che questi agenti eseguono.

3.5.3 Security

La sicurezza ha assunto un importante ruolo nelle reti mobili ad-hoc, le quali riconfigurandosi continuamente risultano molto suscettibili ad attacchi provenienti dall'esterno. L'assenza di una infrastruttura fissa centrale e le risorse limitate peggiorano la situazione rendendo le MANET altamente vulnerabili alle minacce alla sicurezza.

Per sviluppare l'affidabilità di una rete mobile vengono inseriti nella rete dei Key Distribution Center distribuiti. Il problema sorge quando il numero di dispositivi aumenta e le chiavi devono essere aggiornate ogni qualvolta un nodo accede o lascia la rete. Questo porta ad alcuni problemi come l'aumento del traffico e la quantità di calcolo da eseguire per l'aggiornamento. Al fine di risolvere questo problema, i dispositivi vengono classificati per livelli di sicurezza. Si ottiene una struttura topologica della rete definita multi-cluster, in cui viene selezionato un cluster head, ovvero un nodo leader all'interno del gruppo, responsabile della gestione delle chiavi e della *fiducia*. Si distinguono tre tipologie di cluster head a seconda dei ruoli che il programmatore gli affida:

1. il leader viene utilizzato solo come nodo di distribuzione delle chiavi. Tutti i nodi comunicano con crittografia simmetrica e utilizzano una chiave unificata per realizzare l'autenticazione;
2. il leader ha il compito di fornire i certificati per le autorizzazioni e eventuali funzioni di certificazione per i nodi all'interno del proprio cluster;
3. il leader oltre ai compiti definiti al punto 2 può fornire certificati per comunicare con nodi appartenenti a cluster esterni. Se i dispositivi sono all'interno di un cluster si applica il meccanismo di chiave simmetrica per la realizzazione delle comunicazioni. Se i dispositivi appartengono a gruppi diversi devono utilizzare un processo incrociato per ottenere il certificato dal proprio head cluster e dal leader dell'altro gruppo coinvolto.

Utilizzando la tecnologia ad agenti per la programmazione e gestione dei dispositivi, il ruolo di cluster head viene assunto da un agente che attraverso comportamenti adeguati genera e distribuisce chiavi di sicurezza, fornisce certificati e autorizzazioni, etc., eseguendo tutti i comportamenti necessari per ottenere un adeguato livello di sicurezza.

3.5.4 Scalability

La scalabilità è una caratteristica che ben si adatta alla tecnologia ad agenti. Un'applicazione basata sugli agenti mobili è sicuramente molto più scalabile rispetto alle applicazioni monolitiche. Si evidenzia infatti l'efficienza della tecnica di programmazione modulare che incapsula caratteristiche comuni in behaviour che possono essere acquisiti dinamicamente dagli agenti, e non necessita l'eventuale riprogrammazione *from scratch* da parte dello sviluppatore, anche in ambienti molto scalabili. A differenza delle wired networks una rete mobile wireless non deve disporre di servizi centralizzati ma bensì servizi ristrutturati secondo la logica distributiva. Ad esempio, il componente principale di una rete fissa di grandi dimensioni è il router. Esso inoltra i pacchetti provenienti da un computer verso un altro dispositivo seguendo una tabella di routing che utilizza delle porte di connessione. In una rete distribuita, mobile, di grande dimensioni, questa caratteristica di router che inoltra i pacchetti viene rimpiazzata da una visione decentralizzata dei suoi compiti. Ogni dispositivo mantiene al suo interno un agente router che ha il compito di tracciare una piccola quantità di connessioni. La struttura creata avrà dimensioni logaritmiche rispetto alla rete e in cambio il percorso calcolato per raggiungere la destinazione sarà maggiormente lungo. In questo modo si raggiunge un compromesso nell'overhead introdotto dal routing che cresce al massimo in tempo logaritmo con la dimensione della rete.

4 Design Pattern

In questo capitolo vengono presentate le diverse tipologie di Design Pattern applicabili agli agenti nel contesto delle mobile ad-hoc networks. In particolare si procederà alla classificazione secondo alcuni criteri strutturali e procedurali che risultano fondamentali per una realizzazione corretta ed efficiente dei sistemi che il programmatore vorrà sviluppare.

4.1 Pattern applicati alla mobilità

Nella progettazione dei sistemi software esistono dei problemi che si presentano entro svariati campi di applicazione con diverse forme e varianti, ma aventi una struttura comune. Per questi problemi esistono delle soluzioni standard la cui utilità è stata dimostrata con l'esperienza di numerosi sviluppatori nell'ambito di progetti diversi. Con il binomio *design pattern* vengono chiamati tali modelli di soluzione che hanno dimostrato di funzionare e lavorare in maniera efficiente in più situazioni aventi caratteristiche comuni. Uno degli obiettivi principali della modelizzazione attraverso i design pattern è proprio la semplificazione del problema, rendendolo più chiaro nella comprensione e facilitando la risoluzione.

I pattern sono classificati in gruppi differenti sulla base del loro livello di astrazione. Per le reti MANET con tecnologia ad agenti consideriamo quindi la seguente distinzione:

- **Architectural Pattern.** Rappresentano modelli adatti alla progettazione dell'architettura delle comunicazioni tra gli agenti, ovvero determinano chi comunica con chi, nel momento in cui un agente raggiunge un host esterno

per la computazione locale. In particolare un pattern architetturale consiste di agenti, regole e vincoli che regolano i rapporti tra gli agenti stessi;

- **Behavioural Pattern.** Rappresentano strategie di progetto language independent, tipicamente object-oriented. Identificano un modello di programmazione da adottare ogni qual volta si presentano determinate condizioni; in questa tesi i pattern comportamentali sono rappresentati dai prototipi integrati nel framework;
- **Migration Pattern.** Rappresentano modelli procedurali che evidenziano la successione temporale nella migrazione degli agenti verso i nodi esterni.

Un design pattern non è un componente software, ma solo uno schema di soluzione per un particolare aspetto del funzionamento di un sistema. Gli elementi strutturali di un pattern rappresentano i ruoli che saranno interpretati dagli agenti effettivamente realizzati. Ciascuna di queste entità può interpretare un ruolo diverso in diversi pattern, poiché in un singolo sistema (o sottosistemi) si devono risolvere diversi problemi con diversi modelli. Inoltre, è bene tener presente che un pattern non è una struttura rigida da applicare meccanicamente, ma uno schema che deve essere adattato alle diverse situazioni.

In generale, i design pattern tendono ad essere usati in una fase del progetto, detta progetto dei meccanismi, intermedia fra il progetto di sistema e quello in dettaglio, molti pattern sono applicabili in tutto l'arco del progetto, mentre altri sono specifici per le diverse fasi affrontate.

La loro rappresentazione grafica permette di far comprendere il maggior numero di informazioni al programmatore che sta costruendo quel sistema.

4.2 Pattern Architetturali di Comunicazione

La realizzazione di buoni schemi strutturali di comunicazione prevede come primo passo l'associazione di agenti e ruoli, la definizione della numerosità per ogni tipologia e una rappresentazione delle comunicazioni che devono avvenire. Graficamente l'utente può, infatti, posizionare gli agenti in modo da realizzare uno schema e rappresentare lo scambio di messaggi attraverso una linea che unisce le coppie di agenti coinvolti nella conversazione, che saranno gli estremi del lato.

Nei paragrafi successivi vengono identificate le varie tipologie di pattern architetturali per quanto riguarda le comunicazioni.

4.2.1 Pattern Star

Il pattern Stella è uno dei modelli d'interazione primari per la definizione degli agenti comunicanti e dei ruoli che gli agenti stessi andranno a ricoprire.

In un pattern Star di N agenti, esiste un agente centrale collegato a N-1 agenti periferici non comunicanti tra di loro. Gli N-1 nodi periferici devono essere della medesima tipologia, e quindi avere lo stesso ruolo e similmente richiedere lo stesso servizio, fornito dal nodo centrale, necessario al raggiungimento del loro obiettivo.

Nel caso in cui gli agenti periferici siano eterogenei, offrano quindi servizi diversi, si ottiene una specializzazione del pattern Star ovvero il Branching pattern.

Questa topologia consente di realizzare reti con un numero di nodi anche elevato ma presenta maggiori vulnerabilità ai guasti: ad esempio, se consideriamo un fallimento del nodo centrale, questo compromette l'intera rete.

Caratteristiche delle comunicazioni.

- Il nodo centrale invia messaggi a N-1 destinatari.
- I nodi periferici hanno come unico destinatario il nodo centrale. Si parla di comunicazione punto-punto

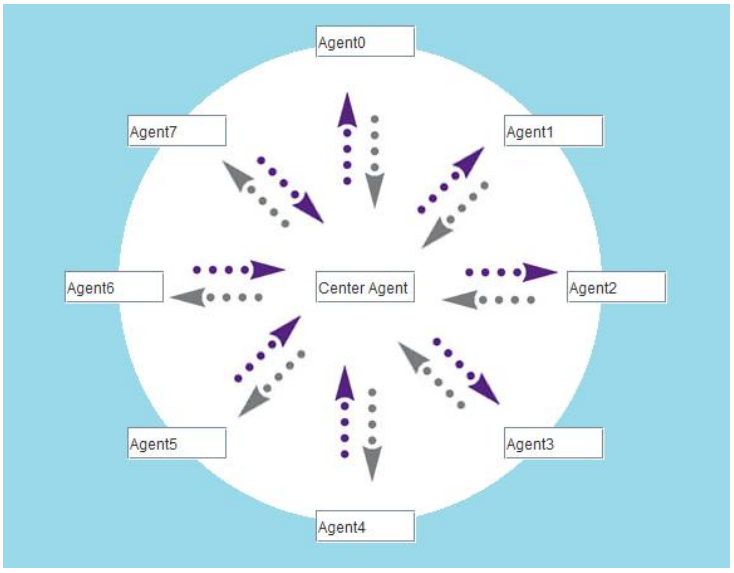


Figura 4.1 – Architectural Pattern Star a 9 nodi

Casi di applicazione.

La struttura a stella può essere applicata a diversi casi tra cui:

- Negoziazione in cui il center agent è rappresentato dal venditore e i peripheral agents dai compratori. Nel caso del Branching pattern il center agent è il mediatore mentre un agente tra i periferici compratori è il venditore.
- Gestione delle risorse, in cui il center agent assume il ruolo di gestore della risorsa richiesta dagli n-1 agenti esterni.
- Casi legati al pattern di migrazione a stella.

4.2.2 Pattern Pipeline

Il Pattern Pipeline è un altro schema d'interazione in cui i nodi si dispongono consecutivamente secondo una struttura linear array o secondo un cerchio interrotto, simulando un pattern Ring troncato. I servizi che gli agenti offrono in questo caso possono essere eterogenei o omogenei a seconda del contesto. Uno scenario tipo, rappresentabile da questo pattern, lo si identifica in un agente che esegue una migrazione *one-way*. L'agente, una volta raggiunta la destinazione ritorna alla sorgente passando per lo stesso cammino. Questo scenario avviene quando non accadono fallimenti di alcun genere o riconfigurazioni dinamiche della rete, che coinvolgono quel path, durante la mobilità.

Il vantaggio dato da una comunicazione unilaterale o bilaterale deriva dalla quantità minima di messaggi scambiati. Se da un lato questo aspetto favorisce il basso consumo delle risorse energetiche e la minima riduzione delle capacità di storage del dispositivo, dall'altro il tempo per ottenere un risultato dall'elaborazione cresce in modo lineare nel numero di agenti coinvolti.

Caratteristiche delle comunicazioni.

- I nodi agli estremi comunicano con l'unico nodo adiacente;
- I nodi interni comunicano con i due nodi adiacenti, rispettivamente a sinistra e a destra della sequenza di appartenenza.

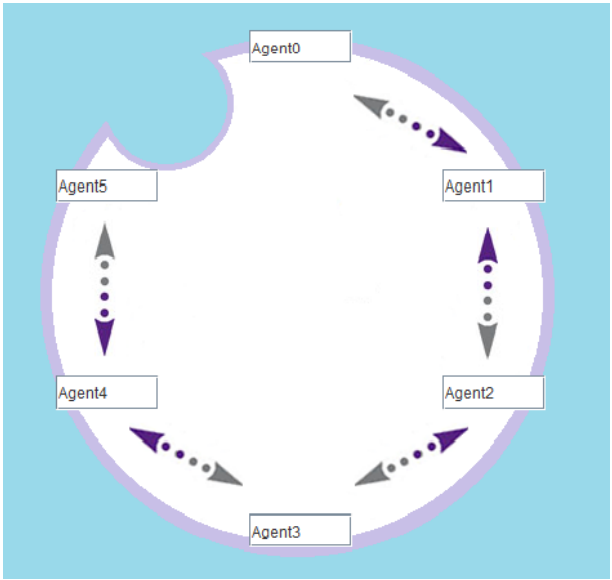


Figura 4.2 - Architectural Pattern Pipeline a 6 nodi

Casi di applicazione.

Un pattern a struttura lineare come la pipeline può essere utilizzato nei casi di richiesta gerarchica di servizi/risorse complesse che necessitano a loro volta di richiedere altre funzioni a supporto. Il pattern può essere considerato come un *one-way tree*.

4.2.3 Pattern Ring

Il modello ad anello è uno schema circolare nel quale gli agenti che interagiscono si distribuiscono lungo la circonferenza di un cerchio.

Se consideriamo come numero di agenti N, possiamo definire il pattern Ring come un pattern Pipeline i cui nodi estremi sono collegati tra di loro.

Anche per questo modello i servizi che gli agenti offrono possono essere eterogenei o omogenei a seconda del contesto. Questo pattern è molto diffuso nelle reti ad-hoc in quanto simula lo scenario di un agente che visita i (0<i<=N) nodi prima di raggiungere l'host terminale, e visita N-i nodi differenti nel cammino di ritorno. Questo avviene quando l'agente calcola un path alternativo per il ritorno, a causa di possibili mutazioni avvenute nella rete durante l'itinerario già percorso.

Caratteristiche delle comunicazioni.

- Tutti gli agenti della rete comunicano con i due agenti adiacenti, rispettivamente a sinistra e a destra della sequenza di appartenenza.

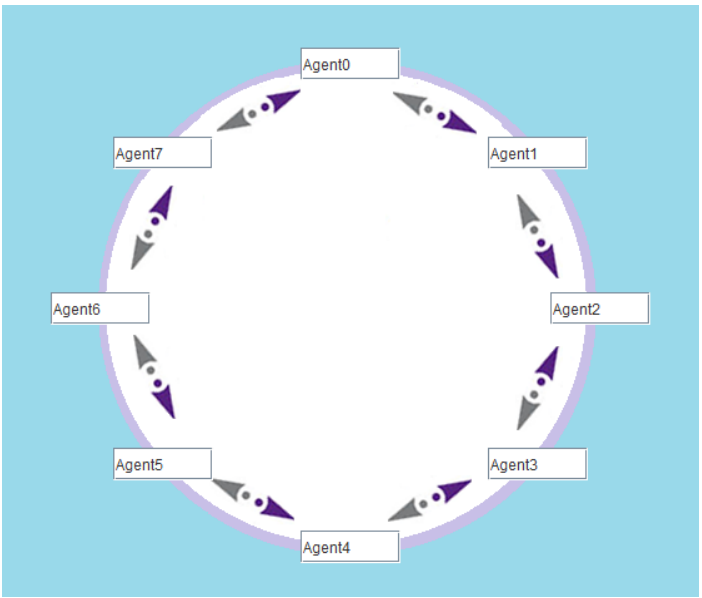


Figura 4.3 - Architectural Pattern Ring a 8 nodi

4.2.4 Pattern Tree

Il Pattern Tree è un albero binario il cui albero completo ha altezza logN, dove N è il numero di tutti i nodi.

I servizi che gli agenti offrono in questo caso possono essere eterogenei o omogenei a seconda del contesto. La struttura presentata ha un duplice vantaggio: il numero di messaggi scambiati non supera mai il threshold di tre e, se consideriamo ogni agente situato su un diverso dispositivo, la migrazione coinvolge logN nodi, riducendo notevolmente la probabilità di incorrere in un fallimento.

Caratteristiche delle comunicazioni.

- Il nodo radice comunica con i due nodi a livello inferiore
- I nodi interni comunicano con 3 dispositivi, uno a livello superiore, due a livello inferiore (caso particolare di Branching Pattern)
- I nodi foglia comunicano con un unico dispositivo padre a livello superiore.

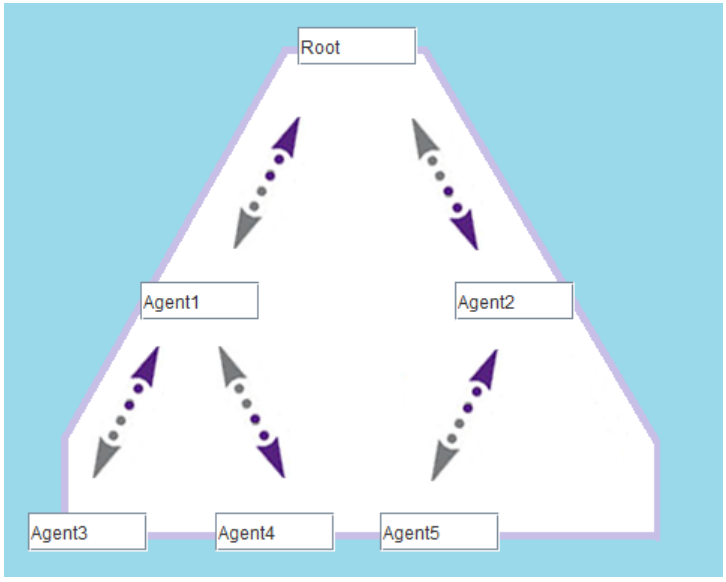


Figura 4.4 - Architectural Pattern Tree a 6 nodi

Casi di applicazione.

Questa struttura può essere utilizzata in diversi casi, di cui ne citiamo alcuni:

- Organizzazione di reti amministrative gerarchiche dove i dispositivi hanno autorizzazioni e certificati diversi;
- Simulazione di meccanismi di delega, dove un agente master incarica uno o più agenti ad eseguire metodi specifici per poi riportare i risultati all'agente padre;

- Information retrieval, meccanismo di ricerca web supportato dalla capacità di clonazione per effettuare ricerche su fonti esterne rispetto quella principale di partenza.

4.2.5 Pattern Grid

Il Pattern Grid è un reticolo che implementa una struttura di connessioni a matrice con disposizione di N^i nodi sulle righe per N^j colonne. È un pattern di struttura secondaria in quanto a seconda della numerosità di agenti presenti può essere costituito da insiemi di pattern star, pattern ring o pipeline.

I servizi che gli agenti offrono in questo caso possono essere eterogenei o omogenei a seconda del contesto. Inoltre offre un buon compromesso tra numerosità degli agenti che comunicano e degrado delle prestazioni dovute al carico del sistema.

Caratteristiche delle comunicazioni.

- Gli agenti angolari comunicano con soli due agenti del reticolato, simulando una struttura pipeline/ring;
- Gli agenti borderline comunicano con 3 nodi e implementano un pattern a stella a 4 nodi;
- Gli agenti interni comunicano con 4 nodi e implementano un pattern a stella a 5 nodi.

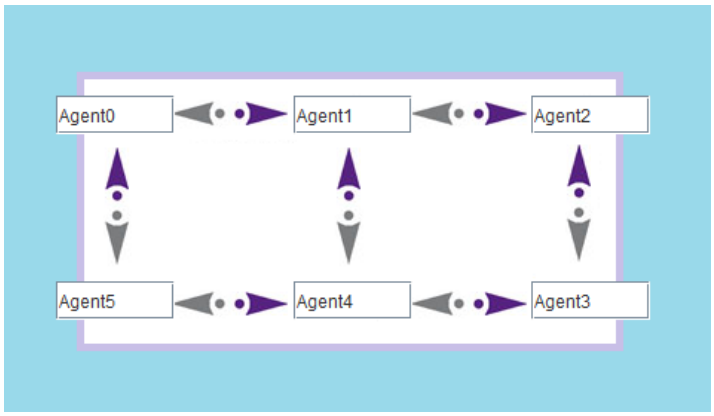


Figura 4.5 - Architectural Pattern Grid a 6 nodi

Il reticolato lo si può pensare composto anche da più anelli connessi, aventi due agenti in comune che utilizzano lo stesso canale comunicativo. Inoltre questa struttura può anche contenere più agenti intermedi tra un agente e l'altro.

Casi di applicazione.

Il pattern Grid può essere utilizzato in scenari che prevedono una qualsiasi applicazione basata sullo spostamento dell'utente all'interno di una città. L'area urbana, infatti, per una migliore gestione viene suddivisa in reticoli più o meno grandi, all'interno dei quali operano gli agenti che si trovano ai vertici di ognuno e che in ogni caso possono spostarsi su altri host per eventuali richieste ed elaborazioni.

4.2.6 Pattern Complete Graph

Il pattern descrive la struttura di una rete puramente peer to peer dove tutti i nodi sono connessi tra di loro. Le comunicazioni così gestite offrono prestazioni elevate in termini di velocità in quanto gli agenti che richiedono un servizio comunicano direttamente con l'agente di riferimento. Un altro vantaggio introdotto dai numerosi collegamenti è dato dalla massima robustezza rispetto ai guasti, ovvero, se un agente non è soggetto a isolamento, è sempre possibile determinare un percorso che colleghi due agenti qualsiasi della rete. lo svantaggio, al contrario, è rappresentato dl numero di collegamenti che è quadratico nel numero di agenti e questo limita di fatto l'applicazione del pattern a reti con un numero relativamente piccolo di agenti.

Inoltre, se consideriamo ogni agente depositato su un dispositivo diverso, incide notevolmente anche frequenza di riconfigurazione della rete, che introduce complessità alle operazioni di aggiornamento. Ogni nodo deve gestire la disattivazione degli agenti e gli allacciamenti di nuove connessioni quando al contrario nuovi agenti entrano nello stato di attivazione. Pertanto gli agenti risiedenti su un nodo devono richiedere con la stessa frequenza i provider dei servizi richiesti portando a dei rallentamenti nelle computazioni. Si parla di ritardi impercettibili inizialmente che possono portare con l'aumento del tasso di riconfigurazione ad un overload del sistema.

Caratteristiche delle comunicazioni.

- Ogni nodo comunica con N-1 nodi della rete. Si hanno N(N-1)/2 connessioni totali

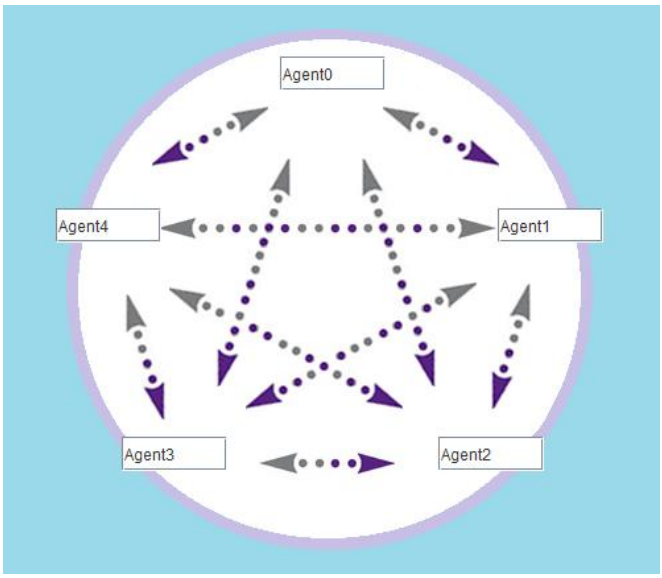


Figura 4.6 - Architectural Pattern Complete Graph a 5 nodi

Casi di applicazione.

Questo modello, proprio perché puramente peer to peer, può essere applicato a scenari di ricerca e download di file, sfruttando la mobilità degli agenti che ottimizzano le operazioni segnalate. Inoltre può essere associato a casistiche di configurazione di reti a bassa densità permettendo la comunicazione di tutti gli agenti coinvolti e allacciati alla rete.

4.3 Pattern Comportamentali

I pattern comportamentali rispecchiano per la maggior parte la struttura dei pattern architetturali evidenziando la temporizzazione degli spostamenti durante il processo di migrazione. Viene definita la successione dei nodi visitati e le fasi dello spostamento di un agente.

4.3.1 Star-Shaped

Questo modello riprende la struttura del pattern a Stella. Il comportamento dell'agente considerato prevede che l'agente si sposti presso un'agency esterna e poi ritorni a quella di origine per poi spostarsi nuovamente verso un diverso dispositivo, etc.. Il ripetersi di questa successione di eventi più e più volte dà vita alla tipica forma a stella propria di questo pattern. Lo scheduling degli spostamenti nodo-nodo è pertanto sequenziale.

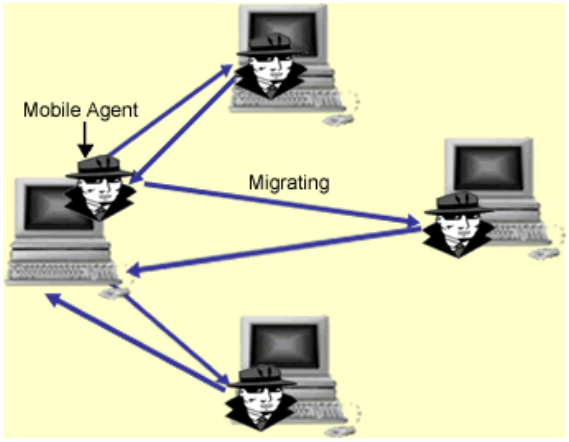


Figura 4.7 – Star-Shaped Pattern

4.3.2 Branching

Lo schema Braching rappresenta un modello a stella parallelo, ovvero all'agente vengono comunicate una serie di agency presso cui migrare, definite destinazioni. Al passo successivo l'agente clona se stesso indicando ad ogni clone uno degli indirizzi delle destinazioni. Queste copie iniziano il loro processo di migrazione parallelamente e ritorneranno poi al dispositivo di partenza per comunicare i risultati ottenuti dalle computazioni locali presso le agency. Lo scheduling temporale delle comunicazioni è pertanto parallelo, ogni clone esegue in maniera indipendente dagli altri.

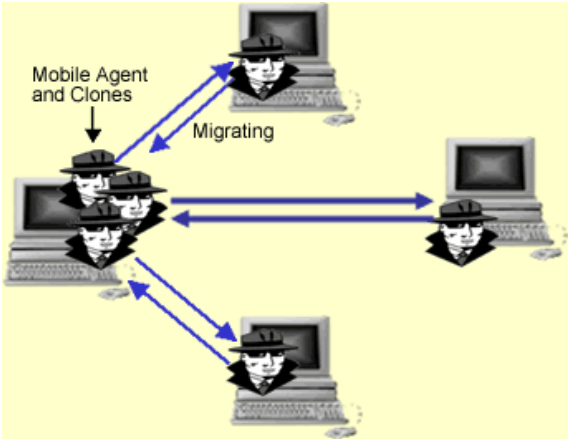


Figura 4.8 – Branching Pattern

4.3.3 Itinerary

Questo modello viene applicato quando un agente non deve calcolare on-demand i nodi su cui spostarsi ma gli viene comunicato prima della migrazione un itinerario, ovvero una sequenza di nodi che deve visitare. Una volta arrivato presso un nuovo

dispositivo, l'agente esegue il suo compito a livello locale e poi continua il suo itinerario. Dopo aver attraversato tutte le tappe identificate l'agente torna presso l'agency di origine. La migrazione avviene quindi sequenzialmente seguendo uno schema punto-punto di coppie di nodi differenziate.

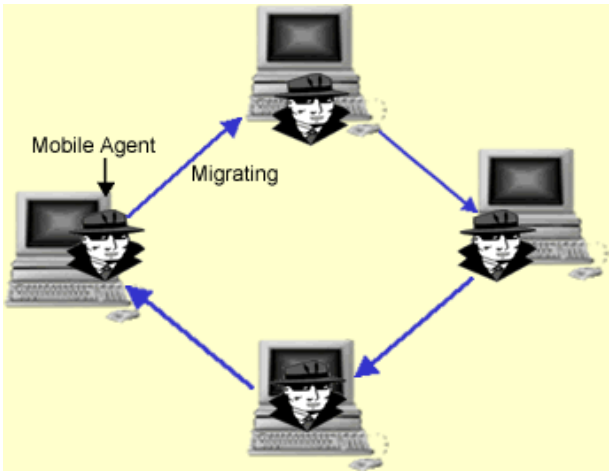


Figura 4.9 – Itinerary Pattern

4.3.4 Meeting

Questa tipologia di pattern si applica quando più agenti devono incontrarsi su un certo nodo. Il Meeting agent che implementa questa procedura di comunicazione, invia a tutti gli agenti che rientrano nella comunicazione l'indirizzo del dispositivo presso cui deve avvenire l'incontro e l'identificativo del meeting stesso. Successivamente gli agenti contattati e il Meeting agent stesso migrano verso il dispositivo identificato. Questo pattern viene utilizzato di norma per computazioni tra agenti che devono avvenire in un sito sicuro, che si avvale di meccanismi di sicurezza complessi ed efficienti per evitare attacchi esterni. Si evidenziano scenari di questo tipo quando si devono trattare dati strettamente riservati, ad esempio quando sono coinvolti dispositivi bancari o postali-

Un altro esempio di questo pattern si ha nel caso di un'asta, e nello specifico quando questa si deve svolgere presso un nodo diverso rispetto a quelli che contengono gli agenti interessati.

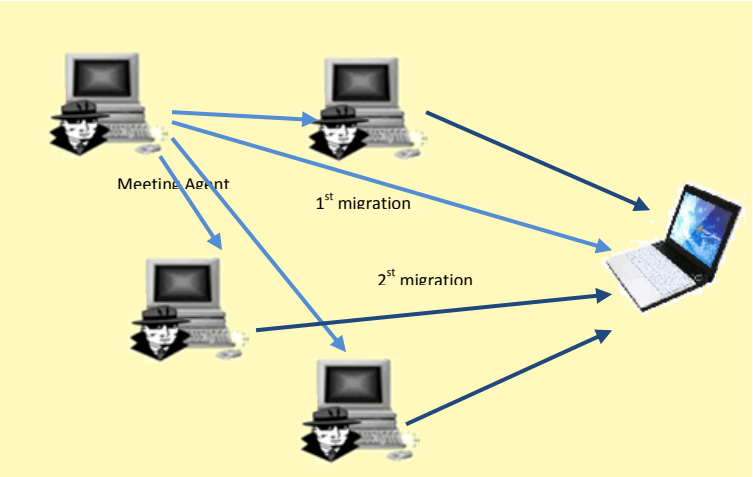


Figura 4.10 - Meeting Pattern

4.3.5 Master-Slave

Un pattern di comunicazione Master-Slave viene strutturato proprio come il nome stesso indica: un agente principale delega un agente inferiore, nella taglia e nelle capacità funzionali possedute, ad eseguire un determinata operazione presso un nodo esterno. In questo modo il dispositivo è in grado di risparmiare energia per l'invio dell'agente 'ridotto' e permette all'agente principale di continuare ad operare parallelamente, ottimizzando il tempo di esecuzione. Questo pattern implementa il modello da applicare nei casi in cui avvenga un meccanismo di delega.

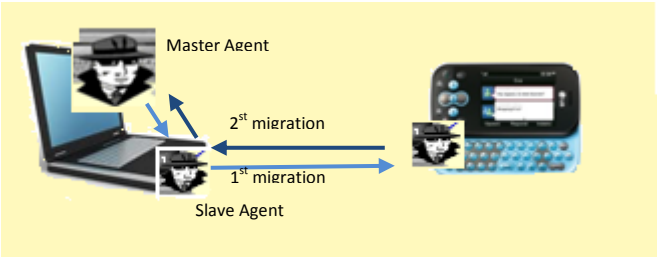


Figura 4.11 - Meeting Pattern

5 Progettazione del sistema

In questo capitolo si delinea la fase iniziale di modellizzazione e progettazione del framework. Nel paragrafo introduttivo si esplicitano obiettivi e requisiti che il sistema interattivo deve presentare, mentre, nelle sezioni successive, vengono presentate le specifiche dei prototipi realizzati e gli approcci adottati per l'utilizzo dei design pattern grafici.

5.1 Introduzione

5.1.1 Definizione degli obiettivi

Il lavoro presentato in questa tesi si pone l'obiettivo di realizzare un framework per la definizione, l'implementazione e la gestione di agenti software. Si focalizza quindi sullo sviluppo del componente relativo agli strumenti di sviluppo.

L'idea della progettazione di un sistema che supporti la programmazione ad agente, nasce dalla necessità di facilitare il compito iniziale dello sviluppatore. Nello specifico questo tool fornisce dei prototipi di agenti e pattern di design che cercano di appoggiare e semplificare il processo di analisi e progettazione di sistemi altamente complessi.

L'obiettivo principale è quello di velocizzare lo sviluppo del prodotto finito evitando di riscrivere codice già redatto in precedenza per compiti simili.

5.1.2 Requisiti

La progettazione e l'implementazione del software ha seguito un approccio modulare per permettere la riutilizzabilità del codice e la possibilità di aggiungere o togliere funzionalità alle entità sviluppate in maniera più veloce e semplice. Il sistema dovrà quindi presentare caratteristiche di:

- semplicità nell' utilizzo, offrendo un'interfaccia di uso intuitivo;

- chiarezza nell'esposizione dei contenuti che andranno a descrivere gli agenti e i modelli implementati;
- integrabile con nuove strutture agenti che il programmatore stesso andrà a inserire.

5.2 Prototipi realizzati

5.2.1 Agent

Un agente non è solo un oggetto intelligente, composto da codice attivo, ma un più complesso sistema computazionale che assume veri e propri comportamenti.

I prototipo presentato è uno schema iniziale, un punto di partenza per lo sviluppo delle varie specializzazioni.

Classi e elementi che compongono il prototipo.

- anAgent.java: Classe che crea un agente generico servizio e avvia i comportamenti dell'agente;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- GenericBehaviour.java: Classe che gestisce il comportamento che assumerà l'agente. Il comportamento è un'estensione della generica classe Behaviour.java padre di tutte le tipologie di comportamenti.

Parametri di GenericBehaviour.

- *@param a* – Agente proprietario del comportamento
- *@param param* – Eventuali parametri da associare all'agente

5.2.2 Auctioneer Agent

Classi e elementi che compongono il prototipo.

- AuctioneerAgent.java: Classe che crea un'istanza dell'agente mediatore e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- AuctioneerBehaviour.java: Classe che gestisce la negoziazione. Il comportamento estende la classe CyclicBehaviour.

Parametri di AuctioneerBehaviour.

Il comportamento non ha propri parametri in quanto l'agente riceve tutte le specifiche dall'agente venditore.

Struttura di AuctioneerBehaviour.

Il comportamento prototipo è un comportamento ciclico suddiviso in due sezioni in base allo stato dell'agente, gestito dalla variabile booleana `free`.

1. Caso `free = 'true'`: il mediatore è libero, ovvero non gestisce negoziazioni. In questa sezione l'agente attende richieste di vendita da Seller Agents. Se riceve una richiesta, imposta lo stato a false e si passa alla seconda sezione;
2. Caso `free = 'false'`: caso gestito al suo interno da un ciclo while che termina quando la negoziazione è arrivata ad una soluzione finale. Il ciclo while è a sua volta strutturato come una macchina a stati finiti, i cui stati sono scanditi dalla variabile step e dalla clausola switch(step):
   - step 0: l'agente ricerca compratori interessati al bene proposto dal venditore. Se trova degli acquirenti dà inizio alla negoziazione e passa allo step1: invia un messaggio di conferma al venditore e invia i dati iniziali per la negoziazione ai compratori; altrimenti risponde negativamente al venditore e ritorna allo stato libero (`free = 'true'`);
   - step 1: Passo di ricezione delle proposte. Il mediatore riceve le offerte base e i rilanci. Se riceve offerte passa allo step 2 altrimenti passa allo step 3;
   - step 2: Passo di gestione della negoziazione. Il mediatore esegue le computazioni opportune in base ai rilanci ottenuti e alla volontà del venditore, risponde a tutti gli agenti con la nuova proposta e ritorna allo step 1;
   - step 3: Passo conclusivo. Il mediatore non riceve più offerte, decreta il vincitore, comunica al venditore e ai compratori il vincitore e il prezzo di vincita e ritorna allo stato free.

Sintassi dei principali messaggi accettati.

- Messaggio di apertura di negoziazione ricevuto dai possibili venditori per la richiesta di apertura compravendita

```
(ACLMessage.REQUEST:
:sender Seller Agent[i]
:receiver Auctioneer
:content bene da vendere e dati relativi alla vendita
:conversation id "OPEN_NEGOTIATION"
)
```

- Messaggio accettato per la registrazione degli agenti partecipanti al conflitto corredato dell'invio della proposta base offerta dall'agente

```
(ACLMessage.AGREE:
:sender Agent[i]
:receiver Mediator
:content basic rise
:conversation id "OPEN_NEGOTIATION"
)
```

- Messaggio accettato per la gestione vera e propria della negoziazione. Contiene le puntate effettuate dall'agente

```
(ACLMessage.PROPOSE:
:sender Buyer Agent[i]
:receiver Auctioneer
:content rise
:conversation id "NEGOTIATION"
)
```

5.2.3 Buyer Agent

Per le strategie di negoziazione di un agente compratore, nel prototipo vengono utilizzate terminologie che si riferiscono all'acquisto di un bene, queste possono essere anche reinterpretate a seconda dello scenario in cui avviene la negoziazione, come l'allocazione delle risorse. In questi casi le strategie di acquisizione andranno diversificate dalle strategie di asta di cui si fornisce una breve descrizione nel paragrafo 3.2.2 .

Classi e elementi che compongono il prototipo.

- BuyerAgent.java: Classe che gestisce la lettura dei parametri dell'agente compratore e che avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- BuyerBehaviour.java: Classe che gestisce la negoziazione. Il comportamento estende la classe OneShotBehaviour.

Parametri di BuyerBehaviour.

- *@param a* – Agente proprietario del comportamento

- *@param budget* - Budget totale dell'agente proprietario
- *@param goods* – Beni di interesse da comprare/negoziare
- *@param prices* – Prezzi stimati per l'acquisto dei beni di interesse
- *@param overshoot* – Percentuale massima di sforo dal prezzo stimato, utilizzata come vincolo per terminare i rilanci dell'agente in gioco

Struttura di BuyerBehaviour.

Il comportamento prototipo è gestito al suo interno da un ciclo while che termina quando tutti i beni di interesse sono stati acquistati. Il ciclo while è a sua volta strutturato come una macchina a stati finiti, i cui stati sono scanditi dalla variabile step e dalla clausola switch(step):

- step 0: l'agente analizza l'array dei beni da acquistare e decide il bene corrente da richiedere. Se tutti i beni sono venduti va allo step 4, ovvero esce dal ciclo e termina la sua esecuzione, altrimenti passa allo step 1;
- step 1: questo step esegue due casi:
  - il compratore cerca venditori disposti alla vendita del bene corrente e invia richieste;
  - il compratore riceve messaggi da parte di un mediatore per la negoziazione di un certo bene;

  Se il bene è accettato si inizia la negoziazione e si accede allo step 2, altrimenti si rimane allo step 1 per un certo periodo dopo il quale si ritorna allo step 0 per la scelta di un altro bene;
- step 2: Passo di gestione della negoziazione. A descrizione del programmatore l'agente può negoziare direttamente con il venditore o con un mediatore coinvolto dal venditore;
- step 3: Passo che gestisce la risoluzione della negoziazione, viene decretato il vincitore ed effettuate le eventuali computazioni per aggiornare le variabili di stato dell'agente. Si ritorna poi allo step 0 per la richiesta di un nuovo bene;

Sintassi dei principali messaggi accettati.

- Messaggio di apertura di negoziazione ricevuto dal mediatore con la richiesta di compravendita

```
(ACLMessage.CFP:
:sender Auctioneer
:receiver Buyer Agent[i]
:content bene da vendere e dati relativi alla vendita
:conversation id "OPEN_NEGOTIATION"
)
```

- Messaggio accettato per la gestione della negoziazione e in particolare dei rilanci

```
(ACLMessage.PROPOSE:
:sender Auctioneer
:receiver Buyer Agent[i]
:content rise + current winner agent
:conversation id "NEGOTIATION"
)
```

- Messaggio accettato con la risoluzione del conflitto

```
(ACLMessage.INFORM:
:sender Auctioneer
:receiver Buyer Agent[i]
:content winner rise + winner agent
:conversation id "CLOSE_NEGOTIATION"
)
```

5.2.4 Seller Agent

Un agente venditore è un agente che possiede dei beni da vendere e vuole ottenere il massimo profitto dalla loro vendita. Può negoziare direttamente con i compratori o come nel prototipo realizzato affidarsi ad un agente esperto che funge da mediatore. Viene quindi dato un esempio del meccanismo di delega da un master agent, in questo caso il Seller agent, ad un slave agent, ovvero l'Auctioneer agent.

Seller Agent e MANET.

Di norma l'agente venditore non migra verso altri dispositivi. La migrazione può avvenire in caso di negoziazioni di beni esterni al dispositivo, tali per il processo può svolgersi in un altro dispositivo più performante; oppure in caso di fallimenti, attraverso l'invocazione del metodo `handleFailure()`.

Classi e elementi che compongono il prototipo.

- SellerAgent.java: Classe che gestisce la lettura dei parametri dell'agente venditore e che avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;

- LightSellerBehaviour.java: Classe che gestisce la delega della negoziazione ad un mediatore. Questo è il motivo per cui il comportamento ha assunto nella descrizione la parola *Light*, parola che vuole sottolineare la delega della gestione della negoziazione ad un agente più esperto. Il comportamento estende la classe OneShotBehaviour.

Parametri di LightSellerBehaviour.

- *@param a* – Agente proprietario del comportamento
- *@param goods* – Beni di interesse da vendere
- *@param prices* – Prezzi di base, di partenza per i beni in possesso
- *@param estimatedPrices* – Prezzi di vendita stimati per i beni in possesso

Struttura di LightSellerBehaviour.

Il comportamento prototipo è gestito al suo interno da un ciclo while che termina quando tutti i beni di interesse sono stati venduti. Il ciclo while è a sua volta strutturato come una macchina a stati finiti, i cui stati sono scanditi dalla variabile step e dalla clausola switch(step):

- step 0: l'agente analizza l'array dei beni da vendere e decide il bene corrente. Se tutti i beni sono venduti va allo step 4, ovvero esce dal ciclo e termina la sua esecuzione, altrimenti passa allo step 1;
- step 1: l'agente ricerca in rete i mediatori disponibili. Se non trova mediatori disponibili rimane nel ciclo altrimenti passa allo step 2;
- step 2: l'agente invia ai mediatori disponibili la proposta di negoziazione per il bene corrente e attende risposta. Alla prima conferma ricevuta invia il messaggio con le proprie intenzioni e tutti i dati che servono al mediatore per la gestione della vendita, e passa allo step 3; altrimenti dopo un periodo di richieste inoltrate ritorna allo step 0 per scegliere un altro bene da vendere;
- step 3: Passo che gestisce la risoluzione della negoziazione. L'agente riceve un messaggio dal mediatore contenente il vincitore e il prezzo di vendita. Vengono poi effettuate le eventuali computazioni per aggiornare le variabili di stato dell'agente. Si ritorna poi allo step 0 per la vendita di un nuovo bene;

Sintassi dei principali messaggi accettati.

- Messaggio di apertura di negoziazione ricevuto dal mediatore con la conferma della compravendita

```
(ACLMessage.REQUEST:
:sender Auctioneer
```

```
:receiver Seller Agent[i]
:content CONFIRM_SALE
:conversation id "OPEN_NEGOTIATION"
)
```

- Messaggio accettato per la comunicazione del vincitore dell'asta e del prezzo effettivo di vendita

```
(ACLMessage.INFORM:
:sender Auctioneer
:receiver Seller Agent[i]
:content winner rise + winner agent
:conversation id "CLOSE_NEGOTIATION"
)
```

5.2.5 Locator Agent

La gestione della migrazione del Locator può essere opzionalmente gestita dal metodo `handleFailure()` all'interno del behaviour principale `ProvideLocationBehaviour`. A seconda della rete e delle politiche di gestione adottate, l'agente può migrare nel momento in cui il nodo riscontra un 'errore' oppure decidere di attivare un nuovo localizzatore, posizionato altrove, attraverso uno scambio di messaggi e poi disattivarsi al momento della disconnessione.

Classi e elementi che compongono il prototipo.

- LocatorAgent.java: Classe che crea un'istanza dell'agente mediatore e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- ProvideLocationBehaviour.java: Classe che gestisce la creazione della struttura e gli aggiornamenti. Il comportamento estende la classe TickerBehaviour;
- ServicePosition.java: Classe che gestisce la struttura dati nella quale viene tracciato l'ID del container, l'ID dell'agente e il servizio fornito.

Parametri di ProvideLocationBehaviour.

Il comportamento non ha propri parametri in quanto l'agente interroga periodicamente il Directory Facilitator e l'agente AMS.

Struttura di ProvideLocationBehaviour.

La struttura del prototipo si può definire completa ed è stata fornita volontariamente senza commenti a sezioni di codice all'interno del corpo.

Il comportamento si divide in due sezioni:

1. Creazione di una struttura dati temporanea dall'interrogazione degli agenti DF e AMS;
2. Ricopiatura della struttura temporanea nella struttura finale, oggetto d'interrogazioni da parte dei Router Agents.

La prima parte è quella più complessa e utilizza le seguenti classi:

- `Jade.domain.JADEAgentManagement.QueryPlatformLocationsAction`: rappresenta un'azione che può essere richiesta all'AMS per ottenere la lista dei Container di una piattaforma: una volta ottenuto l'oggetto Result dalla risposta dell'AMS è possibile estrarre da questo un oggetto di tipo List, ovvero una lista degli oggetti Location rappresentanti tutti i container della piattaforma;
- `Jade.domain.JADEAgentManagement.QueryAgentsOnLocation`: rappresenta un'azione che può essere richiesta all'AMS per ottenere la lista di agenti eseguiti in un certo container: Si istanzia un oggetto `QueryAgentsOnLocation` e si utilizza il suo metodo `setLocation()` per impostare il container su cui si desidera effettuare la ricerca. Si ottiene sempre un oggetto di tipo `Result` da cui si estrae la lista di tutti gli AID presenti nel container considerato.

Sintassi dei principali messaggi accettati.

- Messaggio di richiesta provider e container in riferimento ad un servizio passato nel contenuto da parte di un router

```
(ACLMessage.REQUEST:
:sender Router Agent
:receiver Locator Agent
:content service
:conversation id "ROUTING"
)
```

5.2.6 Router Agent

Classi e elementi che compongono il prototipo.

- RouterAgent.java: Classe che crea un'istanza dell'agente tracker e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- RoutingBehaviour.java: Classe che gestisce l'inoltro degli agenti verso dispositivi che forniscono il servizio richiesto. Il comportamento estende la classe CyclicBehaviour.

Parametri di RoutingBehaviour.

Il comportamento non ha propri parametri in quanto l'agente viene contattato da agenti esterni che richiedono un certo servizio, comunicato al router come parametro di ricerca.

Struttura di RoutingBehaviour.

Il comportamento può essere gestito in due modi:

1. il router richiede al Locator Agent la locazione del servizio richiesto che poi comunica all'agente richiedente per l'inoltro;
2. il router esegue la ricerca direttamente interfacciandosi con l'agente DF, per la lista degli agenti, e l'agente AMS, per la lista dei container dove l'agente fornitore è contenuto.

Il comportamento viene gestito come una macchina a stati finiti gestita dalla variabile step e dalla clausola switch(step):

- step 0: Il router riceve messaggi provenienti da agenti esterni per la richiesta di un servizio. Quando riceve un messaggio passa allo step 1;
- step 1: per la prima modalità di implementazione il router invia un messaggio all'agente Locator con in servizio e attende la risposta; nella seconda modalità il router esegue la ricerca direttamente. Si passa allo step 2;
- step 2: il router analizza gli agenti provider e i loro container, risultanti dalla ricerca, e invia la migliore soluzione all'agente richiedente.

Sintassi dei principali messaggi accettati.

- Messaggio ricevuto con la richiesta di inoltro da parte di un agente esterno

```
(ACLMessage.REQUEST:
:sender Incoming Agent[i]
```

```
:receiver Router Agent
:content service
:conversation id "ROUTING"
)
```

- Messaggio ricevuto dal Locator Agent con i provider e i container richiesti dal router stesso in riferimento al servizio, oggetto di ricerca

```
(ACLMessage.REQUEST:
:sender Locator Agent
:receiver Router Agent
:content provider + container
:conversation id "ROUTING"
)
```

5.2.7 Searcher Agent

Classi e elementi che compongono il prototipo.

- SearchingAgent.java: Classe che crea un'istanza dell'agente di ricerca e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- SearchingBehaviour.java: Classe che gestisce la ricerca vera e propria a partire da parole chiavi date in input e da un path iniziale che può essere un uri o il path di una cartella all'interno del dispositivo. Il comportamento estende la classe OneShotBehaviour.

Parametri di SearchingBehaviour.

- *@param a* – Agente proprietario del comportamento
- *@param path* – Cammino di partenza della ricerca (path cartella o uri)
- *@param keywords* – Chiave/i dati come criterio di ricerca

Struttura di SearchingBehaviour.

Il behaviour ha il compito di ricercare dalla fonte di partenza le keyword evidenziate in input attraverso un metodo di ricerca, implementato a discrezione del programmatore. Ad ogni indirizzo esterno trovato l'agente compie una clonazione di se stesso e attiva l'agente alla ricerca partendo dall'indirizzo individuato. Il

numero di livelli e di clonazioni non deve superare un certo threshold dato. Viene poi creata una struttura gerarchica che traccia i dati risultati dalla ricerca.

5.2.8 Tracker Agent

Classi e elementi che compongono il prototipo.

- TrackerAgent.java: Classe che crea un'istanza dell'agente tracker e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- TrackingBehaviour.java: Classe che gestisce il meccanismo di registrazione dei cambiamenti di stato in una struttura opportuna. Il comportamento estende la classe CyclicBehaviour;
- LogStructure.java: Classe che gestisce la struttura dati nella quale viene tracciata la storia dell'agente di riferimento. Se vengono tracciati più agenti dallo stesso Tracker vengono istanziate più strutture, ognuna relativa al proprio agente.

Parametri di TrackingBehaviour.

Il comportamento non ha propri parametri in quanto l'agente viene contattato da agenti che vogliono tracciare il proprio log e che comunicano i dati necessari alle operazioni di aggiornamento.

Struttura di TrackingBehaviour.

Il comportamento prototipo è un comportamento ciclico suddiviso in due sezioni in base allo stato dell'agente, gestito dalla variabile booleana `free`.

- Caso `free = 'true'`: l'agente è libero, ovvero non gestisce alcun log. In questa sezione l'agente attende le richieste di log da parte di altri agenti. Se riceve una richiesta imposta lo stato a false;
- Caso `free = 'false'`: sezione in cui viene gestito il tracciamento del log e le eventuali interrogazioni da parte dell'agente o di altri agenti per sapere stati precisi contenuti nella storia di un agente. Quando un agente termina la sua esecuzione invia al tracker una notifica tale per cui quest'ultimo esegue una cancellazione dei record creati nella stuttura di log.

Sintassi dei principali messaggi accettati.

- Messaggio ricevuto con la richiesta di history tracking da parte di un agente esterno

```
(ACLMessage.REQUEST:
:sender Agent[i]
:receiver Tracker Agent
:content null
:conversation id "HISTORY_TRACK"
)
```

- Messaggio ricevuto con la notifica dei cambiamenti di stato da parte dell'agente di riferimento

```
(ACLMessage.INFORM:
:sender Agent[i]
:receiver tracker Agent
:content agent[i] internal state + environment conditions
:conversation id "LOG"
)
```

- Messaggio ricevuto con la richiesta dell'ultimo tracciamento a causa di un fallimento incorso

```
(ACLMessage.REQUEST:
:sender Agent[i]
:receiver tracker Agent
:content time interval for getting log between it
:conversation id "LOG"
)
```

- Messaggio ricevuto con la notifica di chiusura del tracciamento per quell'agente

```
(ACLMessage.INFORM:
:sender Agent[i]
:receiver tracker Agent
:content null
:conversation id "STOP_TRACK"
)
```

5.2.9 Battery Status Agent

Questa tipologia di agente legge lo stato della batteria del sistema. Si riscontra la notevole importanza di questa funzionalità nei meccanismi di predizione delle disconnessioni utilizzate dagli agenti per gestire i propri comportamenti a riguardo. Questo prototipo è stato creato con l'obiettivo di dare un esempio di gestione delle risorse interne del dispositivo, meccanismo importantissimo all'interno delle reti mobili e dai cui dipende il ciclo di vita degli agenti e del nodo stesso.

Battery Status Agent e MANET.

Un agente di questa tipologia influenza la migrazione o la disattivazione degli agenti nel dispositivo, ma non è soggetto a spostamenti in quanto ogni device ha il proprio agente batteria.

Classi e elementi che compongono il prototipo.

- BatteryStatusAgent.java: Classe che crea un'istanza dell'agente tracker e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- BatteryNoticeBehaviour.java: Classe che gestisce l'invio in broadcast a tutti gli agenti del dispositivo del livello di batteria. Il comportamento estende la classe TickerBehaviour.

Parametri di BatteryNoticeBehaviour.

Il comportamento non ha propri parametri in quanto l'agente contatta direttamente la macchina nel quale è situato per accedere alla variabile di sistema.

Struttura di BatteryNoticeBehaviour.

Il behaviour gestisce la comunicazione a tutti gli agenti registrati del livello della batteria quando questo è inferiore ad una certa percentuale. Gli agenti che riceveranno la comunicazione effettueranno le dovute operazioni in previsione di una disconnessione del dispositivo. Per ogni dispositivo esiste quindi un agente BatteryStatus che provvede alle notifiche.

La lettura della variabile di batteria viene eseguita tramite un comando batch che segue il contenuto del file BattStatt.bat che si preoccupa di riportare in un file di testo la percentuale di batteria letta in quel momento.

Sintassi dei principali messaggi accettati.

- In casi particolari può essere richiesto all'agente batteria di fornirne il livello

```
(ACLMessage.REQUEST:
:sender Agent[i]
:receiver BatteryStatus Agent
:content null
:conversation id "BATTERY_STATUS"
)
```

5.2.10 Generic Service Agent

Il Service Agent è un agente che fornisce un determinato servizio dato in input e può ricercare a sua volta uno o più servizi. Il prototipo è stato creato per dare un modello generico per tutti gli agenti che devono eseguire queste operazioni.

Classi e elementi che compongono il prototipo.

- GenServiceAgent.java: Classe che crea un'istanza dell'agente servizio e avvia i comportamenti dell'agente in modo sequenziale;
- RegisterInDF.java: Classe che gestisce la registrazione del servizio offerto nel Directory Facilitator;
- GenServiceBehaviour.java: Classe che gestisce la fornitura di un proprio servizio agli agenti esterni che lo richiedono e la richiesta di altri servizi, forniti da altri agenti. Il comportamento estende la classe CyclicBehaviour.

Parametri di GenServiceBehaviour.

- @param a – Agente proprietario del comportamento
- @param ownService – Servizio fornito dall'agente proprietario
- @param service – Servizi esterni richiesti

Struttura di GenServiceBehaviour.

Il comportamento è strutturato da operazioni iniziali di selezione del servizio esterno da richiedere e da una struttura a stati finiti gestita dalla variabile step e dalla clausola `switch(step)`. Se i servizi necessari all'agente sono stati tutti completati si passa direttamente allo step 4:

- step 0: l'agente ricerca eventuali router per la ricerca del servizio. Se la ricerca da esito positivo si passa allo step 1, altrimenti si passa allo step 2;

- step 1: l'agente invia richiesta al router e attende risposta del servizio. Quando riceve la risposta migra o meno a seconda del container ritornato e passa allo step 3;
- step 2: l'agente richiede il servizio direttamente al DF e invia il messaggio per la richiesta al provider; In questo step gestisce anche messaggi di richiesta per il proprio servizio ed eventuale rilascio delle risorse possedute che erano già state allocate in richieste precedenti. L'agente passa poi allo step 4;
- step 3: l'agente invia richiesta del servizio al provider ottenuto e passa allo step 4;
- step 4: in questo stato vengono gestiti i messaggi di arrivo per la soddisfazione o meno del servizio richiesto e messaggi per la richiesta del servizio;
- step 5: step di conclusione, il servizio richiesto è stato soddisfatto, si passa allo step 0 per ricercare altri servizi necessari.

Sintassi dei principali messaggi accettati.

- Messaggio ricevuto da un Router Agent con la notifica del provider che offre il servizio richiesto

```
(ACLMessage.REQUEST:
:sender Router Agent
:receiver Agent[i]
:content provider + container
:conversation id "ROUTING"
)
```

- Messaggio ricevuto dal provider del servizio con la conferma o meno della fornitura del servizio richiesto

```
(ACLMessage.REQUEST:
:sender Agent[i]
:receiver Agent[j]
:content yes or no + conditions
:conversation id "ASK_FOR_SERVICE"
)
```

- Messaggio ricevuto da agenti esterni per la richiesta del servizio fornito dall'agente

```
(ACLMessage.REQUEST:
:sender Agent[i]
```

```
:receiver Agent[j]
:content service
:conversation id "SERVICE"
)
```

5.3 Gestione dei pattern architetturali

5.3.1 Approccio bottom-up per agenti prototipo specifici

La maggior parte di agenti prototipo creati deve rispettare uno specifico pattern architetturale, già fornito, che fornisce la struttura ottimizzata per la gestione delle comunicazioni associate al ruolo. Da qui il motivo per cui per i ruoli specificati nel framework è stato utilizzato un approccio bottom-up, ovvero la struttura delle comunicazioni viene già fornita nella parte descrittiva, a seconda dell'agente selezionato. La specifica è data in merito alla struttura fornita ai prototipi.

Agenti Negoziatori.

Gli agenti negoziatori devono rispettare la struttura del Branching Pattern: l'agente centrale riveste il ruolo di mediatore che coordina la compravendita tra venditore e i rispettivi compratori. Di conseguenza per le tipologie di agenti contenute nella sezione *Negotiation Agents* viene data l'immagine del pattern a cui si riferiscono.

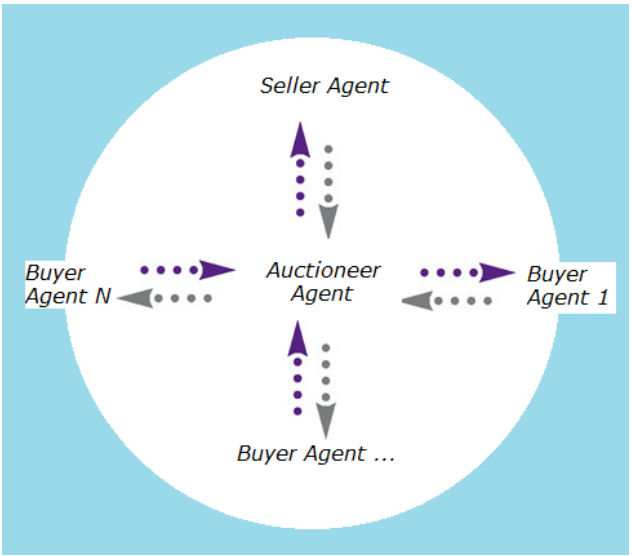


Figura 5.1 – Branching Pattern for Negotiation Scene

Network Agents.

Gli agenti network sono più predisposti per una rete puramente peer-to-peer gestita quindi secondo la struttura di un grafo completo. Se più router sono presenti data la taglia della rete molto elevata si avrà una gestione di più cluster a grafo completo connessi l'un l'altro dagli head cluster, ovvero i Router agent associati a quella subnet.

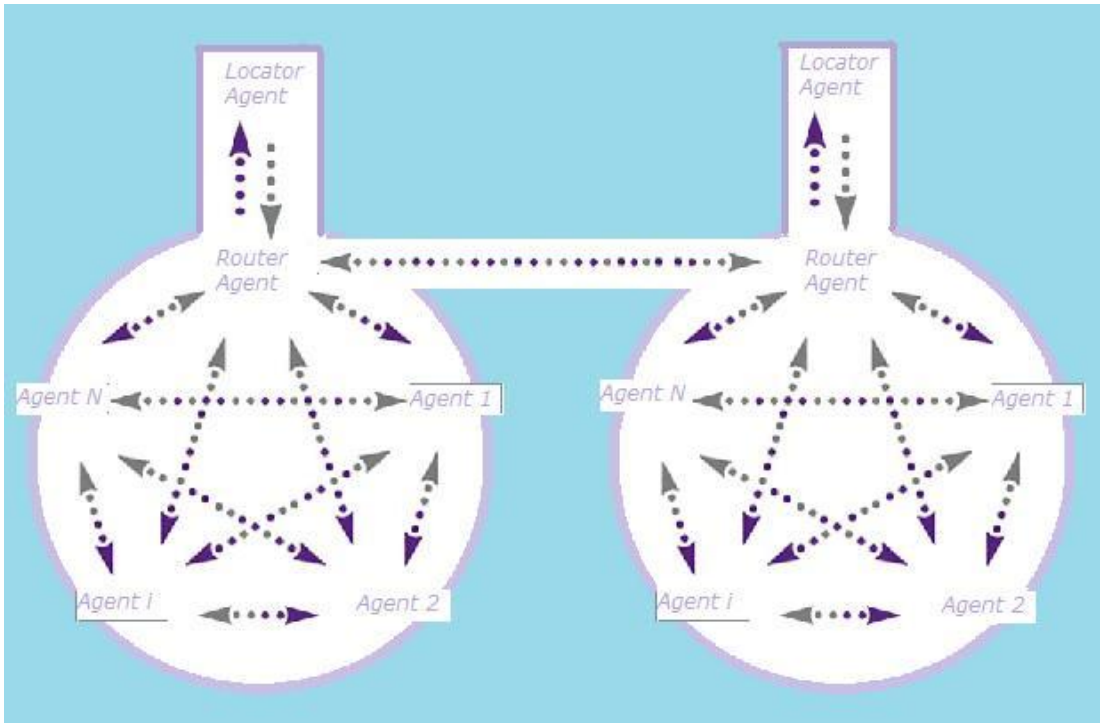


Figura 5.2 – Esempio di architectural pattern per gli agenti di rete

Service Agents.

Gli agenti che forniscono dei servizi generici o agenti di servizio propri del sistema possono adottare qualsiasi tipo di configurazione a seconda delle specifiche adottate dal programmatore. Per questo motivo si rimanda all'approccio top-down descritto nel paragrafo successivo.

5.3.2 Approccio top-down per agenti generici

Gli agenti che forniscono servizi generici o specifici per la gestione delle risorse possono adattarsi a tutti i pattern a seconda delle proprie funzioni specifiche associate, delle politiche di gestione della rete stessa e della struttura delle interazioni. Il sistema interattivo fornisce il supporto per la struttura architetturale attraverso il modulo Pattern Viewer accessibile dal pulsante omonimo nella sezione *Other Systems* (vedi paragrafo 6.3).

6 Implementazione

Questo capitolo presenta il sistema nel dettaglio dal punto di vista dell'implementazione e delle funzionalità fornite per il supporto alla progettazione. Dapprima si presentano le piattaforme utilizzate, successivamente vengono dettagliati i componenti del framework e le funzioni associate.

6.1 Piattaforme utilizzate e librerie

Per l'implementazione del framework è stato utilizzato come ambiente di sviluppo la piattaforma Eclipse integrata con la piattaforma JADE, attraverso l'importazione dei file di estensione .jar appropriati.

6.1.1 JADE

Si presentano in questa sezione alcune notizie più dettagliate e le caratteristiche principali della piattaforma JADE utilizzata come supporto. Per la programmazione si faccia riferimento alle guide scaricabili alla pagina http://jade.tilab.com/.

Il nome della piattaforma JADE deriva dall'acronimo Java Agent DEvelopment framework ed è un programma libero sviluppato e distribuito dalla divisione R&D di Telecom Italia. Attualmente a questa società si sono affiancati nomi noti come Motorola, Whitestein Technologies AG., Profactor GmbH, and France Telecom R&D, proprio a supporto dello sviluppo di questa piattaforma che sembra essere il futuro per quanto riguarda i prossimi software ad agenti.

L'ultima versione rilasciata è la 4.1 del 13 Luglio 2011 ed è quindi molto recente. A dimostrazione del fatto che in questo momento gli agenti mobili stanno riscuotendo molto successo per superare le problematiche portate alla luce dalle tecnologie dei dispositivi attuali.

La piattaforma è un'implementazione Java di specifiche FIPA e precisamente utilizza la tecnica FIPA-compliant. Gli standard agent-based, promossi da questa società, the Foundation of Intelligent Physical Agents, mirano alla comunicazione e all'interoperabilità tra agenti eterogenei, creati da diverse piattaforme, per ottenere una condivisione completa di tutti i servizi che essi offrono. La tecnica sfruttata da JADE è mirata al dialogo tra piattaforme diverse.

I principali componenti che costituiscono la piattaforma nel suo insieme sono:

- Un ambiente di runtime dove gli agenti creati vengono attivati ed eseguiti;
- Una suite di tool a supporto della grafica che permettono di monitorare e di gestire l'attività degli agenti in esecuzione;
- Una libreria di classi che i programmatori possono usare, con la possibilità di estenderle, per sviluppare gli agenti voluti;
- Un set di add-on che aumentano prestazioni e funzioni della piattaforma e che permettono l'installazione della suite nei dispositivi portatili.

Quando il framework JADE viene attivato vengono creati immediatamente:

- L' RMA agent. Un agente che implementa e attiva la piattaforma JADE vera e propria;
- Il main-container. Container principale di esecuzione dove vengono abilitati gli agenti elencati di seguito;
- Il DF agent (Directory Facilitator). È un agente che offre la possibilità agli altri agenti di registrare il servizio fornito. In questo modo un agente che necessita di un servizio non posseduto può interrogare il DF per ottenere una lista di provider. Il DF è in realtà n servizio opzionale: se da una parte può esistere una piattaforma senza l'agente, dall'altra, in una piattaforma possiamo trovare anche più agenti di questo tipo;
- L' AMS agent (Agent Management Service). È un agente che gestisce le operazioni all'interno delle diverse piattaforme, come ad esempio la creazione e l'eliminazione di un agente. Mantiene al suo interno una lista di containers (dispositivi) presenti per piattaforma e rispettivamente la lista di agenti appartenenti.

JADE supporta inoltre la fase di deployment. Il deployment, inteso come rilascio o distribuzione, è l'applicazione della soluzione sviluppata al problema reale in un dato dominio. Consiste nell'avviare il sistema ad agenti, tipicamente su una rete di dispositivi, e quindi di collaudarlo, effettuarne la manutenzione e/o estenderne le funzionalità. La piattaforma è in grado inoltre di monitorare lo stato di tutti gli agenti del sistema, anche di quelli in esecuzione su computer remoti, mentre, ad esempio FIPA-OS si limita a quelli eseguiti nella stessa Virtual Machine. JADE possiede infatti lo strumento Sniffer che rende molto semplice seguire le conversazioni che avvengono tra gli agenti, tramite l'uso di sequence diagrams tracciati graficamente dall'agente stesso.

Il processo tramite cui JADE crea una rete con tecnologia ad agenti mobili è un processo che sfrutta l'infrastruttura della rete fissa per supportare i servizi da fornire alla rete mobile. Questa tipologia ibrida di rete viene definita Grid Network, ne verranno date alcune specifiche nel capitolo conclusivo.

La creazione della rete inizia dall'attivazione di una piattaforma JADE all'interno di un dispositivo fisso. Quando un terminale mobile si collega alla rete viene creato un nuovo container, situato al suo interno. Il container si registra nella piattaforma attivata in precedenza e subisce poi, un processo di suddivisione, definito processo di split, in due sezioni: un container back-end e uno front-end. Il primo verrà gestito da un server situato nella rete cablata, attraverso un mediatore fornito dalla piattaforma stessa, il secondo verrà invece attivato nel dispositivo mobile e conterrà le APIs di JADE per la creazione e l'esecuzione degli agenti nel dispositivo.

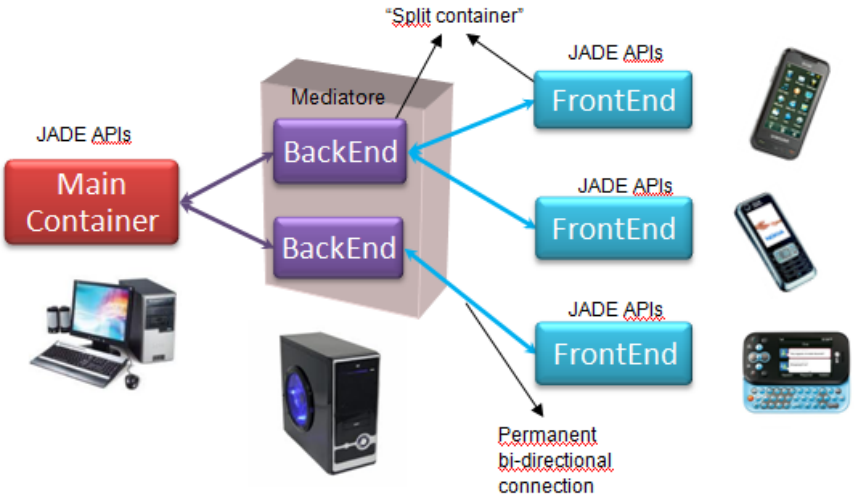


Figura 6.1 - Struttura dei containers in Jade nei dispositivi mobili

Creare un JADE agent richiede la creazione di una classe che estende la classe principale jade.core.Agent. Di seguito è riportato un semplice esempio di creazione di un agente che stampa il proprio nome:

```
(anAgent.java):
import jade.core.Agent;
public class anAgent extends Agent {
   protected void setup() {
      System.out.println(getLocalName()+ " is running!");
   }
}
```

Il metodo `setup()` è il metodo principale di un agente in cui vengono implementate tutte le operazioni che l'agente dovrà eseguire alla ricezione o all'invio di un messaggio. La differenza principale con la programmazioni ad oggetti risiede proprio nella gestione dei metodi che svolgono gli agenti. Questi non sono implementati attraverso metodi tradizionali ma attraverso la creazione di nuove istanze di oggetti definiti behaviour.

Gestione dei behaviour.

JADE gestisce le funzioni di un agente raggruppandole in classi definite behaviour.

Questi vengono eseguiti secondo una successione stabilita dall'agente stesso che tramite un behaviour padre schedula dei sub-behaviour funzionali. I comportamenti vengono usati per definire i vari ruoli di un agente.

JADE fornisce la classe jade.core.behaviours.Behaviour, classe principale che definisce come vengono gestiti i comportamenti degli agenti. Lo sviluppatore definisce il comportamento estendendo la classe Behaviour.

JADE fornisce anche delle sottoclassi che definiscono i comportamenti standard:

- SimpleBehaviour: behaviour base che implementa i metodi `action()` and `done();`
- OneShotBehaviour: behaviour eseguito una volta soltanto all'attivazione dell'agente proprietario. Il metodo `done()` ritorna sempre `true;`
- CyclicBehaviour: behaviour che viene eseguito cicliclamente e non finisce mai. Il metodo `done()` ritorna sempre `false;`
- TickerBehaviour: behaviour che viene eseguito periodicamente. Il periodo può essere dato in input dal programmatore. Il programmatore definisce il metodo `onTick()`, il metodo `action()` è già implementato e definisce l'esecuzione periodica di `onTick();`
- WakerBehaviour: behaviour che viene eseguito solo una volta dopo che un certo periodo è trascorso. Il metodo `action()` è già implementato e definisce l'esecuzione di `onWake()` che deve essere esplicitamente implementato dallo sviluppatore.

Inoltre definisce alcuni behavior composti che definiscono la successione temporale dei sub-behaviour schedulati al loro interno:

- ParallelBehaviour: prevede uno scheduling parallel dei sub-behaviour;
- SequentialBehaviour: prevede uno scheduling sequenziale;
- FSMBehaviour: scheduling per macchina a stati finiti dove ogni sub-behaviour rappresenta uno step della macchina.

Un behaviour gestisce l'invio e la ricezione dei messaggi e determina le operazioni da eseguire a seconda dello stato in cui si trova. Queste operazioni possono essere listate direttamente nel corpo del metodo `setup()` o possono essere

implementate in metodi privati della classe dell'agente che vengono poi richiamati nei punti opportuni. Questa seconda opzione aggiunge ulteriormente efficacia all'obiettivo della programmazione ad agenti che vuole ottenere una particolare predisposizione alla dinamicità di acquisizione di metodi e behaviour da parte di più agenti.

Nella realizzazione dei prototipi, all'interno del metodo `setup()`, si è utilizzata principalmente una struttura a step, simile ad una macchina a stati finiti. Grazie alla suddivisione del listato si è potuto trattare in maniera chiara e ordinata ogni singola condizione in cui l'agente può ritrovarsi. Attualmente i prototipi sono strutture semplici, corredate da poche righe di codice per step. Una macchina a stati finiti può essere il passo successivo di uno sviluppo più complesso dei prototipi stessi. Può essere, infatti, utilizzata quando i vari step hanno raggiunto una complessità tale da richiedere una semplificazione. Questa tipologia di infrastruttura permette di strutturare i singoli passi come behaviour indipendenti, che vengono richiamati man mano si esaurisce la condizione corrente e se ne presenta una nuova. può essere un'ottima soluzione, altamente dinamica, per l'acquisizione degli stessi da parte di agenti diversi.

Gli elementi principali della classe Behaviour, utilizzati nella composizione dei prototipi sono:

- Il metodo `block()`: metodo che sospende l'esecuzione dell'agente fino all'arrivo di un nuovo messaggio, momento in cui ritorna allo stato attivo.
- Il metodo `block(long millis)` : questo metodo esegue la sospensione del behavior in esecuzione per la durata del periodo dato in input;
- La variabile `myAgent`: variabile che si riferisce all'agente proprietario del comportamento. Il suo utilizzo premette la dinamicità dei behaviours nell'essere associati a più agenti diversi. Infatti, richiama sempre l'agente a cui il comportamento è associato nel metodo `setup()`.

La particolarità dei comportamenti è legata al fatto che non implementano metodi per gestire i messaggi, e devono quindi accedere alla classe `Agent` che fornisce dei metodi opportuni, ad esempio per l'invio si utilizza il metodo `send()`, mentre per la ricezione si utilizza il metodo `receive()`. La ricezione può essere differenziata in asincrona o sincrona, a seconda della sincronizzazione o meno dei messaggi inviati e delle risposte ricevute in merito; vengono quindi utilizzati rispettivamente il metodo `receive()` e `blockingReceive()`. Il secondo metodo è tuttavia pericoloso, perché sospende tutte le attività dell'agente, compreso il behaviour corrente che si attiva nuovamente alla ricezione di quel particolare messaggio.

La coda dei messaggi in arrivo di un agente è unica, non esiste un meccanismo di dispatching dinamico tra i behaviour, e per decidere quali messaggi ricevere è possibile specificare un `MessagePattern` che definisce un modello di messaggio che viene accettato in quel punto dell'esecuzione del codice. La comunicazione inoltre può essere punto-punto o multicast a seconda della numerosità di destinatari dei messaggi inviati.

Metodologie di implementazione.

Un behaviour può essere implementato in una classe interna dell'agente stesso, ad esempio:

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
public class HelloWorldAgent extends Agent {
   protected void setup() {
      addBehaviour(new OneShotBehaviour() {
         public void action(){
          System.out.println(getLocalName()+ "I srunning!");
         }
      });
   }
}
```

Un secondo metodo di implementazione è creare una classe esplicita del comportamento tale che anche altri agenti possano utilizzare quel behaviour.

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
public class HelloWorldAgent extends Agent {
   protected void setup() {
      addBehaviour(new HelloWorldBehaviour(agent));
   }
}

class HelloWorldBehaviour extends OneShotBehaviour{
   HelloWorldBehaviour(Agent agent){
   }
   public void action(){
     System.out.println(myAgent.getLocalName()
                                        + " is running!");
   }
}
```

Questa seconda modalità di implementazione permette un'efficiente programmazione ad agenti nelle reti mobili. L'obiettivo perseguito, infatti, è rendere il codice il più modulare possibile. Per modulare s'intende implementazione di ogni ruolo o addirittura di ogni metodo, avente complessità elevata, in un singolo behaviour che può operare indipendentemente dagli altri metodi associati all'agente stesso. In questo modo più agenti possono assumere lo stesso comportamento senza il bisogno di implementare nuovamente il metodo al loro interno.

Gestione della mobilità.

La mobilità di un JADE Agent avviene invocando il metodo `doMove(Location container)`, ovvero un metodo che riceve in input un oggetto di tipo `Location` che identifica sia l'indirizzo del nodo di destinazione sia la piattaforma di destinazione. Una volta che l'agente è giunto nell'agency destinataria, riprende la sua esecuzione invocando il metodo `afterMove()` in cui vengono implementati i passaggi che l'agente deve effettuare per riprendere l'esecuzione corretta e ottenere i risultati voluti. Ad esempio, nei behaviour strutturati per step, il metodo riporta lo step da cui deve riprendere la sua esecuzione.

Per gestire la trasmissione degli agenti in altri container da quelli di appartenenza vengono quindi utilizzati i metodi:

- `beforeMove()`: metodo facoltativo, chiamato prima della migrazione, per gestire alcune funzioni in previsione dello spostamento;
- `doMove(Location container)`: metodo obbligatorio, che gestisce la serializzazione dei dati da trasmettere e sposta fisicamente l'agente dal container di appartenenza al container di destinazione;
- `afterMove()`: metodo richiamato dopo la migrazione per stabilire le funzioni da eseguire post-migrazione. JADE infatti implementa la tipologia di migrazione debole con invocazione di metodo fisso.

```
import jade.core.ContainerID;
import jade.core.Location;
…
   Location loc = here();
   if (!loc.getName().equals("Main-Container")){
      doMove( new ContainerID("Main-Container", null) );
   } …
protected void beforeMove() {}
protected void afterMove() {}
```

Tutti i prototipi realizzati gestiscono la caratteristica della mobilità, all'interno del behaviour principale, come ad esempio nella ricerca di un servizio voluto e nella gestione dei fallimenti all'interno metodo `handleFailure()`.

La migrazione viene introdotta nei prototipi in due punti:

1. la ricerca di un servizio;
2. la gestione dei fallimenti.

La mobilità nella ricerca di un servizio si attiva nel momento in cui l'agente riceve risposta dall'agente router. Il messaggio contiene, infatti, il container dove migrare e il provider del servizio richiesto. L'agente verifica se il container ricevuto è lo stesso in cui è situato o si tratta di una diversa locazione. Se si verifica questa seconda possibilità l'agente esegue il metodo `doMove(container)` e, una volta raggiunta la piattaforma destinataria, esegue il metodo `afterMove()`. Questo metodo andrà a richiamare un determinato step all'interno del behaviour in esecuzione che in questo caso gestirà la richiesta diretta al provider del servizio.

Il router può inoltre inviare più oggetti `Location` con i relativi agenti fornitori. In questo caso l'agente richiedente dopo aver ricevuto risposta negativa da un provider potrà procedere al container successivo nella lista. Questa metodologia di gestione è un caso speciale di *Itinerary Pattern* di cui si è discusso al paragrafo 4.3.3.

Per la mobilità nella gestione dei fallimenti si rimanda al paragrafo successivo.

Gestione dei fallimenti.

Nelle reti MANET ogni agente ha un proprio metodo di gestione dei fallimenti a seconda delle politiche adottate dall'intero sistema.

Per questo motivo i prototipi creati sono stati predisposti per ricevere messaggi di predizione dei fallimenti, la cui gestione è rimandata al metodo privato `handleFailure()`. Ogni agente, a seconda del proprio ruolo all'interno del sistema, gestirà in questo metodo le opportune soluzioni a seconda del fallimento riscontrato.

Per uniformare le politiche adottate dall'intera rete, può essere istanziato un behaviour specifico che venga acquisito dinamicamente dagli agenti al momento opportuno. Adottando questa soluzione si continua a perseguire l'obiettivo di una programmazione modulare che faciliterà nel futuro la scalabilità di queste reti. Inoltre un behaviour generico può essere poi specializzato per ogni agente mantenendo in egual modo struttura e regole di gestione adottate.

La sintassi dei messaggi gestiti nei prototipi è la seguente:

- Messaggio di notifica di fallimento

```
(ACLMessage.INFORM:
:sender System Agent/Resource Management Agent
:receiver Agent[j]
```

```
:content Failure type + necessary data according to type
of failure
:conversation id "FAILURE"
)
```

- Messaggio di notifica di predizione di una disconnessione a causa dell'esaurimento della batteria

```
(ACLMessage.INFORM:
:sender BatteryStatus Agent
:receiver Agents
:content level of battery
:conversation id "BATTERY_NOTICE"
)
```

Di seguito il codice inserito nei prototipi relativo alla sintassi dei messaggi sopra specificata:

```
MessageTemplate node_failure = MessageTemplate.and(
       MessageTemplate.MatchPerformative(ACLMessage.INFORM),
       MessageTemplate.MatchConversationId(FAILURE));

ACLMessage switch_off = myAgent.receive(node_failure);
if (switch_off != null) {
   handleFailure();
} else {
   block();
}
```

L'esempio più specifico di gestione è dato dalla ricezione del livello di batteria inviato dal BatteryStatus Agent che manda periodicamente dei messaggi agli agenti per segnalare la prossimità di un distacco dalla rete del nodo.

Di seguito viene riportato un esempio di codice per la ricezione del messaggio:

```
MessageTemplate battery = MessageTemplate.and(
       MessageTemplate.MatchPerformative(ACLMessage.INFORM),
       MessageTemplate.MatchConversationId(BATTERY_NOTICE));
ACLMessage batt = myAgent.receive(battery);
   if (batt != null) {
      if(batt.getContent().equals(BATTERY_DEAD)){
         handleFailure();
```

```
        }
       } else {
        block();
      }
```

A seconda della politica di gestione dei fallimenti e della tipologia di agente viene gestita la possibilità di migrare verso altri nodi ancora attivi. La locazione di destinazione, in questo caso, è il risultato di un'analisi delle prestazioni degli altri dispositivi e della posizione in cui essi si trovano. Una volta determinata l'agency di arrivo, l'agente eseguirà il metodo `doMove(container)`, specificato al paragrafo precedente, dove riprenderà la sua esecuzione.

6.1.2 Librerie aggiunte

Nella cartella di progetto è stata creata una cartella lib contenente tutte le librerie utilizzate per l'implementazione del sistema interattivo.

Per l'integrazione con la piattaforma JADE e il supporto ad agenti sono state utilizzate le librerie `jade.jar` e `commons-codec-1.3.jar` dell'ultima versione 4.1 della piattaforma ad agenti.

Per l'integrazione con l'editor java all'interno del framework è stata utilizzata la libreria `jsyntaxpane-0.9.5-b29.jar` che definisce il riconoscimento del codice java adatto per la compilazione delle classi.

6.2 Struttura del Framework

Il framework è composto da cinque sezioni:

1. il menù laterale;
2. il menù superiore;
3. l'activity log;
4. il content pane;
5. il code pane;
6. il compiler pane.

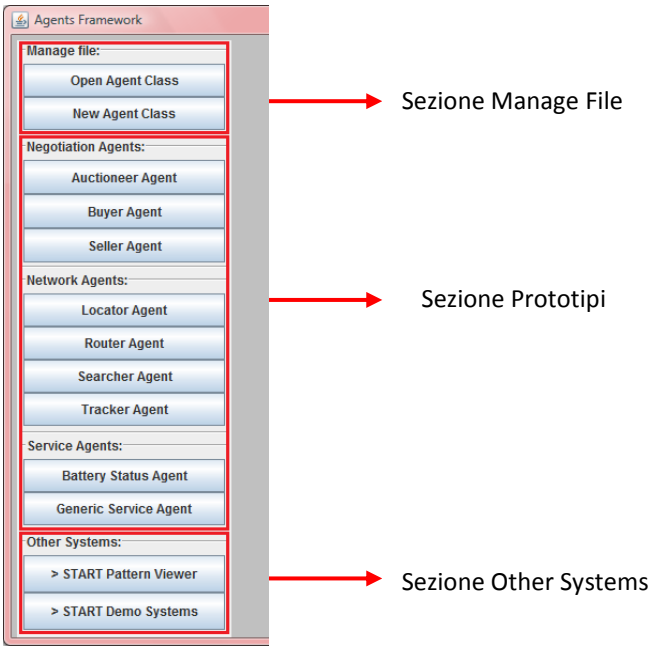
Figura 6.2 – Agent Framework: schermata principale

6.2.1 Menù Laterale

Il menù laterale è sempre visibile in ogni sezione del framework. Contiene le principali funzioni di gestione per:

1. La gestione dei file già creati nell'editor;
2. La selezione dei prototipi da visualizzare e modificare;
3. L'apertura del modulo Pattern Viewer per la gestione dei pattern architetturali,
4. L'apertura del modulo Demo Systems per l'esecuzione dei sistemi demo forniti a supporto del progettista.

Figura 6.3 – Menù laterale



Sezione Manage File.

A questa prima sezione è associata la funzione di gestione file già creati dallo sviluppatore. Il tasto 'Open File' rimanda direttamente alla cartella di progetto, \editing_file, dedicata a contenere i file .java per la loro visualizzazione nel code pane. I file modificati sovrascrivono i file precedenti, in quanto il nome del file riprende il nome della classe Agent specificata al loro interno.

La seconda funzione di questa sezione è la creazione del prototipo di un agente generico attraverso il tasto 'Create new agent'. Del prototipo viene fornito solo lo scheletro iniziale di un agente che non ha ruoli associati. La funzione è stata volutamente inserita tra le opzioni di gestione in quanto i prototipi forniti attualmente costituiscono solo un piccolo insieme di tutte le tipologie di agenti che si possono implementare, e che potrebbero alimentare il framework in futuro.

Sezione Prototipi.

La sezione dedicata ai prototipi forniti è suddivisa a sua volta in tre sezioni, in base ad una distinzione ponderata sui ruoli associati ad ognuno.

Le categorie di ruolo sulle quali si è basata la suddivisione sono:

1. Negotiation Agents;
2. Network Agents;
3. Service Agents.

Appartengono alla categoria Negotiation Agents gli agenti Auctioneer, Buyer e Seller, i quali ammettono politiche di gestione delle negoziazioni. Segue la categoria Network Agents dedicata agli agenti che hanno funzioni relative alla gestione della rete e funzioni assimilabili di localizzazione dei dispositivi e inoltro degli agenti. Appartengono a questa categoria gli agenti Router, Locator e Tracker. Per ultima una categoria generica di agenti di servizio definiti Generic Service Agents che sono strutturati attraverso un unico behaviour che gestisce la richiesta di uno o più servizi e la fornitura di un proprio servizio, parametri che verranno dati input dallo sviluppatore del sistema. Al programmatore è, infatti, affidato il compito di specializzare tali agenti. È stato aggiunto in questa sezione un esempio di Resource Management Agent, ovvero il BatteryStatus Agent, che ha il ruolo preciso di monitorare il livello di batteria del dispositivo e comunicarlo agli altri agenti appartenenti al nodo.

Sezione Other Systems.

A questa sezione è associata l'apertura di due moduli indipendenti dal framework ai quali sono dedicate rispettivamente nell'ordine: la strutturazione architetturale del sistema, di cui si rimanda al paragrafo 6.3, e la dimostrazione attraverso due sistemi multi agente demo, di cui se ne descrivono le specifiche al paragrafo 6.4.

6.2.2 Menù Superiore

Il menù superiore è dedicato alla gestione effettiva dei prototipi selezionati, in particolare del loro aggiornamento. Alla selezione di un agente appare il menù contenente un campo *nome* editabile, e i pulsanti di gestione del file associato a quel particolare agente. Da questo menù dipartono le funzioni principali di creazione, compilazione e salvataggio delle classi.

Figura 6.4 – Menù superiore

Creazione degli agenti.

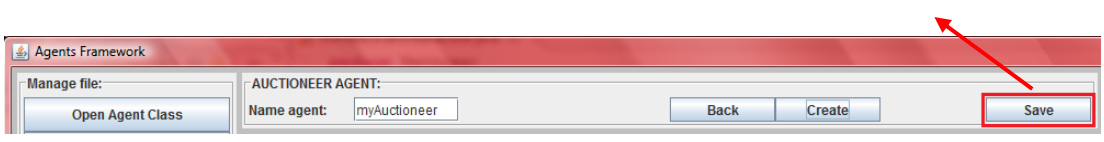
La creazione della classe dell'agente selezionato avviene secondo i seguenti passi:

1. Editare il nome dell'agente nel campo *Name Agent* al quale è già associato un nome di default. Questo permette la corretta inizializzazione della classe con il nome acquisito e il salvataggio del file già nominato correttamente.
2. Cliccare il pulsante 'Create' che andrà a visualizzare il code pane con il codice appartenente al prototipo selezionato.

Compilazione e Salvataggio.

Vista la consueta gestione degli ambienti di sviluppo più diffusi di associare alla compilazione della classe il salvataggio della stessa, queste funzioni sono state uniformate nel tasto 'Save'. Cliccando il pulsante, viene richiesto il salvataggio del file name_agent.java all'interno della cartella dedicata, successivamente nel compiler pane vengono visualizzati gli eventuali errori di compilazione o un messaggio di buona riuscita.

Il tasto di salvataggio viene visualizzato dopo aver creato l'agente.



Di seguito il codice associato al salvataggio vero e proprio:

```
…
JButton savedag_exec_p = new JButton("Save");
…
JFileChooser fChooser = new JFileChooser(new
File(".\\editing_file\\"));
FileFilter type1 = new ExtensionFilter("Java source", ".java");
…
FileOutputStream fos = new FileOutputStream(f.toString()+ ".java");
PrintStream fwriter = new PrintStream(fos);
fwriter.println(pan_code.getText());
fos.close();
…
```

Di seguito il codice associato al tasto per la compilazione:

```
…
String path = new java.io.File("").getAbsolutePath() + "\\lib\\";
…
FileOutputStream fout = new FileOutputStream("compile.bat");
PrintStream fwriter = new PrintStream(fout);
fwriter.println("cd " + f.getParent());
fwriter.println("del result_*.txt");
String path = new java.io.File("").getAbsolutePath() + "\\lib\\";
fwriter.println("javac -classpath " + path + "jade.jar;" + path +
"commons-codec-1.3.jar " + "-Xstdout "+ f.getParent()+"\\result_" +
f.getName() + ".txt "+ f.getParent()+"\\" + f.getName() + ".java");
fout.close();
…
eseguiBat("compile.bat");
…
```

Il metodo `getAbsolutePath()` applicato ad un oggetto di tipo io.File ritorna in output il percorso della cartella di progetto. La gestione del path è una gestione dinamica che aggiorna il percorso a seconda di dove avviene lo store dei file di progetto. Non causa quindi problemi di integrità nei diversi ambienti che possono essere utilizzati dai progettisti.

6.2.3 Activity Log

Il pannello dedicato al log delle attività svolte dallo sviluppatore traccia le operazioni di creazione e di modifica dei file.

Per la creazione di un agente ex-novo, una volta salvato il file, viene visualizzato:

- La tipologia dell'agente, ovvero la classe di partenza per la creazione;
- Il nome editato nel menù superiore associato al file;
- L'ora di creazione.

La modifica di un file precedentemente creato viene indicata attraverso la label ‘modified’ seguita dal nome dell’agente aggiornato e dall’ora di creazione.

Di seguito il codice che aggiunge una riga all’activity log:

```
…
String timeClock = "'</p><p>["+ore+":"+min+":"+sec+"]</p>";
…
DataClass.logs += agent_type + newag_name_t.getText() + timeClock;
logAgents.setText(DataClass.logs);
…
```

Figura 6.5 – Activity Log

6.2.4 Content Pane

Il pannello del contenuto visualizza le informazioni relative alla selezione effettuata. I contenuti associati sono stati strutturati in modo da visualizzare il nome del prototipo o del pattern selezionato, la descrizione e le eventuali specifiche aggiunte. L’obiettivo è quello di fornire un supporto informativo al programmatore che presenta una panoramica dell’agente che andrà a creare.

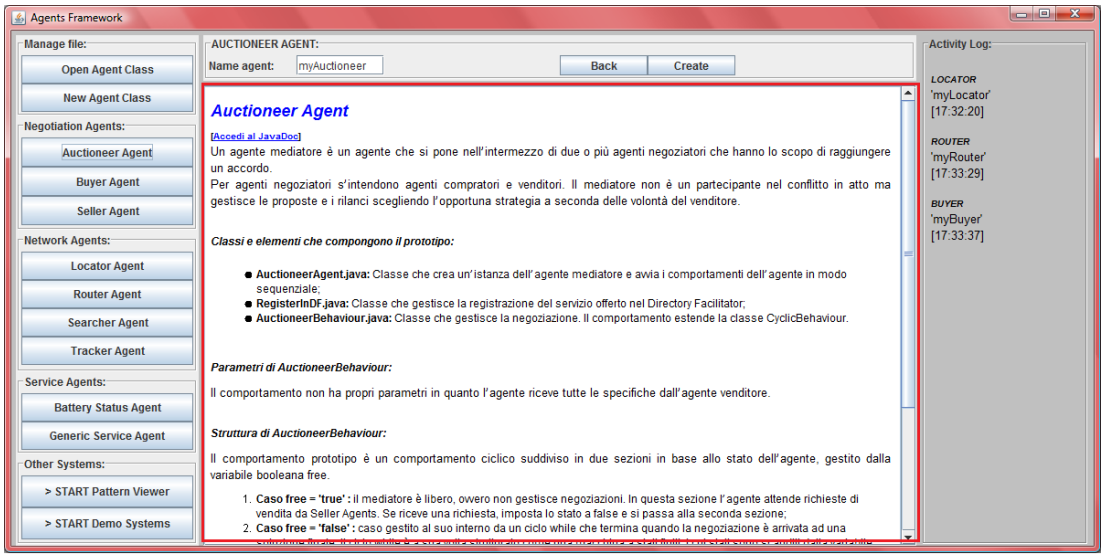


Figura 6.6 – Content Pane

6.2.5 Code Pane

Questo pannello è dedicato alla visualizzazione del codice delle classi. Viene visualizzato quando viene creato un agente per la prima volta e quando viene riaperto un file preesistente. Il pannello permette la corretta identificazione del codice java con l'evidenziazione delle parole chiave del linguaggio. Per ottenere il compilatore Java l'implementazione del pannello vede l'utilizzo della `jsyntaxpane-0.9.5-b29.jar`.

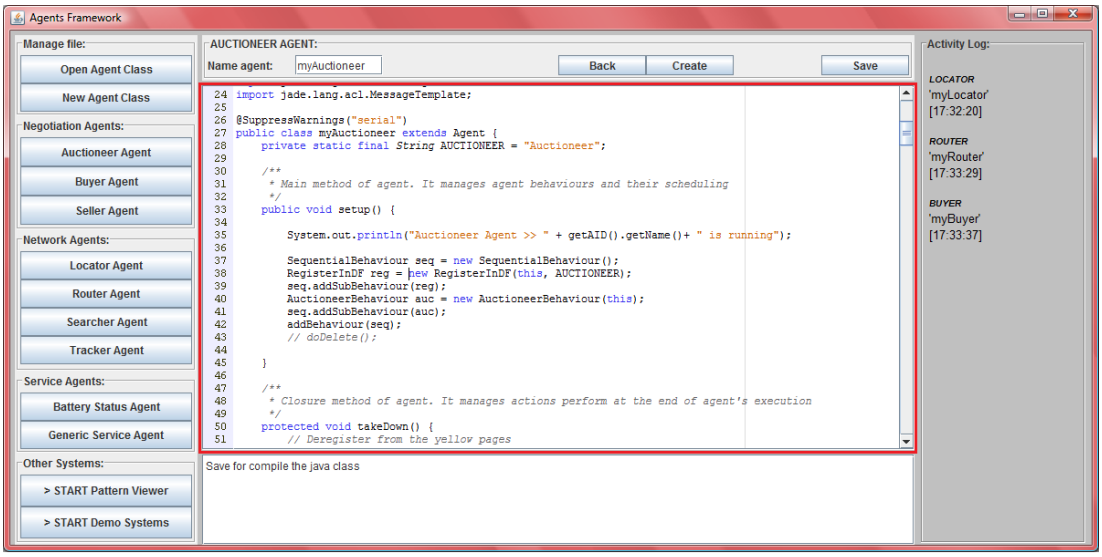


Figura 6.7 – Code Pane

6.2.6 Compiler Pane

Il Compiler Pane è la sezione riservata agli errori di compilazione che appare al di sotto del code pane. La compilazione è stata implementata attraverso la creazione dinamica di un file batch che riporta il comando javac seguito dal classpath relativo alle librerie di JADE e al nome del file da compilare.

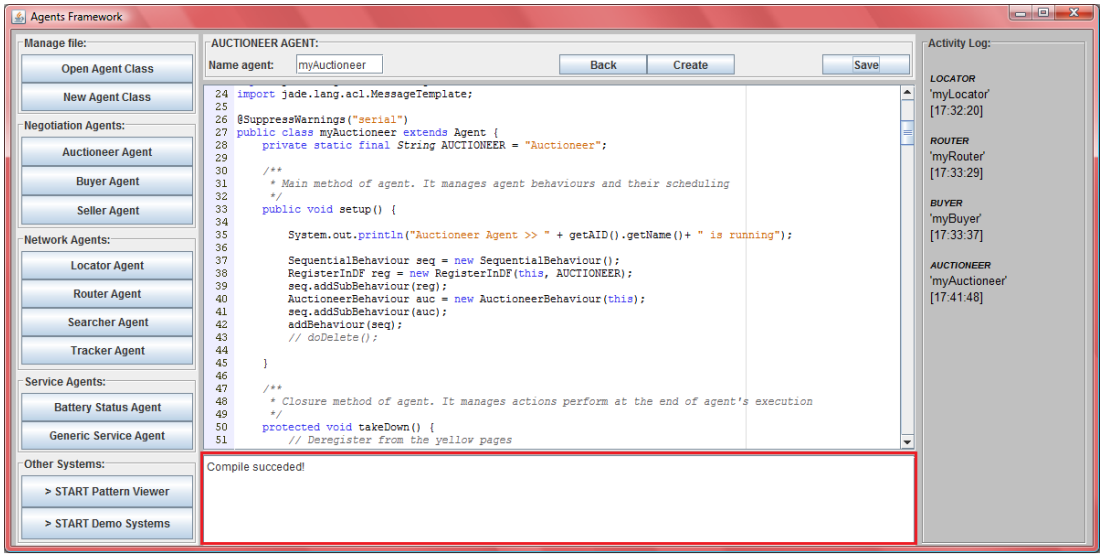


Figura 6.8 – Compiler Pane

6.3 Struttura del modulo 'Pattern Viewer'

Il sistema interattivo fornisce il supporto per la struttura architetturale attraverso il modulo Pattern Viewer accessibile dal pulsante omonimo nella sezione *Other Systems*.

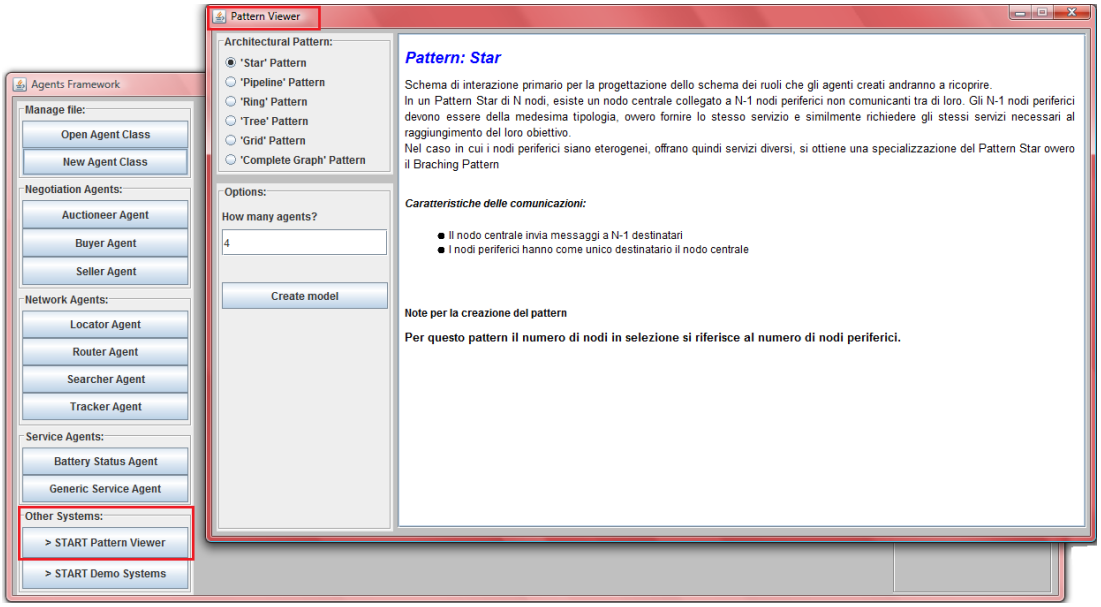


Figura 6.9 – Avvio Pattern Viewer

Dal menù laterale del modulo è possibile selezionare la tipologia di pattern desiderata e nella sezione *Options* è possibile inserire il numero indicativo di agenti del sistema che si vuole realizzare. Attualmente, per la maggior parte di pattern viene dato un range limitato per il numero da inserire che va in media da 3 agenti come limite inferiore ad un massimo di 8 o 9 agenti.

Alla selezione di un pattern viene data una breve descrizione con qualche dato sul numero di connessioni che si possono instaurare tra gli agenti. Il modulo può rimanere in esecuzione durante la fase di progettazione e programmazione per ricordare allo sviluppare la struttura da ottenere.

6.3.1 Principali Funzionalità

Creazione del Pattern di supporto.

Il processo per la creazione di un modello da seguire durante la programmazione è il seguente:

1. Selezionare il pattern desiderato
2. Inserire nella sezione *Options* il numero di agenti da inserire

3. Cliccare il pulsante ‘Create model’
4. Editare i nomi degli agenti che appaiono nella struttura. Al programmatore la scelta di editare il ruolo associato, il nickname dell’agente o le informazioni a lui necessarie per lo sviluppo del proprio sistema.

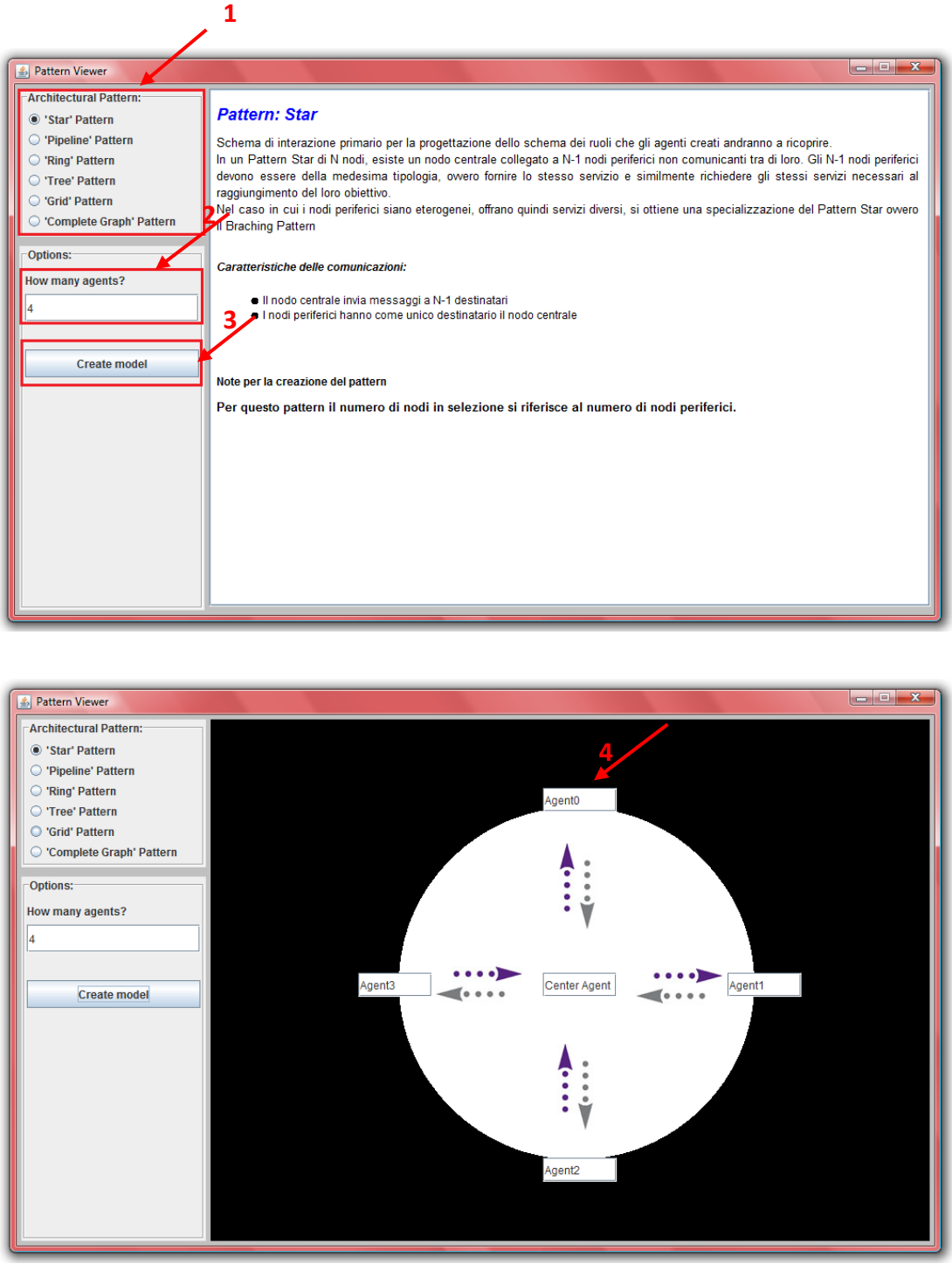


Figura 6.10 – Steps della creazione del pattern

Di seguito viene riportato un esempio di codice per la creazione dinamica del pattern:

```
…
if (pStar.isSelected()) {
    patternImageName = PatternDataClass.getStarImage(nAgs);
    int[][] position = PatternDataClass.getStarPoints(nAgs);
    int i = 0;
    for (i=0; i<nAgs; i++) {
        agents[i].setText("Agent"+i);
        agents[i].setBounds(position[i][0],position[i][1], 80, 26);
        dynScreen.add(agents[i]);
    }
    agents[i].setText("Center Agent");
    agents[i].setBounds(360, 274, 80, 26);
    dynScreen.add(agents[i]);
}
…
```

Metodo per la stampa di un pattern:

```
public void paint(Graphics g){
    super.paint(g);
    ImageIcon imgPath = new ImageIcon(patternImageName);
    Image img = imgPath.getImage();
    g.drawImage(img, 407, 92, this);
}
```

6.4 Struttura del modulo 'Demo Systems'

La sezione riservata ai sistemi demo permette di eseguire due sistemi già istanziati e di osservare le comunicazioni tra gli agenti, tramite il supporto della piattaforma JADE e in particolare dell'agente Sniffer che verrà attivato. I sistemi forniti hanno l'obiettivo di mostrare come l'ambiente creato per la fase di progettazione iniziale di un sistema ad agenti e un ambiente di deployment come quello fornito dalla piattaforma JADE possono integrarsi e collaborare per fornire uno strumento completo allo sviluppatore.

Dalla sezione *Other Systems* è possibile, quindi, accedere al modulo, anch'esso composto da più sezioni, di seguito specificate:

- un menù laterale,
- un data pane, dove verranno inseriti i parametri input,
- un content pane, che visualizzerà le informazioni associate per tipologia di agente,
- un log pane, dove viene mantenuta traccia degli agenti istanziati e dei dettagli associati ai parametri in input.

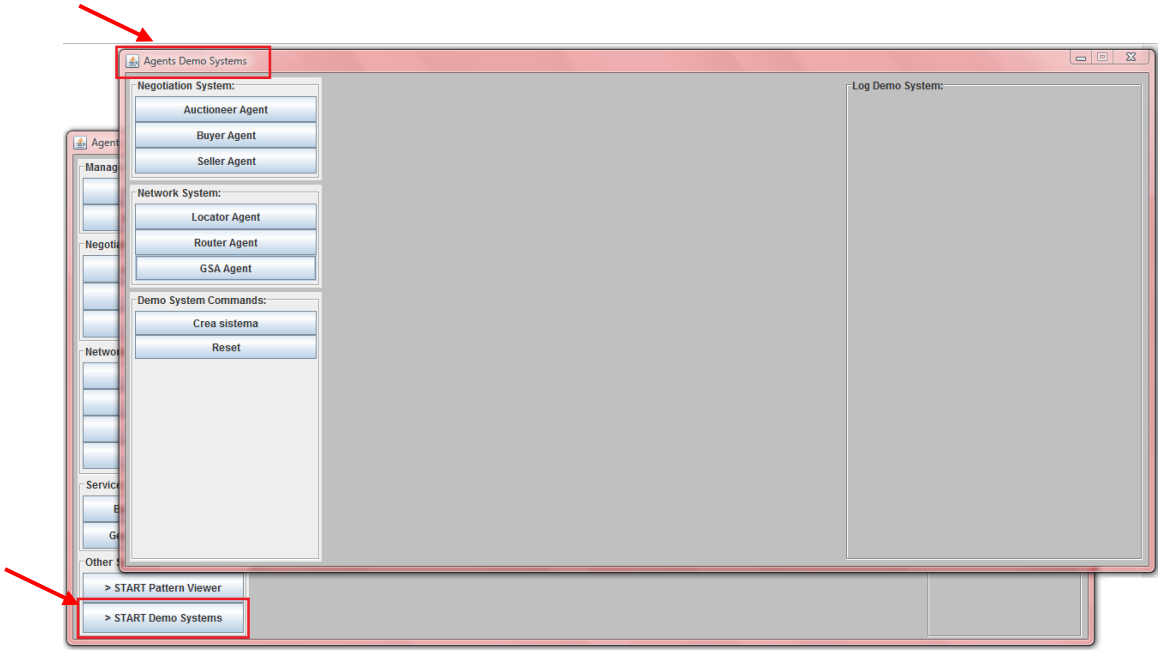


Figura 6.11 – Avvio Agents Demo Systems

Figura 6.12 – Componenti del Pannello Agents Demo Systems

Creazione dei sistemi.

I sistemi demo vengono eseguiti una volta determinati agenti, numerosità per tipologia e parametri di ogni agente.

Il metodo che viene chiamato per l'attivazione è il seguente:

```
public String createSys() {
       String path = new java.io.File("").getAbsolutePath();
       String binPath = path + "\\bin\\";
       String libPath = path + "\\lib\\";
       String command = "java -classpath " + binPath + ";" + libPath
+ "jade.jar;" + libPath + "commons-codec-1.3.jar jade.Boot -gui -
agents ";
       for (int i=0; i<idx; i++) {
           if (i>0 && i<idx) {
               command = command + ";";
           }
           command = command + agents[i];
       }
       command = command + ";sniffer:jade.tools.sniffer.Sniffer(*)";
       return command;
   }
```

Il percorso della cartella di progetto viene sempre ottenuto in maniera dinamica, vengono poi richiamate le classi di JADE che permettono di istanziare gli agenti creati e seguirne la comunicazione attraverso l'agente Sniffer, al quale viene dato in input la label (*) per la visualizzazione di tutti gli agenti della piattaforma.

6.4.1 Negotiation System

Il sistema negoziazione è composto da tre tipologie di agenti:

- Seller agent
- Auctioneer agent
- Buyer agent

I test sono stati effettuati con in media, un venditore, un mediatore, da due a quattro compratori.

Dal menù laterale del pannello Agents Demo Systems è possibile selezionare quanti più agenti per tipologia vengano richiesti e inserire per ognuno i parametri in input.

Attivazione di un agente.

Dopo aver selezionato un agente si procede con l'inserimento dei parametri da dare in input. In particolare:

Per l'agente mediatore non devono essere forniti dati in quanto gli vengono comunicati dal venditore e dagli acquirenti durante la negoziazione. Viene specificato solamente il nome dell'agente.

Per l'agente venditore devono essere specificati i seguenti dati:

- *Nome Agente*: Campo di testo. Si richiede di non inserire spazi nel nome che potrebbero causare errori per la creazione del file .java annesso;
- *Goods*: Array di stringhe, i valori devono essere separati con un carattere ';' - array che contiene i beni posseduti da vendere;
- *Prices*: Array di interi, i valori devono essere separati con un carattere ';' – array che contiene i prezzi di partenza, intesi come prezzi base, di vendita dei beni. La sequenzialità dei prezzi segue la sequenzialità dei beni ai quali sono associati;
- *Estimated Prices*: Array di interi, i valori devono essere separati con un carattere ';' – array che contiene i prezzi stimati per la vendita, intesi come l'incasso desiderato che il venditore vuole ottenere;
- *Auction Type*: Campo di testo, l'insieme di valori è contenuto nel seguente insieme:

  T = {*null*;English;Dutch;Vickery}

  Se il campo viene lasciato vuoto (corrispondente a valore *null*) l'asta viene decisa dal mediatore in base alla volontà del venditore, ovvero in base ad un rapporto stimato tra il prezzo base di partenza e il prezzo desiderato di vendita. Altrimenti va scelta una delle aste. All'utente la decisione sulla scelta da effettuare per verificarne il funzionamento.

Per l'agente compratore devono essere forniti in input:

- *Nome Agente*: Campo di testo. Si richiede di non inserire spazi nel nome che potrebbero causare errori per la creazione del file .java annesso;
- *Budget*: Campo numerico. Viene completato con il budget totale posseduto dal compratore, che andrà a scalare quando un bene verrà acquistato;
- *Goods*: Array di stringhe, i valori devono essere separati con un carattere ';' - array che contiene i beni che si desiderano acquistare;
- *Prices*: Array di interi, i valori devono essere separati con un carattere ';' – array che contiene i prezzi massimi auspicati per l'acquisto del bene. La sequenzialità dei prezzi segue la sequenzialità dei beni ai quali sono associati;
- *Overshoot*: Campo numerico, nel valore da trascrivere non va riportata la percentuale. Questo campo rappresenta la percentuale massima di sforo a partire dal prezzo desiderato per l'acquisto. Se la negoziazione supera il prezzo desiderato maggiorato della percentuale scelta l'agente abbandona l'asta.

Esempi di parametri utilizzati per il test.

Seller Agent:

- *Nome Agente*: Seller1
- *Goods*: quadro;casa
- *Prices*: 1000;100000
- *Estimated Prices*: 4000;200000
- *Auction Type*:

Buyer Agent:

- *Nome Agente*: Buyer1
- *Budget*: 200000
- *Goods*: quadro;casa
- *Prices*: 3500;180000
- *Overshoot*: 20

Buyer Agent:

- *Nome Agente*: Buyer2
- *Budget*: 230000
- *Goods*: quadro;casa
- *Prices*: 3500;200000
- *Overshoot*: 15

Buyer Agent:

- *Nome Agente*: Buyer3
- *Budget*: 250000
- *Goods*: quadro;casa
- *Prices*: 3900;230000
- *Overshoot*: 20

Per ogni agente, dopo aver inserito i parametri, va cliccato il tasto 'Conferma'. L'esecuzione di questo comando avvia la procedura seguente:

1. scrive un file .txt con i dati indicati nel data pane, i quali saranno successivamente letti dall'agente,
2. crea l'istanza dell'agente,
3. inserisce nel log la traccia di creazione dell'agente, la quale riporterà il nome dell'agente e i dati associati

Si riporta, di seguito, il codice che crea il file di parametri dell'agente:

```
try {
   String path = new java.io.File("").getAbsolutePath() +
"\\bin\\Agents\\File\\";
   FileOutputStream fout = new
       FileOutputStream(path+gsa_name_t.getText()+"_param.txt");
   PrintStream fwriter = new PrintStream(fout);
   fwriter.println(gsa_psv_t.getText());
   fwriter.println(gsa_rsv_t.getText());
   fout.close();
} catch (Exception exc) {
    System.out.println("Errore: " + exc.getMessage());
}
```

Creazione del sistema.

Dopo aver creato gli agenti voluti si passa alla creazione del sistema nella piattaforma JADE attraverso il comando 'Crea sistema' nella sezione *Demo Systems Commands*. Selezionando il comando viene eseguito un comando batch che crea le istanze degli agenti nella piattaforma e inizia automaticamente la negoziazione. Il comando attiva anche lo Sniffer agent che traccia tutte le comunicazioni tra gli agenti.

Di seguito viene riportato lo schema UML sequenziale delle comunicazioni tra gli agenti coinvolti, con rappresentazione delle temporizzazioni.

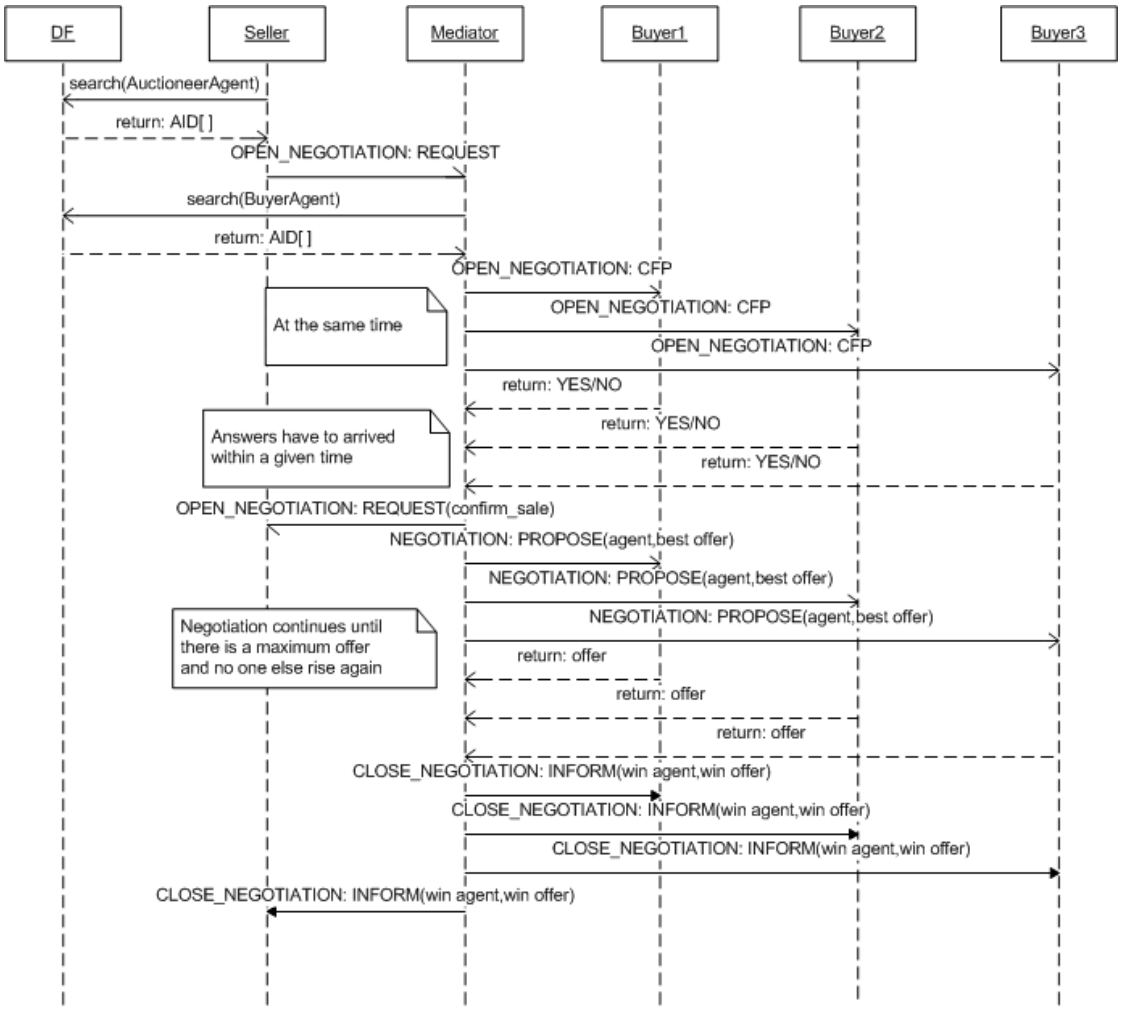


Figura 6.13 – Schema sequenziale UML del Sistema negoziazione

6.4.2 Network System

Il sistema di rete è composto anch'esso da tre tipologie di agenti:

- Router agent
- Locator agent
- Generic Service agent

Questo sistema è stato creato per rappresentare un modello generico di rete in cui operano degli agenti che forniscono servizi generici, un router che gestisce le comunicazioni tra le varie subnet e un locator che fornisce i dati necessari, container e provider, per le migrazioni opportune.

Il sistema demo fornito simula il funzionamento di una piccola rete con n dispositivi mostrando attraverso l'agente Sniffer lo spostamento tra i vari container.

I test di corretta esecuzione del sistema sono stati infatti condotti creando una singola istanza per agente locator e agente router, e n istanze degli agenti di servizio.

Dal menù laterale del pannello Demo Systems è possibile selezionare quanti più agenti per tipologia vengano richiesti e inserire per ognuno i parametri in input

Attivazione di un agente.

Dopo aver selezionato un agente si procede con l'inserimento dei parametri da dare in input. In particolare:

Per gli agenti router e locator non sono previsti parametri in input. Gli agenti infatti ricevono rispettivamente messaggi dagli agenti che richiedono un servizio e messaggi dal router.

Per gli agenti di servizio si devono fornire i seguenti parametri:

- *Nome Agente*: Campo di testo. Si richiede di non inserire spazi nel nome che potrebbero causare errori per la creazione del file .java annesso;
- *Provided Service*: Campo di testo. Il campo contiene il servizio che un agente fornisce;
- *Requested Services*: Array di stringhe, i valori devono essere separati con un carattere ';' - array che contiene i servizi che l'agente deve richiedere ad altri provider;

Esempi di parametri utilizzati per il test.

Generic Service Agent:

- *Nome Agente*: serv1
- *Provided Service*: storage
- *Requested Services*: memory;query analizer

Generic Service Agent:

- *Nome Agente*: serv2
- *Provided Service*: call
- *Requested Services*: storage;video

Generic Service Agent:

- *Nome Agente*: serv3
- *Provided Service*: call
- *Requested Services*: storage;video

Generic Service Agent:

- *Nome Agente*: serv4
- *Provided Service*: query analizer
- *Requested Services*: storage

Generic Service Agent:

- *Nome Agente*: serv5
- *Provided Service*: storage
- *Requested Services*: call, video

Generic Service Agent:

- *Nome Agente*: serv6
- *Provided Service*: video
- *Requested Services*: null

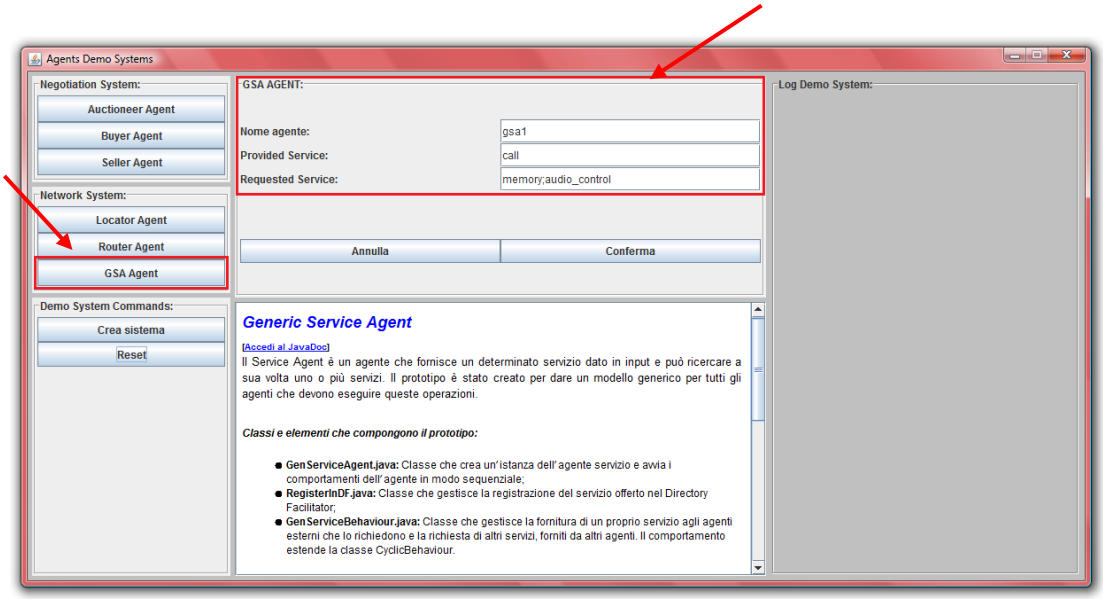


Figura 6.14 – Inserimento parametri Generic Service Agent

Creazione del sistema.

Dopo aver creato gli agenti voluti si passa alla creazione del sistema nella piattaforma JADE attraverso il comando 'Crea sistema' nella sezione *Demo Systems Commands*. Selezionando il comando viene eseguito un comando batch che crea le istanze degli agenti nella piattaforma e inizia automaticamente l'esecuzione del sistema. Nell'esecuzione gli agenti sono associati a due dispositivi differenti. Il comando contenuto nel file .bat attiva quindi uno Sniffer agent per ogni container che traccia tutte le comunicazioni tra gli agenti. Gli agenti di servizio interrogano il router che ritorna i provider dei servizi richiesti e la relativa posizione. Se il container è il medesimo l'agente inizierà una richiesta diretta all'agente, in caso contrario migrerà nel dispositivo destinatario e nella nuova piattaforma inizierà lo scambio di messaggi. Di seguito viene riportato lo schema UML sequenziale delle comunicazioni tra gli agenti coinvolti, con rappresentazione delle temporizzazioni.

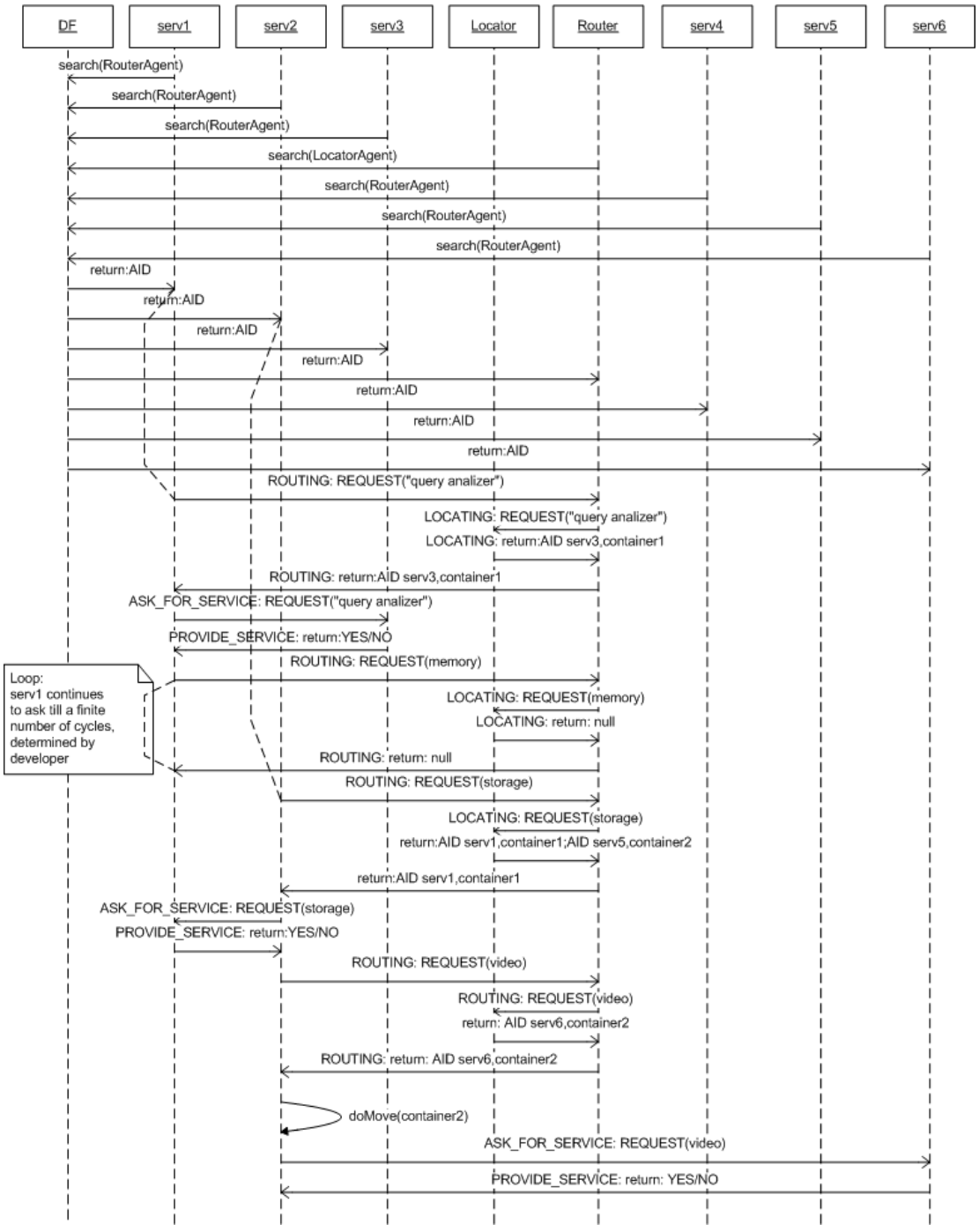


Figura 6.15 - Schema sequenziale UML del Sistema Network

7 Conclusioni

Nell'ultimo capitolo del lavoro di tesi si traggono le conclusioni, viste in un'ottica di sviluppo futuro del sistema nonché di studio sugli agenti e sulle nuove metodologie di interconnessioni tra diverse tipologie di reti per supportare e garantire un servizio continuo nelle reti mobili ad-hoc.

7.1 Sistema interattivo

Il risultato di questo lavoro è un framework innovativo per lo sviluppo di applicazioni ad agenti, non presente nella letteratura attuale e passata. Le piattaforme offrono librerie e supporti agli ambienti di sviluppo, ma mancano di dare al progettista un aiuto per le fasi iniziali.

Il sistema interattivo proposto introduce dei modelli definiti secondo il paradigma di programmazione ad agenti, interpretati nei prototipi come entità multi comportamento. Ciò ha consentito di ottenere un'infrastruttura software in grado di sviluppare i *MAS* all'interno dei quali le attività degli agenti stessi, a seconda delle esigenze, vengono eseguiti singolarmente, sequenzialmente o parallelamente. I behaviour vengono a loro volta strutturati a step o seguono cicli while che gestiscono i diversi stati che possono essere raggiunti.

Il sistema interattivo proposto può essere il punto di partenza per lo sviluppo di un sistema più completo e complesso. I prototipi forniti sono modelli di default semplici, che strutturano i più comuni ruoli associabili agli agenti. Il framework può integrarli fornendo classi di supporto per nuovi ruoli o articolando ulteriormente quelli già forniti.

Uno degli obiettivi principali potrebbe essere quello di uno sviluppo ulteriore del supporto ai design pattern, integrando il modulo Pattern Viewer di nuove funzionalità per ottenere una maggiore efficienza.

Alcune proposte potrebbero essere:

- L'aggiunta di ulteriori modelli architetturali per strutturare le comunicazioni,

- un aumento del range dei valori forniti per i pattern esistenti,
- una gestione maggiormente dinamica delle immagini proposte per favorire la scalabilità del framework,
- l'eventuale possibilità di esportare il modello editato in vari formati.

Il framework potrebbe inoltre addizionare all'infrastruttura ulteriori moduli per favorire la progettazione del sistema ad agenti in fase di creazione. Esempi di moduli da integrare potrebbero essere:

- un modulo di supporto alla grafica per tracciare i caratteri di mobilità e comunicazione tra gli agenti, ad esempio integrando il sistema con nuovi prototipi dedicati proprio alla struttura delle connessioni;
- un modulo di supporto alla strutturazione dei comportamenti da associare agli agenti, con implementazione di prototipi comportamentali per diverse casistiche di problem solving;
- un modulo per la gestione del deployment che può sfruttare la piattaforma JADE attraverso l'implementazione di un'interazione dinamica, come già utilizzata nel framework, o aggiungere ex-novo il supporto a questa fase di rilascio; in aggiunta possono essere integrate le funzionalità di monitoraggio, debugging e controllo delle connessioni.

Il vantaggio offerto dalla progettazione del framework stesso come infrastruttura modulare, consiste nella facilità di renderlo scalabile con la semplice aggiunta di moduli che gestiscono indipendentemente le funzioni dedicate.

Durante gli studi della tecnologia ad agenti mobili e la progettazione del sistema interattivo si è riscontrato l'effettivo bisogno di un framework con le funzionalità proposte, data la complessità degli stessi e dalla mancanza di strumenti completi per il loro sviluppo. Si è potuto verificare in questo modo la bontà del progetto nel suo complesso e del lavoro di ricerca e analisi iniziale che trova riscontro nelle implementazioni effettuate. Dal punto di vista personale, si sottolinea l'importanza di un tale sistema, in quanto in circolazione il materiale fornito per la manipolazione del paradigma trattato risulta molto scarno. Nella ricerca si sono rivelati di fondamentale importanza papers molto recenti, delle ultime conferenze e workshop proprio su queste tecnologie. Se da un lato le piattaforme a pagamento, come Voyager, risultano le più funzionali per supportare le problematiche risultanti dalla mobilità degli apparati, dall'altro soluzioni open source mancano di alcune semplicità di utilizzo, ma con uno sviluppo intensivo potrebbero assumere un ruolo dominante nelle applicazioni di domani.

7.2 Sistemi multi agente

I sistemi multi-agenti nelle reti MANET hanno come obiettivo futuro quello di ottimizzare gli algoritmi dinamici di localizzazione di dispositivi e servizi. L'ottimizzazione mira al raggiungimento di alte prestazioni in termini di riduzione

temporale per le operazioni di ricerca e di riduzione del consumo energetico a livello computazionale. Ogni agente deve essere in grado di ottenere dal dispositivo in loco le informazioni topologiche e di posizionamento, non solo del proprio dispositivo ma anche degli altri appartenenti al raggio di connettività e nel minor tempo possibile. Gli obiettivi si possono riassumere nei seguenti punti:

1. Ottimizzazione del traffico attraverso la riduzione del tempo di occupazione della banda durante il processo di migrazione;
2. Autonomia degli agenti sempre più sviluppata. Questi oggetti evoluti devono essere capaci di gestire e prendere decisioni in qualsiasi situazione a portata, permettendo operazioni off-line in tutta sicurezza ed efficacia;
3. Utilizzo di metodologie cooperative che permettano ai nodi di avere comportamenti assimilabili nei casi di:
    1. Ricezione delle informazioni di range e posizionamento dai nodi vicini,
    2. Risoluzione di un problema locale di posizionamento,
    3. Trasmissione del risultato ottenuto ai vicini.
4. Implementazione di nuovi algoritmi distribuiti , particolarmente usabili quando la mobilità dei nodi è limitata. Algoritmi che non richiedono un meccanismo di comunicazione globale, ma gestibili on-demand nella frazione di secondo necessaria. Per comunicazione globale s'intende un processo di analisi critica dello scenario di riferimento e degli agenti di interesse che vi operano.

7.3 Paradigma Grid

Nello sviluppo di una potente infrastruttura che deve fornire servizi garantendo una certa qualità di servizio (QoS), la scarsità di risorse rappresenta un ostacolo. Questo è stato il punto di partenza per studiare un probabile coordinamento e una successiva integrazione tra le reti fisse, siano esse wireless o cablate, e le reti mobili. Questa tipologia di sistema ibrido viene definito come sistema Grid.

Il paradigma Grid Computing si è infatti rivelato una valida soluzione per implementare strategie di gestione distribuita con l'obiettivo di fornire sempre più servizi sofisticati, con elevati livelli di qualità. Questo paradigma si riferisce ad una nuova infrastruttura di calcolo distribuito che fornisce un metodo innovativo per l'accesso e la distribuzione di dati e risorse. L'idea su cui si basa la rete grid è appoggiare, qualora sia possibile, una rete mobile ah hoc ad una rete fissa esistente nel raggio di copertura evidenziato. In questo modo molte operazioni pesanti in termine di *resource consumption* per i dispositivi mobili, possono essere prese in carico da host fissi, più performanti. Tra le operazioni che evidenziano criticità nelle MANET ci sono il bilanciamento del carico, la gestione QoS, e la questione della sicurezza. Mentre un sistema fisso potrebbe svolgere compiti complessi per una rete mobile, in modo parallelo garantendo stabilità maggiore nelle connessioni, una rete ad hoc, puramente peer to peer, potrebbe focalizzarsi sulla localizzazione di servizi e risorse all'interno del raggio.

Al momento, l'utilizzo di un sistema a griglia mobile nella progettazione di middleware per ambienti ad hoc sembra essere una delle direzioni più interessanti da perseguire.

8 Bibliografia

[1] D. Bruneo, A. Puliafito, M. Scarpa, *Mobile Middleware: Definition and Motivations*, In: A. Corradi *Mobile Middleware*, London: CRC press (UK), 2006.

[2] M. Fazio, M. Villari, A. Puliafito - *Improvement Of Simulative Analysis In Ad Hoc Networks*. Published on International Journal of Business Data Communications and Networking (IJBDCN), IGI Publishing , January 2009

[3] Alberto Grosso, Christian Vecchiola, Antonio Boccalatte, *Un'infrastruttura per la Mobilità in AgentService*, WOA, Workshop from Objects to Agents, 2005

[4] Ming Kin Lai, *Programming Languages for Mobile Code*, ACM Computing Surveys, 2007

[5] Leonardo Puleggi, *Java e Serializzazione*, Corso di Sistemi Distribuiti, UNIVERSITA' DEGLI STUDI DI MILANO-BICOCCA, 2009

[6] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa, *Bytecode Transformation for Portable Thread Migration in Java*, Department of Information Science, Faculty of Science, University of Tokyo, 2002

[7] Mohammad Muztaba Fuad and Debzani Deb, *Java object migration using intelligent software agents*, International Conference on Communications and Information Technology ICCIT 2002

[8] Oscar Urra, Sergio Ilarri, Raquel Trillo and Eduardo Mena, *Mobile Agents and Mobile Devices: Friendship or Difficult Relationship?*, JOURNAL OF PHYSICAL AGENTS, VOL. 3 NO. 2 MAY 2009

[9] Raquel Trillo, Sergio Ilarri and Eduardo Mena, *Comparison and Performance Evaluation of Mobile Agent Platforms*, ICAS 2007

[10] Rohit Mangla, *Mobile Ad Hoc Networks*, International Journal of Educational Administration ISSN 0976-5883 Volume 2, Number 4 (pp. 697-706), 2010

[11] Tanya Araújo, and Francisco Louçã , *Modeling a Multi-Agents System as a Network: A Metaphoric Exploration of the Unexpected*, Networking and Telecommunications: Concepts, Methodologies, Tools and Applications (pp. 1728-1740), 2010

[12] Geraud Allard, Pascale Minet, Dang-Quan Nguyen1, and Nirisha Shrestha, *Evaluation of the Energy Consumption in MANET*, AdHoc-Now06, 5th international conference on AdHoc and Wireless", Ottawa, Canada, August 2006.

[13] Maitreya Natu, Adarshpal Sethi, *Integrating Detection and Faul Localization in MANETS*, Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2011

[14] IEEE Computer Society, *Group Communications in Mobile Ad Hoc Networks*, 2004

[15] Justin W. Dean, Joseph P. Macker, William Chao, *A study of multiagent system operation within dynamic ad hoc networks*, IEEE Military Communications Conference (MILCOM) 2008 Proceedings, 17-19 Nov 2008, San Diego, CA.

[16] Emerson Ferreira, *Implementing Mobile Agent Design Patterns in the JADE framework*, Paper 2003

[17] K. Yasser, A.Hesham, N. Elmahdi, S. Allola, H. Ahmad., *Optimizing Mobile Agents Migration Based on Decision Tree Learning*, Proceedings of World Academy of Science Engineering and Technology. 2007

[18] Mohammed Eshtay, Hierarchal Traveling Design Pattern for Mobile Agents in JADE Framework, ICIT 2009

[19] Okuthe P. Kogeda and Johnson I. Agbinya, *Cellular Network Fault Prediction Using Mobile Intelligent Agent Technology*, Southern Africa Telecommunication Networks and Applications Conference, 2007

[20] Sasu Tarkoma, *Mobile Middleware: Architecture, Patterns and Practice*, John Wiley & Sons, L td. ISBN: 978-0-470-74073-6, 2009

[21] Manuela Mantione, *Analisi delle prestazioni di sistemi ad agenti mobili per la rimodulazione dell'organizzazione e dei comportamenti*, Tesi di Laurea Specialistica, Anno Accademico 2006/2007

[22] Fabio Bellifemine, Giovanni Caire, Dominic Greenwood, *Developing multi-agent systems with JADE*, John Wiley & Sons, 2007

[23] JADE Foundation, http://jade.tilab.com/

[24] JADE programming for beginners, http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf

[25] JADE programmer's guide, http://jade.tilab.com/doc/programmersguide.pdf

[26] JADE administrator's guide, http://jade.tilab.com/doc/administratorsguide.pdf

[27] JADE tutorial Application-de_ned content, languages and ontologies, http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf

[28] LEAP user guide, http://jade.tilab.com/doc/tutorials/LEAPUserGuide.pdf