Università degli studi di Padova

Facoltà di Ingegneria

Tesi di Laurea Magistrale in

Ingegneria Informatica

Infrastruttura ad agenti per sistemi ubiqui di ausilio alla viabilità in ambiente urbano

Relatore
Prof. Carlo Ferrari

Laureando
Dario Menegon

Anno Accademico 2010/2011

Ai miei genitori,

con amore e riconoscenza

Sommario

L'ubiquitous computing e i sistemi di comunicazione si stanno espandendo sempre più andando a coprire diversi ambiti applicativi. In un prossimo futuro contribuiranno ad aumentare l'efficienza e la qualità della vita delle persone grazie all'interazione sempre più frequente con oggetti della vita quotidiana che saranno dotati di capacità di elaborazione. Questo lavoro di tesi mira a dimostrare come l'utilizzo del concetto di ubiquità abbinato al paradigma ad agenti mobili porti ad avere un sistema affidabile, efficace ed efficiente rispetto ad altre tecnologie come il noto client-server oppure il P2P classico. La dimostrazione passa tramite l'implementazione di un'infrastruttura d'esempio che cerca di sottolineare i vantaggi che emergono utilizzando questi paradigmi. Nel sistema realizzato gli agenti hanno un ruolo fondamentale nel rendere il più possibile trasparente all'utilizzatore finale le operazioni di calcolo e di ricerca. Nell'ambito del progetto è stata realizzata anche l'integrazione nel sistema di un dispositivo mobile dotato del sistema operativo Android.

Abstract

Ubiquitous computing and communication systems are now spreading every aspect of our daily life and are expected to profoundly change our life in the near future. They will contribute to increase efficiency and life quality of people to perform their daily activities in a world of increasing mobility and easy access to all kinds of services. This thesis work is focussed on the demonstration that the use of ubiquity with an agent-based paradigm builds a reliable, effective and efficient system compared to other technologies, as the well-known client-server or the generic P2P. The system described in this paper proves the implementation of a specific infrastructure that emphasizes the advantages of the usage of ubiquitous and agent-based paradigms. Here, agents have a very important role in the specific system making all processing and research operations transparent. During the implementation an Android mobile device has been integrated to the system.

Indice

CAPITOLO 1

Introduzione

Indice

Il cambiamento e l'evoluzione della tecnologia alterano, in maniera sempre più marcata, il luogo in cui questa si pone all'interno della vita quotidiana dando sempre maggiore rilevanza alla relazione che essa ha con le persone. Negli ultimi cinquant'anni di evoluzione tecnologica ci sono stati due grandi trend in questa relazione: prima quella con il mainframe e a seguire quella con il PC. Internet, in questa fase di transizione, ci sta portando ora verso l'*ubiquitous computing*, una nuova tipologia di relazione caratterizzata da un ampio utilizzo del calcolo distribuito. I sistemi ubiqui richiedono un nuovo tipo di approccio per adattare la tecnologia alla vita delle persone, criterio che viene detto *calm technology*. Questo nuovo modo di vedere la tecnologia, in questi ultimi anni, ha aperto le porte a nuove tipologie di applicazioni che vanno sempre più in aiuto dell'utente umano.

Gli ambiti di interesse sono davvero molteplici tra cui spicca, ad esempio, la robotica nella quale si sono ottenuti degli ottimi risultati in termini di ubiquità [2, 3]. Prendendo in considerazione un altro settore, quello della viabilità e dei trasporti, si trovano diversi studi che mirano ad avvolgere le persone con soluzioni riguardanti la sicurezza, la comunicazione e l'efficienza. Parole, queste, che sono alla base della ricerca in questo campo per migliorare sempre più il servizio che la tecnologia può offrire all'uomo. Si può sottolineare, ad esempio, il lavoro che viene svolto dal *CAR 2 CAR Communication Consortium* [21] che ha come obiettivi lo sviluppo di uno standard europeo per i sistemi di trasporto intelligenti, per quelli di comu-

nicazione interveicolare e per il dialogo tra veicoli ed elementi intrafrutturali della strada. Gli ambiti applicativi sono molteplici e puntano ad essere un valido strumento di supporto al guidatore. In questo panorama si trovano anche i parcheggi: sono numerosi, infatti, gli studi che sono stati fatti per migliorare sia il controllo delle infrazioni che la comunicazione al guidatore di informazioni utili. Alcuni comuni hanno commissionato ad aziende esterne lo sviluppo di sistemi che risolvessero queste problematiche: si citano a titolo di esempio l'*IPark Trevisosta* di Treviso [22, 23] e la possibilità di consultare via web la situazione dei parcheggi cittadini per il comune di Verona [24].

L'utilizzo del concetto di ubiquità anche nel settore dei trasporti e della viabilità può portare ad ottime soluzioni che vanno a migliorare sempre più l'efficienza nell'utilizzare la tecnologia da parte dell'uomo. La possibilità di rendere trasparente all'utente il sistema sottostante può essere, inoltre, uno dei punti di forza per lo sviluppo e l'affermazione nel mercato.

I sistemi ubiqui stanno alla base del nuovo modo di vedere la tecnologia. Negli ultimi anni, infatti, si è affermato sempre più il cosiddetto *ubiquitous computing* che realizza il concetto di onnipresenza ponendo la tecnologia a pieno supporto delle necessità dell'utente umano. I sistemi possono essere realizzati attraverso diverse architetture che dipendono dalle specifiche dell'applicazione. Tra le possibili si citano il tipico client-server, un'architettura peer-to-peer e una sua specializzazione ad agenti mobili. Quest'ultimo paradigma, in particolare, sta riscuotendo sempre più successo nella comunità scientifica e molti studi ruotano attorno ad esso. La definizione di standard e di piattaforme che lo utilizza ha portato a un suo crescente impiego per realizzare anche sistemi ubiqui.

1.1 Obiettivi della tesi

L'obiettivo principale della tesi è dimostrare come l'utilizzo delle più recenti tecnologie per realizzare i sistemi ubiqui non offre solo una possibilità implementativa ma anche che diventa un vero e proprio punto di forza in termini di efficienza e di affidabilità dei sistemi stessi. Al fine di raggiungere dei risultati per trarre le conclusioni è stato necessario implementare un'infrastruttura di esempio che segue un'analisi delle tecnologie disponibili. Il lavoro che è stato svolto, quindi, mira a dimostrare che l'utilizzo di un tale sistema permette all'utente di avere a disposizione una serie di servizi e funzionalità che vengono svolti in totale trasparenza senza escludere aspetti di efficienza ed affidabilità.

L'utilizzo di una piattaforma multiagente per il sistema ubiquo di riferimento è stata la vera sfida della realizzazione dell'infrastruttura di esempio. Le funzionalità, realizzate sfruttando le caratteristiche dei sistemi agent-based, sono sufficienti per dimostrare anche l'efficacia di una tale soluzione rispetto ad altre tecnologie.

Il settore che è stato scelto per l'esempio è quello della viabilità con particolare riferimento alla problematica dei parcheggi. L'infrastruttura che è stata realizzata è un esempio pratico di come i sistemi ubiqui possano essere di supporto all'uomo in situazioni anche critiche. Nell'implementazione sono stati presi in considerazione gli esempi dei parcheggi comunali di Treviso e Verona riportati precedentemente.

1.2 Struttura della tesi

La tesi è formata, oltre a questo capitolo introduttivo, da altri quattro capitoli e nel dettaglio sono strutturati nel seguente modo:

- *capitolo 2*: nel corso di questo capitolo vengono presentate le caratteristiche principali dei sistemi ubiqui e dei sistemi multiagente;
- *capitolo 3*: in questo capitolo vengono illustrate tutte le tecnologie utilizzate nello sviluppo del lavoro di tesi;
- *capitolo 4*: questa parte consiste nella presentazione del progetto di tesi illustrando il lavoro nel dettaglio dal punto di vista implementativo;
- *capitolo 5*: in quest'ultimo capitolo vengono tratte le conclusioni del lavoro di tesi e vengono presentati degli spunti per gli sviluppi futuri.

CAPITOLO 2

Sistemi ubiqui e sistemi multiagente

Indice

In questo capitolo vengono presentate dettagliatamente le caratteristiche dei sistemi ubiqui e di quelli multiagente mettendo in luce i loro punti di forza. Poiché il lavoro di tesi utilizza questi concetti, questa parte dello scritto risulta essere un valido supporto al lettore non esperto per acquisire le conoscenze necessarie e poter affrontare meglio i capitoli che seguono. Nella prima parte del capitolo viene presentato il nuovo mondo dell'*Ubiquitous Computing* evidenziando le specifiche di base e le numerose applicazioni che possono sfruttarlo. A seguire vengono illustrati i punti principali dei sistemi agent-based che possono essere utilizzati per implementare un sistema ubiquo, come è stato fatto nel lavoro di tesi.

2.1 Sistemi ubiqui

Ripercorrendo la storia dell'informatica, si può riscontrare un netto cambiamento nel modo di utilizzare la tecnologia da parte delle persone. Se nei primi anni, nell'era dei mainframe, un computer doveva essere utilizzato da molte persone e quindi condiviso tra di loro, con l'avvento dei personal computer, come dice il nome stesso, si riesce, in linea di massima, ad associare a una persona un computer. Nell'attuale fase di transizione, grazie all'aumento dei servizi offerti da Internet e al miglioramento dei dispositivi, si può affermare di essere nella situazione in cui ciascuna persona ha più di un dispositivo a disposizione. L'ubiquitous computing, detto anche *ubicomp*, cambierà in modo netto il tipo di rapporto che l'uomo ha con la tecnologia infatti ciascuno di noi avrà a disposizione molti computer condivisi: possono essere quei dispositivi che vengono interrogati durante la navigazione in Internet o altri inglobati in muri, sedie, vestiti, automobili o in qualsiasi altro oggetto della vita quotidiana. L'ubicomp è caratterizzata fondamentalmente dalla connessione di cose nel mondo con la possibilità di computazione ed ha radici in diversi ambiti dell'informatica. Nel 1988 Mark D. Weiser, uno dei componenti del Computer Science Lab della Xerox PARC, coniò per primo l'espressione *Ubiquitous Computing* divenendo di fatto il padre di tale modo di pensare la tecnologia. Weiser scrisse infatti il seguente abbozzo di articolo, ricordato col nome *Ubiquitous Computing #1* [25], nel quale è evidente la sua visione futura:

> *Inspired by the social scientists, philosophers, and anthropologists at PARC, we have been trying to take a radical look at what computing and networking ought to be like. We believe that people live through their practices and tacit knowledge so that the most powerful things are those that are effectively invisible in use. This is a challenge that affects all of computer science. Our preliminary approach: Activate the world. Provide hundreds of wireless computing devices per person per office, of all scales (from 1"*

> *displays to wall sized). This has required new work in operating systems, user interfaces, networks, wireless, displays, and many other areas. We call our work 'ubiquitous computing'. This is different from PDA's, dynabooks, or information at your fingertips. It is invisible, everywhere computing that does not live on a personal device of any sort, but is in the woodwork everywhere.*

Visti i numerosi passi in avanti sulla miniaturizzazione dei dispositivi e sull'ottimizzazione delle operazioni wireless, la comunità scientifica sembra orientata alla realizzazione e alla concretizzazione dei concetti forniti dall'ubicomp. È da sottolineare tuttavia come il concetto di sistema ubiquo non coincida esattamente con quello di realtà virtuale: mentre quest'ultima proietta le persone in un mondo generato da un computer, l'ubiquitous computing forza il computer a vivere nel mondo reale delle persone presentando una difficile integrazione di fattori umani, informatica, ingegneria e scienze sociali.

2.1.1 Caratteristiche dei sistemi ubiqui

L'ubiquitous computing può essere spiegato, da un punto di vista concettuale, con l'espressione *anytime anywhere*, ovvero deve offrire le funzionalità proposte dai sistemi ovunque in qualsiasi momento. Questo porta a pensare come in un prossimo futuro sarà possibile eseguire azioni remote per controllare diversi sistemi utilizzando reti, computer e software. I sistemi ubiqui sono utilizzati in vari settori e sfruttano ampiamente il controllo di rete, la supervisione e le tecnologie del calcolo distribuito. I progettisti di tali sistemi hanno come obiettivo quello di integrare, in modo sempre maggiore e migliore, i dispositivi di calcolo in vari oggetti fisici e in diversi luoghi. Solitamente queste unità di calcolo hanno la possibilità di comunicare in wireless.

In letteratura si trovano numerose discussioni che riguardano i sistemi ubiqui, sia dal punto di vista concettuale che da quello implementativo. Nelle prossime sezioni vengono presentati alcuni aspetti che sono oggetto di ricerca.

2.1.1.1 L'integrazione fisica

Un sistema ubiquo coinvolge l'integrazione tra i nodi di calcolo e il mondo fisico. In letteratura si trovano numerose considerazioni che evidenziano come l'ubicomp abbia luogo in *environment* ben definiti quali una stanza, un'automobile o un edificio. Si ritiene necessario, pertanto, suddividere il mondo ubiquo in diversi ambienti caratterizzati da confini che demarcano il loro contenuto senza limitare l'interoperabilità tra essi. Si possono pensare diverse soluzioni per l'interazione tra due environment, come ad esempio l'utilizzo di componenti mobili. Per integrare gli ambienti di calcolo con

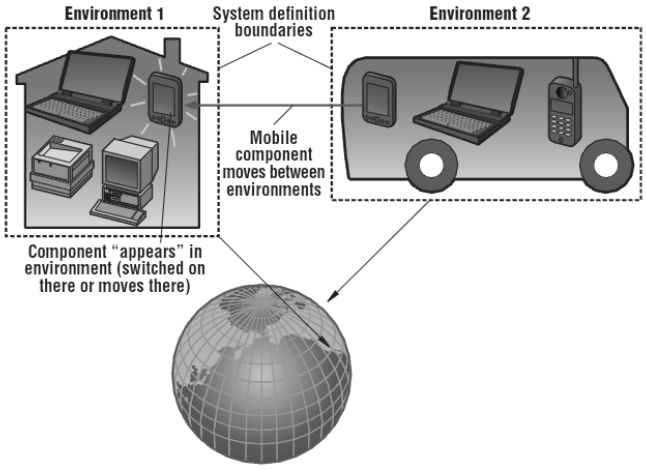


Fig. 2.1: Il mondo visto come insieme di vari environment

il mondo fisico è necessario avere a disposizione delle interfacce a basso livello che consentono al software di lavorare con i sensori e gli attuatori e dei framework ad alto livello che consentono alle applicazioni di sentire e operare nel loro ambiente: un framework generico verrà presentato nella sezione 2.1.2. In figura 2.1, tratta da [25], si può osservare un esempio di diversi ambienti ben delimitati.

L'ubiquità sarà un concetto sempre più concreto nel futuro e si affiderà totalmente alle tecnologie informatiche. I sistemi ubiqui, però, sono di natura piuttosto complessi dal punto di vista della creazione e della gestione: per questo motivo saranno sempre più richiesti framework ed interfacce volendo assicurare l'azione remota sui dispositivi, un controllo sicuro a distanza e la possibilità di avere informazioni in qualsiasi punto ci si trovi. Un tale sistema dovrebbe comunque essere progettato con visione a medio-lungo termine per permettere l'integrazione con i nuovi dispositivi o il nuovo modo di utilizzarli. Risulta improponibile infatti pensare a un reboot dell'intero sistema per aggiungere nuove funzionalità. Per questo motivo, lo sviluppo dei sistemi ubiqui diventa necessariamente modulare e dovrebbe essere estensibile in modo incrementale.

2.1.1.2 L'interazione

Un ambiente contiene diversi componenti che possono essere anche unità software che implementano l'astrazione di servizi, risorse o applicazioni. Si possono distinguere due tipologie di componenti: quelli infrastrutturali, che sono in linea di massima fissi e stabili nel tempo, e quelli spontanei che consistono nei dispositivi che arrivano e lasciano il sistema liberamente.

In un sistema ubiquo viene richiesta l'interoperabilità spontanea tra le parti al fine di rendere il più possibile trasparente all'utente il processo di

elaborazione sottostante. Poiché questi sistemi sono caratterizzati da una continua variazione dei componenti presenti, in termini di utenti, hardware e software, diventa necessario identificare i criteri di interazione che governano l'intero sistema.

Per meglio definire un ubiquitous system è necessario chiarire quali sono i protagonisti delle interazioni al suo interno. Si identificano gli umani, detti anche utenti, e le macchine o dispositivi. Si possono quindi individuare quattro tipi di interazione:

- *Humans to humans* (H2H);
- *Machines to machines* (M2M);
- *Humans to machines* (H2M);
- *Machines to humans* (M2H).

Le interazioni M2M vedono coinvolti i dispositivi che comunicano direttamente senza l'intervento dell'uomo; quelle M2H possono esser viste, invece, come il feedback dalle macchine all'utente, fornendo quindi supporto alla supervisione. Per esemplificare l'interazione tra dispositivi e uomo si può pensare all'utente che riceve le informazioni mandate dalle telecamere, dai microfoni e da altri sensori posizionati nell'ambiente. L'interazione H2H, che avviene ad esempio nella telefonia o nelle videoconferenze, può essere compiuta da persone sia vicine che lontane. In quest'ultimo caso, l'interazione utilizza dei dispositivi per comunicare con l'obiettivo di rendere l'utilizzo di tali dispositivi e della rete di comunicazione il più possibile trasparente all'utente. Il tipo di interazione di maggiore interesse è la H2M in quanto rappresenta il controllo umano delle macchine e può essere caratterizzata dall'interfaccia uomo-macchina.

2.1.1.3 La scoperta di nuovi dispositivi

Uno degli aspetti che caratterizzano i sistemi ubiqui riguarda l'ingresso di un dispositivo in un ambiente e in particolare come avviene la scoperta degli altri dispositivi e dei servizi disponibili. In letteratura si trovano diversi approcci che affrontano i seguenti aspetti:

- *bootstrapping*: il nuovo dispositivo deve conoscere a priori alcuni indirizzi, come ad esempio multicast e broadcast, in aggiunta ad altri parametri necessari per l'accesso al sistema;
- *interazione*: gli elementi del sistema devono seguire un modello comune per l'interazione;
- *service discovery*: consente la localizzazione di un servizio del sistema che coincide con le esigenze del dispositivo. Si trovano diverse realizzazioni di service discovery che utilizzano e forniscono una sintassi e

un vocabolario propri per specificare i servizi: l'implementazione può sfruttare strumenti quali l'allocazione di indirizzo e la risoluzione dei nomi. In genere il service discovery e l'interazione sono separabili.

2.1.1.4 La robustezza

Lo scopo principale dei sistemi ubiqui è quello di operare in modo efficiente con gli umani mettendo tutto ciò che la tecnologia offre al loro servizio.

Mentre in un mondo ideale si possono trascurare i ritardi, nel mondo reale risulta obbligatorio tenerne conto in quanto il delay di una trasmissione non è mai nullo. Poiché il mondo reale è in continuo cambiamento quando una comunicazione viene interrotta o ritardata, in fase di progettazione di un sistema ubiquo è necessario valutare nel dettaglio i ritardi, quindi la loro localizzazione e il peso che hanno nel sistema, e conoscere i limiti della percezione e della reazione umana. Alcuni esempi di limiti di Round Trip Time (RTT) accettabili dall'uomo sono:

- $<$ 350 ms per la telefonia;
- 200 ms per le teleoperazioni con feedback forzato;
- tra 20 e 40 ms per suonare musica simultaneamente.

È da tener sempre conto, inoltre, dei limiti fisici a cui sono soggette le trasmissioni digitali. Alcuni esempi possono essere forniti dalla fibra ottica la cui velocità di trasmissione è prossima ai 200000 km/s, ovvero circa 1 ms di ritardo ogni 200 km, oppure dall'aria in cui la velocità si avvicina a quella della luce, cioè $299792,458$ km/s, presentando un delay di 1 ms ogni 300 km circa. Durante la fase di analisi dei limiti del sistema, oltre a quelli fisici, è necessario prendere in considerazione anche quelli dovuti alla computazione e quelli alla conversione analogico-digitale dei segnali raccolti dai sensori. L'interruzione o il ritardo di un servizio può avere effetti negativi sulle macchine o sul loro ambiente e la stabilità di un processo può non essere assicurato se il delay supera una determinata soglia.

Con l'utilizzo delle reti wireless, i sistemi ubiqui vedono un sostanziale numero di failure rispetto a un sistema distribuito wired. La comunicazione wireless, infatti, risulta meno affidabile rispetto a quella su cavo in quanto è soggetta alla limitata copertura e alle interferenze delle strutture adiacenti al sistema.

2.1.1.5 La sicurezza

La sicurezza è un punto fondamentale per un qualsiasi sistema, pertanto non può essere escluso nella trattazione dei sistemi ubiqui. Grazie alla varietà di requisiti di sicurezza e al carattere dinamico di questi sistemi, viene

richiesta una protezione forte e flessibile. In precedenza è stato descritto, infatti, come nuovi componenti possano entrare nel sistema ed interagire tra di loro esplicitando una forte necessità in aspetti di sicurezza per le risorse e, in molti casi, di privacy per gli utenti. Il mobile computing ha permesso di capire e di analizzare numerose vulnerabilità come l'apertura delle reti wireless ma aspetti come l'integrazione e l'interazione fanno emergere la necessità di avere nuovi modelli di trust e di autenticazione. Si evidenzia come il trust sia una stretta necessità alla possibilità di spontanea interazione che i sistemi ubiqui offrono ai dispositivi. Si possono utilizzare strumenti come la crittografia tenendo sempre presente il fatto che l'integrazione fisica ha un impatto non irrilevante nei protocolli di sicurezza. Si pensi ad esempio ai dispositivi dotati di batteria per cui si vede necessario avere dei protocolli che minimizzano il carico computazionale e comunicativo al fine di preservare la batteria stessa.

Esistono diversi approcci per ottenere la flessibilità di sicurezza ed automatizzare la riconfigurazione dei meccanismi di protezione a livello di sistema. In questo modo si riesce ad ottenere una difesa senza il diretto intervento degli utenti. In letteratura si trovano diversi esempi: tra questi si citano, a titolo informativo, il paradigma software basato sui componenti per realizzare questi tipi di sistemi e applicati in contesti come la sorveglianza militare [8] oppure il tentativo di mettere in atto una consapevolezza di sicurezza in presenza di grandi moli di dati multimediali [9]. È necessario poi garantire la sicurezza nella gestione dei dati e nell'affidabilità delle comunicazioni in reti quali Internet applicando ad esempio delle metodologie che prevedono strategie di prevenzione e di predizione, in quanto l'interruzione delle comunicazioni o il presentarsi di failure di vario tipo sono scenari molto frequenti come è stato presentato nella sezione 2.1.1.4. Un altro aspetto molto importante da gestire riguarda l'autenticazione degli utenti: il controllo d'accesso risulta essere un requisito necessario in sistemi in cui le persone e i dispositivi possono entrare o uscire liberamente dall'ambiente ed interagire tra loro in modo spontaneo.

2.1.2 Un framework generico

Per favorire la diffusione e l'utilizzo di questi sistemi è necessario avere a disposizione dei framework generici utilizzabili in ambiti completamente diversi. Framework per sistemi ubiqui esistono già in letteratura, anche se dipendono fortemente dalle specifiche richieste e dalle funzionalità operative dei singoli ambiti d'utilizzo, limitando di fatto la loro generalità e la riutilizzabilità. È questo il caso, ad esempio, della robotica in cui si trovano diversi tentativi di fornire dei framework che coprono gran parte delle categorie del settore. Si trovano esempi anche di framework per software realtime che hanno come obiettivo il miglioramento della scalabilità e della riutilizzabilità dei moduli software come anche altri nell'ambito di applicazioni di telecon-

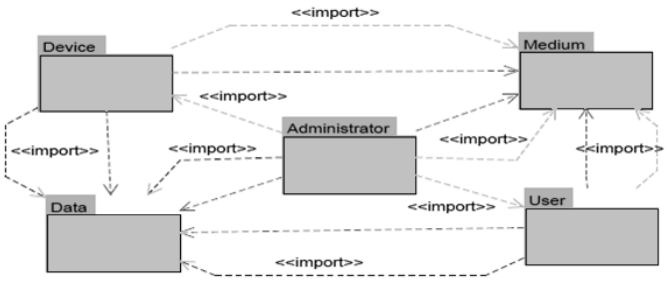


Fig. 2.2: Il framework generico preso in esame

trollo che si basano sulla telemedicina e sul controllo di sicurezza in contesto casalingo.

Essendo i sistemi ubiqui protagonisti di una rapida crescita in cui ci sono dispositivi e componenti di costruttori diversi e con architetture diverse, per garantire l'interoperabilità tra queste parti è necessario quindi avere un framework generico di supporto. Un tale framework fornisce una maggiore riutilizzabilità che deriva dall'indipendenza del settore di utilizzo.

Per offrire al lettore una migliore comprensione, si illustra ora il framework generico proposto in [7] che è stato utilizzato come punto di riferimento per la progettazione dell'infrastruttura oggetto di questa tesi. Il framework adottato a supporto dei sistemi ubiqui può esser visto come l'insieme di cinque package fondamentali collegati tra loro da particolari relazioni di dipendenza, come presentato in figura 2.2. I pacchetti di riferimento, che in seguito verranno presentati singolarmente, sono:

- Administrator;
- Data;
- Device;
- Medium;
- User.

Tutte le figure che modellano i vari package sono tratte da [7].

Il package Device Il pacchetto preso in esame rappresenta i dispositivi e identifica le loro proprietà, il loro comportamento e i vincoli definiti dalle specifiche dei costruttori. Si possono individuare, inoltre, altri elementi importanti per questo tipo di oggetto che vanno a specificare altri concetti per i dispositivi nell'ambito dei sistemi ubiqui. Uno di questi è il contesto che rappresenta informazioni quali la localizzazione ed il profilo mentre il medium connection garantisce la connettività al sistema. È possibile specificare, inoltre, il tipo di dispositivo e, mediante l'attributo Behaviour, i suoi

comportamenti specifici. La comunicazione e la collaborazione con il sistema ubiquo vengono gestite dalla componente che in figura 2.3 viene chiamata DCManager e si occupa della gestione delle connessioni. Un dispositivo, inoltre, offre la possibilità di fornire dei servizi, grazie all'elemento Provided-Service, e di scambiare delle informazioni e dei dati attraverso un'interfaccia utente, nel caso in cui ci si trovi in interazioni diverse dalla M2M.

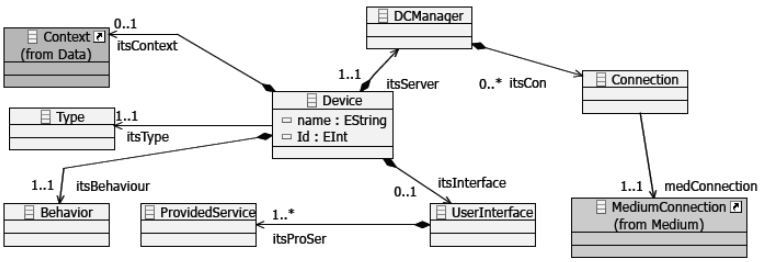


Fig. 2.3: Diagramma del package Device

Il package Medium L'ambiente di comunicazione reagisce sotto alcune condizioni o influenze. Si devono poter utilizzare, quindi, in un sistema ubiquo, infrastrutture già esistenti, mantenerle e specificarne i servizi. Gli ubiquitous system utilizzano, come communication medium, diverse reti esistenti, come Internet LAN e WAN, Bluetooth e WLAN. Le specifiche del mezzo impiegato sono molto importanti e non vanno pertanto trascurate in fase di progettazione ed implementazione delle strategie dei dispositivi e degli utenti. Al fine di generalizzare al meglio il concetto di medium, il pacchetto prevede altri due elementi, oltre alla gestione della connessione che permette di raccogliere dispositivi, utenti e amministratore del sistema ubiquo, a seconda che il mezzo utilizzato sia noto o meno. Mentre nel primo caso, infatti, esiste un attributo che conserva informazioni come protocolli e tipologia di rete in uso, nell'altro si ha un elemento definito come una black box.

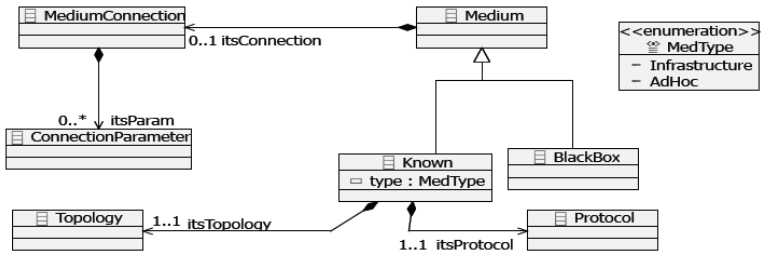


Fig. 2.4: Diagramma del package Medium

Il package User Questo pacchetto offre all'utente i comandi e le funzionalità di controllo che ha a disposizione, specificando tutte le procedure necessarie per interagire con le altre entità del sistema ubiquo. Com'è possibile osservare dalla figura 2.5, l'utente è collegato a un contesto che può comprendere dati, strategie ed informazioni di elaborazione e presenta elementi di comunicazione e di servizi analoghi a quelli del package Device, con la precisazione però che il loro utilizzo può essere diverso a seconda dell'utilizzo e dell'ambiente.

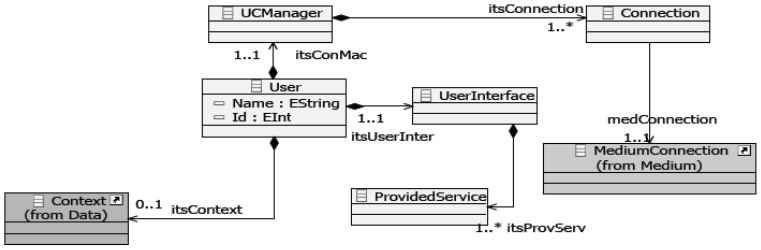


Fig. 2.5: Diagramma del package User

Il package Administrator I ruoli dell'amministratore includono le configurazioni della rete, la sua gestione e il controllo di scambio dei dati tra tutte le entità coinvolte nel sistema ubiquo. È un particolare utente, quindi, che ha la possibilità di gestire l'intero sistema grazie all'entità NetworkManager. A supporto, inoltre, è presente una admin interface che consente l'attuazione di tutte le operazioni a lui consentite.

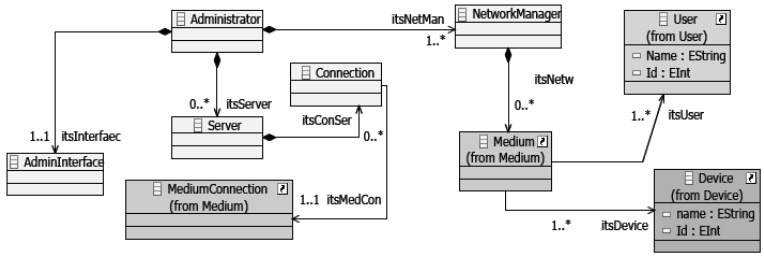


Fig. 2.6: Diagramma del package Administrator

Il package Data Questo package ha una certa rilevanza in quanto i dati rappresentano tutte le informazioni che circolano sulla rete. Questo pacchetto include i dati e tutte le relative tecniche di elaborazione, dal controllo alla protezione. È da osservare come anche l'aspetto sicurezza sia incluso essendo uno dei principali requisiti in un qualsiasi sistema. Nel pacchetto, poi, si trovano elementi quali il processing information, il context ed il trace,

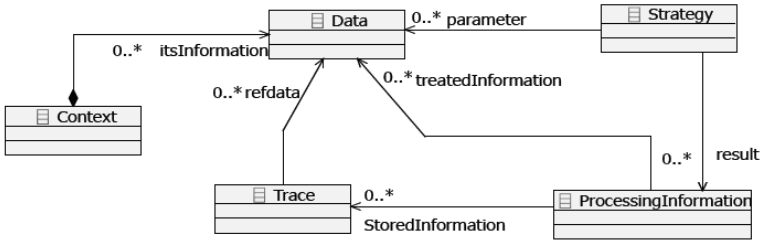


Fig. 2.7: Diagramma del package Data

che identificano rispettivamente le informazioni di elaborazione, il contesto dei dati e lo storico delle informazioni.

2.1.3 Esempi di sistemi ubiqui

Uno degli obiettivi dell'ubiquitous computing è di rendere i computer invisibili, perciò le persone interagiranno, in un prossimo futuro, con dispositivi o oggetti quotidiani in modo del tutto naturale, senza riconoscere la presenza di computer e senza un significativo sforzo cognitivo. Di conseguenza, saranno utilizzate sempre più interfacce per i diversi scenari dei sistemi ubiqui. Gli esempi applicativi di questi sistemi sono quindi innumerevoli e in seguito ne verranno presentati alcuni che riguardano l'interazione con applicazioni attraverso grandi schermi, come televisori, specchi o pareti intere, che diventano sempre più strumenti interattivi e condivisibili tra più utenti simultaneamente.

Lo specchio intelligente Una possibile applicazione del concetto di ubiquità si ha nello specchio-display. In un contesto familiare, una persona si può specchiare in casa e, dopo esser stata riconosciuta dal sistema, le si possono presentare delle specifiche informazioni, come notizie, appuntamenti e l'agenda delle mansioni da svolgere. Lo specchio può diventare anche uno strumento interattivo nel momento in cui offre la possibilità alla persona, ad esempio, di selezionare alcuni tra i to-do del giorno e di inviare i dati al proprio smartphone.

Riunioni interattive I dispositivi dotati di una fotocamera o di una videocamera sono quelli che meglio si prestano all'interazione con i display di grandi dimensioni. Come esempio si presenta uno scenario che coinvolge task di condivisione e di scambio di informazioni presentati in uno schermo grande posto a distanza da diversi partecipanti. Si pensi ad un team aziendale riunito nella sala riunioni per discutere sulle strategie da intraprendere e sull'assegnamento dei compiti. Ciascun membro del gruppo punta il telefonino verso lo schermo di grandi dimensioni e inserisce le proprie idee in modo ordinato a video. Una

Fig. 2.8: Scenario di riunione interattiva

volta che il manager ha selezionato le informazioni, le ha riordinate e le ha elaborate utilizzando come dispositivo di interazione sempre il cellulare, premendo un tasto nel telefono salva il documento permettendo così agli altri membri del team di ottenere le informazioni aggiornate e di visualizzarle attraverso il display del proprio cellulare. Lo scenario è rappresentato in figura 2.8, tratta da [6].

Lezioni universitarie L'esempio precedente può essere trasferito anche in ambito universitario: nel momento in cui il docente avvia la presentazione con le slide della lezione, infatti, tutti gli studenti del corso che si trovano in aula possono utilizzare il file nel proprio dispositivo con la possibilità di annotare direttamente appunti o osservazioni in locale.

2.2 Sistemi multiagente

Il concetto di ubiquità spiegato nella prima parte del capitolo può essere realizzato sfruttando diverse architetture a seconda delle esigenze del sistema. In questa parte vengono presentate le caratteristiche dei sistemi agent-based in quanto sono stati utilizzati nell'implementazione del progetto di tesi.

2.2.1 Analisi dell'architettura

Questo lavoro ha previsto un'analisi preventiva dell'architettura da utilizzare con il fine di individuare il modello che meglio rispecchiasse le specifiche progettuali. La scelta è ricaduta nel modello multiagente che sfrutta le proprietà dell'architettura *Peer-to-Peer* (P2P) in quanto offrono determinate possibilità che altre strutture, come il classico modello *Client-Server*

(C/S), non mettono a disposizione. Nel corso della sezione verranno presentate brevemente le architetture C/S e P2P, spostando poi l'attenzione sul paradigma ad agenti. Di questo verranno analizzati gli aspetti fondamentali come la mobilità e la comunicazione oltre che i vantaggi e gli svantaggi nell'utilizzo di tale tecnologia.

2.2.1.1 Architettura Client-Server

Il modello client-server è ampiamente utilizzato nelle applicazioni distribuite ed è basato su una distinzione di ruoli che va a identificare due gruppi, uno a cui appartengono i server e l'altro a cui fanno parte i client. Un *server* è un processo che implementa un particolare servizio, come ad esempio un servizio di file system, mentre un *client* è un processo che richiede un servizio a un server. L'interazione che c'è tra client e server è nota come *request-reply* in quanto il client, vero protagonista dell'iniziativa dell'intero sistema, invia la richiesta di uno specifico servizio al server rimanendo poi in attesa di una risposta. I server sono totalmente reattivi e per questo non sono in grado di prendere alcuna iniziativa: possono solamente rimanere in attesa di essere invocati dai client. Uno schema dell'interazione è rappresentato in figura 2.9, tratta da [1].

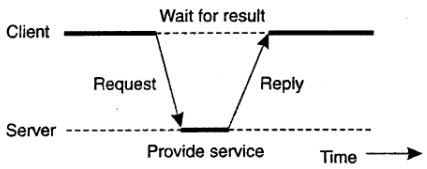


Fig. 2.9: Interazione request-reply

In un ambiente client-server, viene eseguito un programma client che abilita l'utente a spedire una richiesta al server, formattando adeguatamente il messaggio in modo che il server stesso sia in grado di processarlo. Il server rimane in attesa di richieste e non appena ne riceve una, la elabora inviando in seguito il risultato al client. Affinché l'interazione tra il client e il server possa essere effettuata, risulta necessario utilizzare un linguaggio comune ad entrambi, ovvero un protocollo applicativo. In questa architettura, inoltre, si può riscontrare come i client godano di totale libertà per entrare ed uscire dal sistema senza andare ad intaccare il corretto funzionamento dell'infrastruttura, comportamento che però non può essere attuato dai server.

Le applicazioni che si basano sul modello client-server sono davvero molte, come ad esempio il web, anche se per un notevole numero di esse questa architettura non si adatta perfettamente. È questo il caso di applicazioni come chat, sistemi distribuiti di scambio dati e giochi multiplayer: nonostante risulti possibile un'implementazione fedele al modello client-server, l'utilizzo di altre tipologie di architettura consente di sfruttare al meglio quanto richiesto da questo tipo di applicazioni, ovvero la necessità

di avere nodi attivi su terminali utenti capaci di comunicare l'un l'altro. Il modello meglio indicato risulta essere il *Peer-to-Peer*.

2.2.1.2 Architettura Peer-to-Peer

L'architettura P2P identifica un modello nel quale ciascun nodo, chiamato *peer*, svolge il compito sia del client che del server andando a eliminare quella distinzione di ruoli che invece caratterizzava il modello client-server. Il cuore dell'applicazione non risulta essere più tutto concentrato nei server ma è distribuito tra tutti i peer della rete: ogni nodo ha la possibilità di scoprirne altri, può entrare ed uscire liberamente dal sistema in qualsiasi punto e in qualsiasi istante. Il sistema risulta essere, pertanto, completamente distribuito poiché i servizi sono sparsi nell'intera rete. La differenza sostanziale tra questa architettura e quella client-server si riscontra particolarmente nel modo in cui i nodi possono essere scoperti. Mentre nei sistemi client-server i client devono conoscere i propri server ma non necessitano di conoscere gli altri client, nei sistemi P2P devono avere a disposizione i servizi propri del sistema che consentono ai nodi di entrare e di uscire dalla rete e di cercare altri peer in qualunque momento. Questi servizi sono tipicamente dei meccanismi noti come pagine bianche e pagine gialle (*white and yellow page mechanisms*) che consentono la pubblicazione e la scoperta di caratteristiche e di servizi offerti da un nodo. A seconda della modalità realizzativa di questi meccanismi, si possono identificare due diversi modelli basati sul P2P: si tratta di reti P2P pure o decentralizzate e di reti P2P ibride. L'architettura P2P pura ha la caratteristica di avere tutti i nodi completamente autonomi e paritari. L'assenza di un qualsiasi nodo specializzato rende, però, più difficile il mantenimento della coerenza della rete e della scoperta degli altri peer con un aumento di banda e di complessità che tende all'esponenziale al crescere dei nodi. Un'architettura ibrida, invece, è basata su un nodo specializzato che fornisce un servizio che semplifica la ricerca dei nodi attivi, dei loro servizi e delle loro risorse. Diversamente dal modello P2P puro, queste reti utilizzano meno banda in quanto viene generato molto meno traffico. Di contro, però, centralizzare informazioni sensibili, come può essere la lista dei nodi attivi, può essere rischioso, essendo questa soluzione vulnerabile a failure e ad attacchi.

2.2.2 Paradigma ad agenti

Il paradigma ad agenti è relativamente nuovo tra i paradigmi software e applica concetti che derivano dalla teoria dell'intelligenza artificiale nell'ambito dei sistemi distribuiti. L'*Agent-Oriented Programming* (AOP) modella un'applicazione in una collezione di componenti chiamati agenti che possono cooperare tra loro per un goal comune oppure competere tra loro per la riuscita dei propri obiettivi. Il modello architetturale di un'applicazione orientata agli agenti è intrinsecamente P2P poiché ciascun agente può ini-

ziare una comunicazione con un qualsiasi altro agente oppure essere soggetto a comunicazioni in ingresso in qualsiasi istante. Gli agenti sono considerati uno dei più importanti paradigmi software in quanto possono migliorare gli attuali metodi di concettualizzazione, progettazione ed implementazione di sistemi ed essere una soluzione al problema di integrazione. La tecnologia ad agenti è stata infatti materia di intense discussioni e ricerche da parte della comunità scientifica per diversi anni ma solamente di recente si sono visti risultati significativi grazie al suo utilizzo sempre maggiore nelle applicazioni commerciali. Gli ambiti in cui sono utilizzati i sistemi multiagente sono tra i più vari: dai sistemi di controllo al settore manifatturiero, dalla logistica dei trasporti al network management.

Nonostante l'ampio utilizzo di agenti software, in letteratura non si trova una definizione univoca per il termine *agente*. Tutte le definizioni, però, giungono a un punto d'incontro affermando che è uno speciale componente software con una propria autonomia che fornisce un'interfaccia interoperabile a un sistema arbitrario e/o si comporta come un agente umano svolgendo delle mansioni per altri agenti. Il sistema è solitamente costituito da più di un agente anche se nulla vieta di averne uno solo che opera nell'intero ambiente. I *sistemi multiagente* (MAS) possono modellare sistemi complessi dando la possibilità di raggiungere una cooperazione, in cui gli agenti hanno dei goal comuni, ma anche una competizione, dove ciascun componente mira al raggiungimento dei propri obiettivi. L'interazione tra gli agenti può avvenire in modo indiretto operando sull'ambiente oppure in maniera diretta attraverso una comunicazione e una negoziazione.

Da quanto detto in precedenza, è possibile delineare alcune caratteristiche tipiche di un agente. Tale componente è:

- *autonomo*: può portare a termine delle mansioni anche complesse e a lungo termine, ha un certo grado di controllo delle proprie azioni e, sotto alcune circostanze, può anche prendere delle decisioni;
- *proattivo*: oltre ad essere *reattivo*, quindi percepire l'ambiente circostante e reagire tempestivamente ai cambiamenti che si presentano in esso, ha anche un comportamento che mira al raggiungimento dei propri obiettivi con la capacità di prendere l'iniziativa, eseguendo determinati task anche in assenza di esplicite sollecitazioni da parte dell'utente;
- *sociale*: ha la capacità, che talvolta si esprime come una necessità, di interagire con gli altri agenti al fine di eseguire dei task per il conseguimento di goal del sistema;
- *mobile*: ha la possibilità di spostarsi tra diversi nodi della rete;

Alle caratteristiche sopra presentate si possono aggiungere altre qualità come ad esempio la *predisposizione all'apprendimento*, essendo capace di

adattarsi per conformare il proprio comportamento alla volontà dell'utente o del contesto. Un agente, inoltre, può essere definito anche *sincero*, in quanto fornisce la certezza che non comunica deliberatamente false informazioni, *benevolo*, tentando di eseguire sempre ciò che gli viene chiesto di fare, e *razionale*, agendo sempre per completare i propri task e non perchè si realizzi il contrario. A questo, poi, va aggiunta l'*abilità nella comunicazione* che consente l'interazione diretta con altre entità con lo scopo di raggiungere gli obiettivi personali ed altrui.

2.2.2.1 La mobilità

Gli agenti utilizzati per la realizzazione dell'infrastruttura oggetto di questa tesi sono mobili, ovvero sono entità che, come già anticipato nella sezione precedente, possiedono la capacità di spostarsi da un nodo ad un altro di una rete di calcolatori. Poiché la mobilità copre un ruolo importante all'interno delle scelte progettuali dell'infrastruttura che verrà presentata in seguito, si ritiene opportuno proporre un breve approfondimento in modo da facilitare la comprensione al lettore.

Il termine *mobilità* è utilizzato per indicare un cambiamento della locazione messo in atto dalle entità del sistema. Nell'ambito di questa tesi vengono utilizzati gli agenti mobili, entità autonome in grado di spostarsi tra i nodi in modo completamente indipendente dalla rete per trovare risorse e servizi a loro necessari. Solitamente, un agente mobile decide in quale nodo spostarsi mediante, ad esempio, una chiamata a un metodo del tipo `go(newLocation)`. Prima che maturasse l'idea degli agenti mobili esisteva la possibilità di inviare delle parti di programmi ad altre macchine e di eseguirle lì. Si parla infatti di *remote evaluation* (RE) se è colui che invia il programma a prendere l'iniziativa, mentre, nel caso in cui fosse il ricevente a richiederlo, si definisce il *code on demand* (COD). Mentre i paradigmi RE e COD appena presentati si basano sulla mobilità del solo codice, per il paradigma ad agenti mobili è necessario tenere in considerazione un ulteriore elemento fondamentale. Si tratta dello *stato* dell'agente che a sua volta può essere suddiviso in *data state* e in *execution state*, rispettivamente lo stato dei dati, come ad esempio le variabili interne, e quello dell'esecuzione che contiene informazioni quali lo stack e il program counter. È importante, a questo punto, fare una precisazione. Un agente mobile è caratterizzato da una serie di istruzioni che vengono interpretate ed elaborate da un calcolatore. Questo elenco di istruzioni, noto come *programma*, viene generalmente memorizzato nella memoria secondaria. In fase di esecuzione del programma, viene creato un *processo* che viene memorizzato nella memoria volatile con l'insieme delle informazioni su cui esso sta operando, lo *stato* del processo.

Non appena si parla di mobilità, quindi, sorge subito l'interrogativo sulle modalità di gestione dello stato durante la migrazione del processo. La

capacità dell'agente di trasferire il proprio stato durante il movimento da un nodo ad un altro della rete consente, infatti, di classificare i vari tipi di mobilità di cui l'agente stesso può essere dotato. I tipi di mobilità sono due e sono:

- *mobilità debole* o weak mobility: un agente dotato di questo tipo di mobilità è in grado di portare con sé solo il codice del suo programma e lo stato dei dati;
- *mobilità forte* o strong mobility: un agente che possiede questo tipo di mobilità ha la capacità di trasferire l'intero stato, sia quello dei dati che quello di esecuzione.

La differenza di mobilità appena presentata risulta di particolare interesse soprattutto in fase di arrivo dell'agente nella nuova destinazione in quanto, a seconda del tipo di mobilità, un agente può sfruttare le informazioni che possiede. Pensando a un agente con mobilità debole, l'unica possibilità, una volta arrivato nel nuovo nodo, risulta essere infatti quella della riesecuzione totale del suo codice. Se invece si è in presenza di un agente che ha trasportato l'intero stato si può pensare a una semplice riattivazione delle mansioni che stava svolgendo prima della migrazione. La mobilità di un agente può essere generalmente sintetizzata nei seguenti step:

1. Il flusso di esecuzione viene interrotto.
2. Viene fatta una stampa dello stato dell'entità che sta per migrare.
3. Il codice e lo stato dell'entità vengono trasferiti al nodo destinazione.
4. Il codice e lo stato vengono salvati e ripristinati.
5. Il flusso di esecuzione viene fatto ripartire.

2.2.2.2 La comunicazione

La comunicazione è una delle componenti chiave per un sistema basato sugli agenti poichè quest'ultimi devono esser in grado di poter comunicare con gli utenti, con le risorse di sistema e tra di loro al fine di raggiungere la cooperazione, la collaborazione e la negoziazione. Nei sistemi multiagente si possono distinguere le seguenti tre caratteristiche:

- *gli agenti sono entità attive*: la comunicazione tra gli agenti, basata su messaggi, è asincrona e va a sostituire le *remote procedure call*. Un agente, quindi, che vuole comunicare deve solamente inviare un messaggio a una certa destinazione; sarà poi il destinatario a decidere, secondo le proprie specifiche, quali messaggi utilizzare e quali scartare, quali servire prima e quali in seguito. Questo modo di comunicare

elimina qualsiasi dipendenza temporale tra mittente e destinatario risolvendo situazioni bloccanti quale, ad esempio, l'indisponibilità del ricevente al momento dell'invio.

- *gli agenti svolgono delle azioni e la comunicazione è solo un tipo di azione*: avere la comunicazione allo stesso livello delle azioni da svolgere consente all'agente di valutare un piano che include sia azioni fisiche che azioni comunicative. In questo scenario è necessario, per poter pianificare la comunicazione, definire in modo chiaro e completo effetti e precondizioni di ogni possibile atto comunicativo.

- *la comunicazione trasporta un significato semantico*: quando un agente è l'oggetto di un'azione comunicativa, come ad esempio quando avviene la ricezione di un messaggio, deve essere in grado di capire il significato di tale azione e, in particolare, perchè questa deve essere eseguita. Per far questo è necessario avere a disposizione una semantica universale e uno standard da seguire.

Ricollegandosi in modo particolare a quest'ultimo punto, vengono utilizzati linguaggi di comunicazione particolari, detti *agent communication language*, che poggiano in modo diretto sulla teoria degli atti linguistici (John Searle, 1969), la quale fornisce una separazione tra l'atto comunicativo e il linguaggio del contenuto.

2.2.3 Lo standard FIPA

All'aumento del consenso e dell'utilizzo delle piattaforme ad agenti mobili, cresce parallelamente anche il problema dell'interoperabilità tra di esse e tra gli agenti che le popolano. Si è fatto sempre più pressante, quindi, la necessità di avere degli standard che consentissero una compatibilità multipiattaforma. Nei primi anni di sviluppo degli agenti software, infatti, ciascuna piattaforma implementava degli specifici linguaggi e semantiche per la comunicazione e la collaborazione tra i propri agenti rendendole di fatto incompatibili con le altre piattaforme. Sono state elaborate alcune proposte di standard per la comunicazione tra agenti e tra i primi linguaggi di comunicazione standardizzati si segnala il *Knowledge Query and Manipulation Language* (KQML): è un linguaggio e un protocollo per lo scambio di informazioni e conoscenza. Il KQML [4], che risale ai primi anni '90, fa parte dell'ARPA Knowledge Sharing Effort che mira allo sviluppo di tecniche e metodologie per costruire basi di conoscenza su larga scala condivisibili e riutilizzabili.

Attualmente per l'interoperabilità tra agenti viene utilizzato lo standard FIPA elaborato dalla omonima organizzazione di standard, la *Foundation for Intelligent Physical Agents* [26]. Il consorzio, che ha sede in Svizzera, è nato nel 1996 e ad esso fanno parte numerose multinazionali che operano

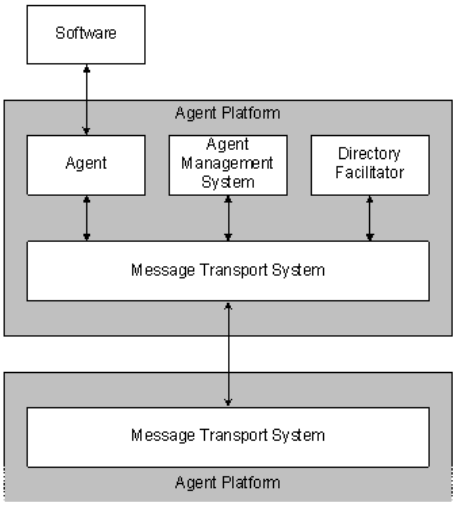


Fig. 2.10: Modello del FIPA Agent Management

nel campo dell'informatica e dell'elettronica. Nel giugno del 2005, inoltre, è stato riconosciuto e accettato dall'IEEE come il suo undicesimo comitato per gli standard. FIPA promuove la tecnologia basata sugli agenti e le sue specifiche rappresentano una collezione di standard che intende promuovere l'interoperabilità tra agenti eterogenei e i servizi che essi possono rappresentare, delineando di fatto le leggi che consentono a una società di agenti di esistere, operare ed essere gestita. Lo standard FIPA coinvolge diversi ambiti: applications, abstract architecture, agent communication, agent management ed agent message transport. Tra questi particolare interesse cade nell'Agent Management e nell'Agent Communication che vengono presentati nelle prossime sezioni.

2.2.3.1 Agent Management

Questo ambito dello standard FIPA offre delle linee guida per la gestione e il controllo degli agenti e del loro comportamento, sia quando si trovano in una piattaforma che durante la migrazione tra una piattaforma ed un'altra. Gli elementi che compongono l'Agent Management sono raffigurati in figura 2.10 tratta da [27] e nel dettaglio sono:

- *Agent Platform* (AP): costituisce l'ambiente vero e proprio in cui agiscono gli agenti comprendendo i dispositivi, i sistemi operativi, i componenti FIPA per l'Agent Management sotto descritti, gli agenti stessi e l'eventuale software di supporto.

- *Agent*: ogni agente è un processo computazionale che risiede nell'AP e che deve necessariamente avere un identificativo univoco detto FIPA *Agent Identifier* (AID).

- *Directory Facilitator* (DF): questo componente dell'AP offre un servizio di pagine gialle agli altri agenti mantenendo al suo interno un elenco aggiornato dei servizi offerti da ogni agente della piattaforma. Nell'AP ci possono essere più DF che possono funzionare da federazione. Un agente ha la possibilità di registrare, di deregistrare o di modificare un servizio comunicandolo al DF di interesse. Nel caso in cui un agente necessitasse uno specifico servizio ha la possibilità di interrogare il DF per conoscere il nome dell'agente che lo rende disponibile.

- *Agent Management System* (AMS): è un elemento obbligatorio per una piattaforma ed ha come compito il gestire le operazioni di un AP. Deve esserci necessariamente un solo AMS nella piattaforma: ciascun agente deve comunicare con l'AMS attendendo l'autorizzazione di poter operare all'interno della piattaforma mediante l'attribuzione di un AID. Nello specifico, l'AMS conserva un elenco degli AID assegnati comprese alcune informazioni accessorie utili alla gestione degli agenti, come ad esempio lo stato corrente. Viene offerto anche il servizio di pagine bianche che può essere utilizzato da un qualsiasi agente che si trova nella necessità di conoscere quali agenti sono presenti all'interno della piattaforma in un dato istante. Si segnala inoltre che la vita di un agente termina con la sua deregistrazione dall'AMS il quale provvede a cancellarne l'AID associato.

- *Message Transport Service* (MTS): è un servizio che viene fornito al fine di gestire e coordinare lo scambio di informazioni tra gli agenti di una o diverse AP. Le comunicazioni avvengono tramite l'utilizzo di messaggi che seguono una particolare struttura, come verrà presentato nella prossima sezione.

2.2.3.2 Agent Communication

La comunicazione è uno dei punti cardine di un sistema multiagente, rappresentandone il cuore del modello. Dallo standard FIPA emerge in particolar modo l'*Agent Communication Language* (ACL), un linguaggio di comunicazione tra agenti per lo scambio di informazioni. FIPA ACL prevede una libreria estensibile di diversi *communicative act* che permettono di rappresentare un elevato numero di intenzioni comunicative diverse. Fanno parte dei tanti communicative act a disposizione intenzioni quali REQUEST, PROPOSE, INFORM, QUERY, ACCEPT e REFUSE. La FIPA ha definito, inoltre, la struttura di un messaggio ACL che si può osservare in figura 2.11 tratta da [12]: i campi del mittente e dei destinatari sono accompagnati dal contenuto del messaggio, dalle sue proprietà, come la codifica e il linguaggio utilizzato, e da informazioni utili per identificare e seguire thread di conversazione in corso tra agenti. È anche possibile trovare la definizione

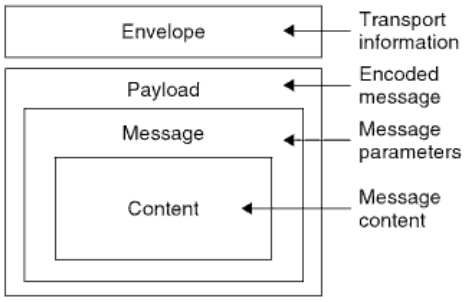


Fig. 2.11: La struttura di un messaggio FIPA

di protocolli di interazione che mettono a disposizione degli agenti una fornita libreria di schemi per raggiungere task comuni, come ad esempio la delega di un'azione.

2.2.4 Vantaggi e svantaggi del paradigma

Nelle sezioni precedenti sono emersi, in modo indiretto, vantaggi e svantaggi che si hanno utilizzando la tecnologia ad agenti mobili rispetto ad architetture quali il classico modello C/S.

Riassumendo i vantaggi, si può affermare come ci sia una notevole diminuzione del traffico di dati nella rete sfruttando la mobilità degli agenti. Grazie a questa proprietà, infatti, è possibile ridurre il numero di messaggi e di pacchetti dati nella rete in modo sensibile. Si pensi ad esempio a un sistema in cui è presente un dispositivo che possiede delle informazioni che periodicamente vengono aggiornate e altre entità che devono ricevere nel minor tempo possibile gli aggiornamenti. È possibile utilizzare in questo caso un agente che si muove dai dispositivi che si devono aggiornare alla macchina centrale e che provvede alla ricerca e all'analisi di aggiornamenti possibili. Finite queste mansioni, l'agente può ritornare ai client con tutte le informazioni necessarie riducendo il traffico all'interno della rete. A quanto appena presentato segue una conseguente riduzione della latenza all'interno della rete, un vantaggio questo di particolare interesse per le reti a limitata velocità di trasmissione. L'utilizzo degli agenti mobili consente di avere inoltre una maggiore tolleranza ai guasti. Nel caso di malfunzionamenti della rete, infatti, gli agenti possono operare anche se non è presente una connessione attiva tra i vari nodi: la loro sospensione avviene infatti solamente nel caso in cui dovessero trasmettere dei risultati oppure muoversi verso un altro nodo della rete. Al momento della riattivazione della connettività, l'esecuzione degli agenti sospesi può riprendere. Si può quindi concludere che, utilizzando il modello ad agenti, è sufficiente che la connessione sia attiva solo per brevi periodi di tempo.

Per quanto riguarda gli svantaggi, viene evidenziato in particolar modo il fatto che l'utilizzo degli agenti mobili porta a ridefinire e a rivalutare l'aspetto della sicurezza. La mobilità, infatti, introduce nuovi problemi legati ai dati non esistenti nell'architettura C/S. Garantire la riservatezza e l'integrità dei dati in possesso di un agente risultano essere punti fondamentali in applicazioni quali l'e-commerce o comunque in un qualsiasi contesto in cui circolano dati sensibili. Si evidenziano due ambiti in cui risulta di particolare interesse la salvaguardia dei dati: da un lato le informazioni trasmesse da un agente ad altre entità, siano esse altri agenti o piattaforme, dall'altro tutti i dati posseduti dall'agente stesso. La questione della sicurezza è sensibile quindi anche per i sistemi multiagente, risultando essere un problema ancora aperto, come descritto in [16].

2.2.5 Esempi di applicazioni agent-based

Come già esposto precedentemente, esistono numerose applicazioni che sfruttano l'architettura ad agenti. In questa sezione vengono presentati alcuni esempi che spaziano in diversi ambiti. Si fa presente, comunque, che la problematica sollevata nella sezione precedente riguardo la sicurezza deve essere gestita anche in queste applicazioni, soprattutto in presenza di dati sensibili.

Network Management A supporto della gestione di reti di grandi dimensioni può esser utile disporre di tool che consentano di effettuare da remoto operazioni di manutenzione e di monitoraggio dei vari nodi. Pensando, ad esempio, alla necessità quotidiana di verifica del funzionamento di tutti i componenti della rete, gli agenti mobili, grazie alla loro autonomia e alla tolleranza ai guasti, rappresentano uno strumento ideale per tali mansioni. Consentono, nell'esempio specifico, di effettuare i controlli in maniera del tutto automatizzata e di avere a disposizione una serie di azioni da intraprendere in caso di malfunzionamenti, come ad esempio l'invio di un report dettagliato riferito al nodo guasto.

E-commerce Un'attraente sfida per la tecnologia ad agenti mobili è data dall'*electronic commerce*. Gli agenti, infatti, si prestano bene al modello in cui ci sono in una rete clienti e rivenditori che scambiano prodotti, servizi e soldi. Considerando i vari nodi della rete come dei negozi virtuali, un agente compratore può viaggiare da un sito a un altro per conto di un utente che vuole comperare dei prodotti. Ci sono diverse modalità per rappresentare l'intelligenza di questi agenti. Un tale agente può, ad esempio, spostarsi tra i diversi siti che offrono uno stesso servizio, registrare ogni prezzo per fare una comparazione finale e poter prendere il più economico. Si possono pensare gli agenti mobili immersi anche in realtà quali ricerche e analisi di mercato: previa

l'autorizzazione da parte dell'utente, infatti, un agente potrebbe sottoporgli un questionario e modificare le domande da porre in funzione dello storico delle risposte degli altri utenti.

Information Retrieval L'utilizzo degli agenti mobili nell'ambito dell'Information Retrieval riprende in parte quanto descritto nel punto precedente. Un utente, ad esempio, che vuole rimanere continuamente aggiornato su determinati argomenti, ha la possibilità di utilizzare un agente che effettua, in modo del tutto automatizzato, la ricerca delle informazioni richieste secondo una particolare politica definita dall'utente stesso, come ad esempio una lista di siti internet.

Load Balancing La tecnologia ad agenti mobili può fornire anche una valida soluzione al problema della distribuzione del carico computazionale in un insieme di entità. Il problema fu inizialmente risolto sfruttando la migrazione dei processi, tipicamente sotto la supervisione di un controllore centralizzato. Grazie ai MAS è possibile decentralizzare la distribuzione del carico computazionale. Un'applicazione anche complessa può essere suddivisa in parti autonome tra di loro, ciascuna delle quali può essere delegata a un agente mobile che ha la responsabilità di cercare il nodo della rete più conveniente dove possa essere eseguito. Durante l'esecuzione, gli agenti possono muoversi tra i vari nodi alla ricerca di maggiore capacità computazionale disponibile con l'obiettivo di completare il proprio compito e di distribuire al meglio il carico di lavoro nella rete. Diversamente dai punti precedenti, quello del Load Balancing è un problema leggermente diverso infatti richiede che il movimento degli agenti risulti del tutto trasparente all'applicazione.

CAPITOLO 3

Le piattaforme tecnologiche

Indice

Nel presente capitolo vengono illustrate le tecnologie utilizzate per la realizzazione dell'infrastruttura che verrà presentata nel capitolo 4. La scelta per quanto riguarda la piattaforma ad agenti mobili è ricaduta su JADE alla quale è stato aggiunto un add-on specifico per consentire l'integrazione dell'ambiente mobile. Nel progetto di tesi, infatti, è stato utilizzato anche uno smartphone dotato di Android[1] sul quale dovevano essere eseguiti degli agenti. Si è reso necessario pertanto l'utilizzo di JADE-LEAP, con l'estensione per Android, che ottimizza JADE per l'esecuzione su dispositivi wireless con risorse limitate come telefoni cellulari o PDA.

[1] Android™ 2.1 (Éclair)

3.1 JADE

Telecom Italia e, più precisamente, CSELT (*Centro Studi E Laboratori Telecomunicazioni*), verso la fine del 1998, mossi dalla necessità di validare le prime specifiche FIPA, iniziarono i primi sviluppi software di ciò che divenne in seguito una piattaforma ampiamente utilizzata. Erano queste, infatti, le basi per il *Java Agent Development* framework (JADE), un middleware[2] implementato da Telecom Italia: nel 2000 JADE divenne open source e fu distribuito da Telecom sotto licenza LGPL (Lesser Gnu Public Licence), una tipologia di licenza che non pone alcuna restrizione al software che utilizza JADE. Per favorire il coinvolgimento industriale al progetto, nel maggio 2003 TILab, il centro di ricerca del Gruppo Telecom Italia, e Motorola Inc. definirono un accordo di collaborazione costituendo il JADE Governing Board, un'organizzazione no-profit con lo scopo di promuovere l'evoluzione della piattaforma JADE e il suo utilizzo come standard. Come si legge in [28], il JADE Governing Board è stato costituito in forma di Consortium Agreement tra tutti i membri consentendo, pertanto, l'ingresso a tutti coloro che abbiano come interesse l'utilizzo, lo sviluppo e la promozione della piattaforma.

Nel corso degli anni JADE ha sempre aderito allo standard FIPA e sono stati realizzati numerosi add-on a supporto, per consentire una sempre maggiore integrazione di altre realtà e tecnologie con la piattaforma. Nel corso del capitolo verranno presentate le estensioni di interesse per lo specifico lavoro di tesi.

Nel momento in cui viene scritto questo testo, l'ultima versione rilasciata della piattaforma risale al 7 luglio 2010: si tratta di JADE 4.0.1, versione che è stata utilizzata anche nello sviluppo del progetto.

JADE è una piattaforma software che fornisce delle funzionalità di base a livello middleware indipendenti dalla specifica applicazione e che facilitano la realizzazione di applicazioni distribuite che utilizzano il concetto di agente software. L'intelligenza, l'iniziativa, l'informazione, le risorse e il controllo possono essere completamente distribuiti su terminali mobili così come in computer fissi all'interno di una rete. Come già detto nella sezione 2.2.2 le applicazioni multiagente si basano su un'architettura di comunicazione P2P in cui l'ambiente può evolvere dinamicamente con i vari agenti che appaiono e scompaiono nel sistema a seconda dei bisogni e delle esigenze delle applicazioni stesse.

JADE è completamente sviluppato in Java e si basa sui seguenti principi guida:

- *interoperabilità*: JADE è compatibile con le specifiche FIPA citate nella sezione 2.2.3, di conseguenza gli agenti JADE possono operare

[2] Il termine *middleware* descrive una serie di librerie ad alto livello che forniscono dei servizi generici utili a una grande varietà di applicazioni

con altri agenti a condizione che questi siano compatibili con lo stesso standard.

- *uniformità e portabilità*: JADE fornisce un set omogeneo di API indipendente dalla rete sottostante e dalla versione di Java. JADE fornisce le stesse API per gli ambienti J2EE, J2SE e J2ME[3], come meglio verrà precisato nella sezione riguardante JADE-LEAP. Dal punto di vista teorico, quindi, gli sviluppatori potrebbero decidere l'ambiente Java al momento della distribuzione.

- *facile da usare*: la complessità del middleware è nascosta allo sviluppatore grazie al set di API semplici ed intuitive.

La versatilità di JADE consente la sua installazione anche sui telefoni mobili, a condizione che siano abilitati a Java.

3.1.1 Il modello architetturale

JADE include sia le librerie richieste per lo sviluppo di applicazioni ad agenti che l'ambiente runtime. Questo fornisce i servizi base che devono essere attivi sul dispositivo prima che gli agenti possano essere eseguiti. Ogni istanza del runtime JADE è detto *container*, così chiamato perchè si tratta di un'entità in grado di ospitare al suo interno gli agenti. L'insieme di tutti i container costituisce la *piattaforma* e fornisce un livello omogeneo che nasconde agli agenti e agli sviluppatori la complessità e la diversità degli strati inferiori, quali hardware, sistema operativo, tipi di rete e Java Virtual Machine (JVM). Tra i container ne è presente uno, detto *main container*, che ha la speciale funzione di controllare e coordinare il funzionamento degli altri contenitori, detti anche *periferici*, oltre che fungere da punto di riferimento. Tra il main container e la piattaforma JADE c'è una corrispondenza univoca, infatti se ne venisse avviato un secondo esso darebbe origine a un'ulteriore piattaforma a sé stante e indipendente dalla prima. All'interno del main container si trova obbligatoriamente l'AMS e almeno un DF che in JADE sono stati implementati come agenti software e che vengono istanziati automaticamente all'avvio della piattaforma garantendo in questo modo l'aderenza alle specifiche FIPA. Il main container mantiene anche altre informazioni necessarie per il corretto funzionamento della piattaforma: si parla ad esempio dell'*Agent Container Table* (ACT), una tabella che mantiene una mappa di tutti i contenitori presenti nella piattaforma, e dell'*Agent Global Descriptor Table* (AGDT) che invece mette in correlazione le informazioni presenti nell'AMS riguardo a un agente con il proprio AID e con altri dati utili alla sua gestione.

[3] J2EE, J2SE e J2ME per esteso sono rispettivamente Java Enterprise Edition, Java Standard Edition e Java Micro Edition

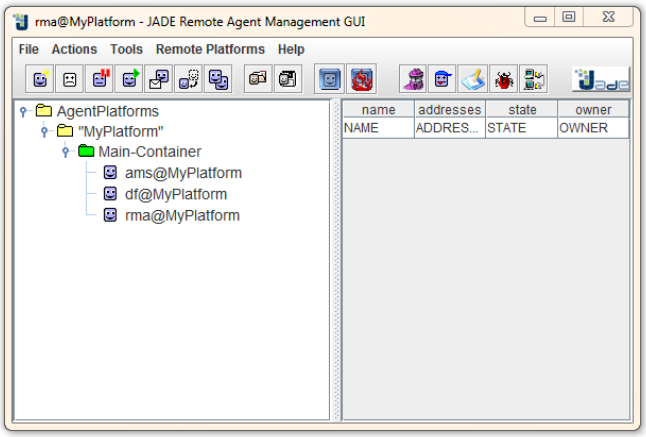


Fig. 3.1: L'interfaccia grafica

A supporto della piattaforma JADE vengono fornite anche delle opzioni quali una Graphical User Interface (GUI) che permette di monitorare lo stato delle componenti attive in un dato istante, siano essi agenti o container. Questa prende il nome *Remote Monitoring Agent* (RMA) che è stata implementata come agente: può essere eseguita all'avvio se richiesto con lo specifico comando[4]. L'RMA offre la possibilità di eseguire numerose operazioni utili alla gestione di una piattaforma JADE come avviare, terminare, sospendere, muovere o clonare un agente. JADE non offre solo questo particolare agente ma ne mette a disposizione anche altri che possono risultare molto utili. In particolare si tratta di:

- *Sniffer Agent*: consente di monitorare ed ispezionare i messaggi scambiati tra gli agenti;
- *Dummy Agent*: permette di fare dei test sul funzionamento dello scambio di messaggi tra l'agente stesso ed un altro agente;
- *Introspector Agent*: è uno strumento che consente di monitorare nella completezza un agente andando a visualizzare stato, messaggi processati e in lista sia in ingresso che in uscita e i behaviour;
- *Log Manager Agent*: permette di eseguire un logging approfondito su un dato container memorizzando tutti gli eventi che accadono in esso.

3.1.2 Il modello funzionale

Dal punto di vista funzionale, JADE fornisce i servizi base necessari alle applicazioni distribuite P2P in ambiente sia mobile che fisso. JADE consente

[4]Da terminale è necessario aggiungere l'opzione `-gui`

la scoperta dinamica degli agenti e la comunicazione tra di loro. Ciascun agente è identificato da un nome univoco e fornisce una serie di servizi che può registrare o modificare oppure può anche cercare degli agenti che offrono uno specifico servizio ad esso necessario.

La comunicazione Uno dei punti di forza del paradigma ad agenti è proprio la comunicazione. Gli agenti hanno la possibilità di comunicare tra loro inviando dei messaggi secondo le specifiche dettate in FIPA ACL, presentato nella sezione 2.2.3.2. Poiché la comunicazione può essere sia intrapiattaforma che interpiattaforma, vengono utilizzati protocolli diversi. Per la messaggistica interna a una stessa piattaforma, JADE utilizza un protocollo detto *Internal Message Transport Protocol* (IMTP): essendo impiegato esclusivamente in comunicazioni interne, è stato deciso di utilizzare un protocollo proprietario che non necessita di essere conforme alle specifiche FIPA, consentendo quindi di migliorare le performance della piattaforma. Questo protocollo non è stato pensato per trasportare solo i messaggi infatti permette anche di inviare comandi interni utili alla gestione della piattaforma e di monitorare lo stato dei container periferici. JADE esegue il routing dei messaggi sia in ingresso che in uscita utilizzando una single-hop routing table che richiede una diretta visibilità IP tra i container. Per quanto riguarda la comunicazione interpiattaforma, invece, JADE segue le specifiche FIPA utilizzando il protocollo *Internet Inter-Orb Protocol* (IIOP), un meccanismo che permette di parlare con altre piattaforme JADE o di altre tecnologie. Questo protocollo assicura la completa trasparenza nel trasferimento di messaggi anche quando sono coinvolte diverse piattaforme per agenti consentendo agli sviluppatori di non preoccuparsi dell'IIOP e della sua gestione.

Gli agenti JADE, come indicato dalle specifiche FIPA, sono caratterizzati da un nome univoco, l'AID, che li identifica all'interno del sistema evitando così al mittente di dover avere una referenza all'oggetto destinatario per inviare un messaggio. Tra di loro gli agenti comunicano attraverso lo scambio di messaggi asincroni, un modello di comunicazione molto utilizzato nell'ambito dei sistemi distribuiti. In questo modo viene tolta ogni dipendenza temporale tra gli agenti comunicanti: questo è una proprietà fondamentale in quanto mittente e destinatario potrebbero non essere disponibili allo stesso tempo. Ogni agente possiede una coda di messaggi, come una mailbox, dove il runtime JADE inserisce i messaggi inviati da altri agenti notificando l'evento ad ogni ricezione, com'è possibile osservare in figura 3.2 tratta da [12].

Il formato del messaggio in JADE segue quanto definito in FIPA ACL, pertanto ogni messaggio è costituito da:

- il mittente;
- una lista di destinatari;

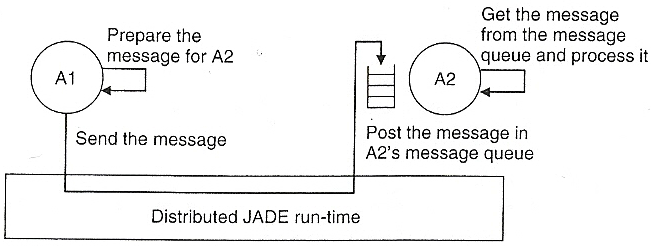


Fig. 3.2: Il paradigma di comunicazione asincrono in JADE

- l'atto comunicativo o *performative* che indica ciò che il mittente intende raggiungere con l'invio del messaggio, come per esempio l'invio di una richiesta (REQUEST), la comunicazione di un dato fatto (INFORM) oppure la volontà di aprire una negoziazione (PROPOSE o CFP[5]);

- il contenuto che consiste nell'effettiva informazione che il mittente invia ai destinatari e che completa il communicative act ad esempio indicando l'azione per cui è inviata una REQUEST;

- il linguaggio indicante la sintassi utilizzata per esprimere il contenuto del messaggio che deve esser nota al mittente e ai destinatari per una sua corretta comprensione ed elaborazione;

- l'ontologia che indica il vocabolario dei simboli utilizzati nel contenuto;

- altri campi addizionali quali il conversation-id, reply-with, in-reply-to e reply-by, strumenti utili per controllare diverse conversazioni concorrenti e per specificare i timeout per la ricezione di una risposta.

L'invio, nella pratica, si ottiene invocando il metodo `send()` mentre per quanto riguarda la ricezione è possibile utilizzare la versione non bloccante grazie al metodo `receive()` oppure quella bloccante utilizzando `blockingReceive()`. Quest'ultima opzione consente, infatti, di interrompere l'esecuzione di un agente fino all'arrivo di un nuovo messaggio, mettendo in attesa un qualsiasi comportamento in esecuzione sull'agente.

Sfruttando il fatto che ogni messaggio ha una propria semantica, è possibile specificare delle sequenze predefinite di messaggi che vanno a definire particolari protocolli di interazione. JADE infatti mette a disposizione dello sviluppatore un set di strutture di interazione per eseguire dei compiti specifici, come la negoziazione e la delega di task, togliendogli di fatto il peso della gestione della sincronizzazione, dei timeout e di tutti quegli aspetti che non sono legati alla logica dell'applicazione.

[5]Call For Proposals

3.1.2.1 I behaviour

Le azioni che un agente svolge vengono concretizzate in oggetti che definiscono dei comportamenti. Un *behaviour* rappresenta un task che un agente può eseguire e viene implementato come un oggetto di una classe che estende `jade.core.behaviours.Behaviour`. Un behaviour può essere aggiunto all'agente sia nella sua fase di inizializzazione, quindi nel metodo `setup()`, che in altri comportamenti mediante il metodo `addBehaviour()`. Un agente può eseguire diversi behaviour in modo concorrente: è importante sottolineare che lo scheduling in un agente è non pre-emptive ma cooperativo. Questo significa che quando un behaviour è schedulato per l'esecuzione viene invocato il suo metodo `action()` che rimane in esecuzione finché non ritorna. Seguendo la figura 3.3 tratta da [12], si osserva come un agente selezioni il behaviour successivo da eseguire da un pool di behaviour attivi e, al termine del metodo `action()`, viene verificato se il task è stato portato a termine con successo o meno attraverso l'invocazione di `done()`. Quando non c'è alcun behaviour disponibile per l'esecuzione l'agente viene sospeso al fine di non utilizzare inutilmente CPU; questo poi viene riattivato non appena si presenta un behaviour eseguibile.

Nel package `jade.core.behaviours` si trovano diverse estensioni della classe generica `Behaviour` dando così la possibilità allo sviluppatore di creare dei comportamenti ad hoc per l'agente, estendendo a loro volta tali classi.

Nello specifico si citano i tipi di behaviour che sono stati utilizzati nello sviluppo del progetto di tesi:

- *OneShotBehaviour*: questo comportamento ha la particolarità di completare la sua esecuzione in un'unica fase. Il metodo `action()` viene quindi eseguito una volta sola e il metodo `done()` restituisce sempre il valore `true`.

- *CyclicBehaviour*: in questo caso il metodo `done()`, diversamente dalla classe precedente, ritorna sempre il valore `false` quindi è stato pensato come un behaviour che non termina mai. Il metodo `action()` infatti viene chiamato ciclicamente eseguendo sempre le stesse operazioni ogni qualvolta lo specifico behaviour viene selezionato per l'esecuzione.

- *TickerBehaviour*: questo comportamento offre la possibilità di scheduling delle attività eseguendole a intervalli regolari di tempo. In fase di inizializzazione, infatti, deve essere indicato il periodo della schedulazione che viene poi preso in considerazione per chiamare il metodo `onTick()` con scadenze regolari. Le attività, quindi, devono essere inserite all'interno di questo metodo. Come per il CyclicBehaviour, il TickerBehaviour non termina mai la sua esecuzione se non per esplicita richiesta di rimozione o di interruzione grazie all'invocazione rispettivamente dei metodi `removeBehaviour()` della classe `Agent` e `stop()`.

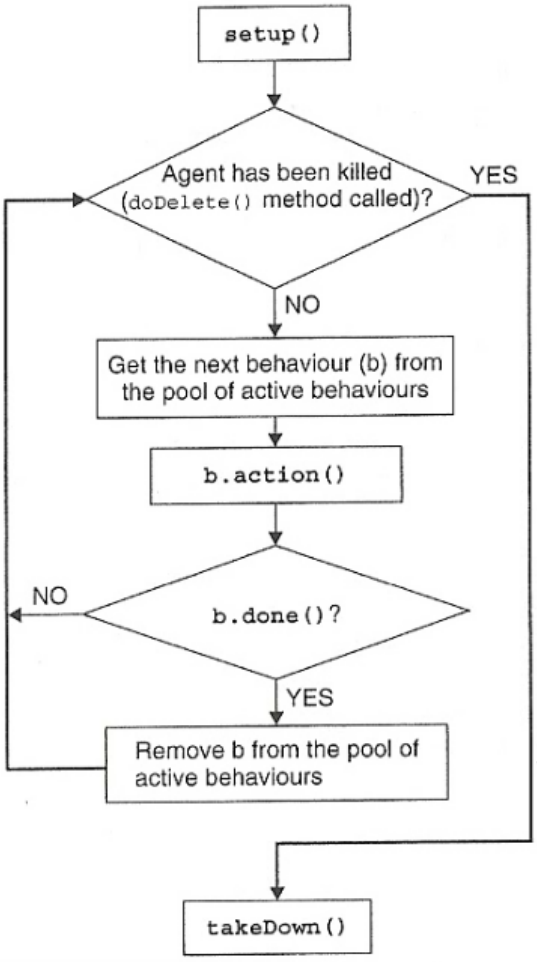


Fig. 3.3: L'esecuzione di un agente

- *WakerBehaviour*: come per il TickerBehaviour, questo comportamento consente di eseguire le operazioni contenute nel metodo `onWake()` dopo un dato timeout indicato in fase di istanziazione. Al termine del metodo il WakerBehaviour è completato.

- *Behaviour*: è il behaviour generico che consente una forte personalizzazione. Quando si utilizza questo tipo di comportamento è necessario implementare il metodo `action()` con le azioni da svolgere ed inserire una condizione di termine nel metodo `done()`.

Si sottolinea come i vari tipi di behaviour possano essere combinati assieme per generare dei task anche complessi.

3.1.2.2 La mobilità

Una delle più importanti caratteristiche di un sistema multiagente è la mobilità, pertanto viene garantita e implementata anche in JADE. Anche in questa situazione, come è stato illustrato in precedenza per la comunicazione, si presentano due casi: la mobilità intrapiattaforma e quella interpiattaforma. JADE implementa nativamente la prima, mentre la possibilità di muovere agenti tra piattaforme diverse viene fornita installando l'*Inter-Platform Mobility Service* (IPMS), un'estensione di JADE che cerca di rendere il più possibile trasparente la migrazione. Per lo sviluppo del progetto di tesi è stata utilizzata esclusivamente la mobilità intrapiattaforma. La gestione della migrazione di un agente da un container a un altro avviene mediante l'utilizzo di tre metodi:

- `doMove(newLocation)`: questo metodo consente di segnalare all'agente invocante di dover iniziare il trasferimento verso un altro contenitore, specificato nel parametro `newLocation`;

- `beforeMove()`: il metodo viene invocato non appena l'agente ha fatto una copia dello stato ed esattamente prima di creare una nuova istanza nel contenitore destinazione; risulta utile inserire tutte le azioni che l'agente deve compiere prima di terminare il thread originale come ad esempio il rilascio delle risorse utilizzate.

- `afterMove()`: questo metodo viene invocato non appena l'agente arriva nel nuovo contenitore ma prima che lo scheduler dei behaviour venga riattivato.

Oltre alla migrazione JADE offre anche il servizio della clonazione degli agenti che consiste nella semplice copia del codice sorgente dell'agente e nell'esecuzione di una nuova istanza applicando il nome passato come parametro al metodo specifico senza interessare però lo stato. I metodi di interesse, per la clonazione, sono del tutto analoghi a quelli della migrazione e sono `doClone(newLocation, newName)`, `beforeClone()` e `afterClone()`.

3.1.2.3 L'interfaccia grafica

JADE offre anche la possibilità ad ogni agente di avere una propria veste grafica per favorire l'interazione con l'utente e per agevolare il controllo da parte dello sviluppatore durante la fase realizzativa. Per creare la GUI di un agente si possono utilizzare gli strumenti forniti da Java mentre per l'interazione tra l'agente e l'interfaccia si sfrutta generalmente il passaggio degli eventi.

3.1.3 JADE in ambiente mobile: JADE-LEAP

Come già presentato precedentemente, JADE può essere eseguito su un'ampia gamma di dispositivi, dai computer ai cellulari. Gli sviluppatori hanno visto la necessità quindi di realizzare JADE-LEAP (*JADE - Lightweight Extensible Agent Platform*), uno dei primi add-on per JADE, che permette lo sviluppo di sistemi multiagente che seguono le specifiche FIPA per dispositivi mobili. Framework come LEAP sono molto importanti per una piattaforma quale JADE poiché le consentono di essere maggiormente accolta all'interno della società: in particolar modo, con l'introduzione di reti wireless sempre connesse come il GPRS, l'UMTS e le WLAN e con la sempre maggior crescita di potenza e di risorse dei dispositivi portatili, come PDA e smartphone, si è reso necessario integrare sempre più gli ambienti wired e wireless. È da sottolineare come, grazie a LEAP, sia possibile eseguire agenti JADE anche su framework Microsoft.NET alla versione 1.1 al momento della stesura di questo testo. LEAP è stato sviluppato nel 2002 da Motorola, ADAC, British Telecommunications, Broadcom, Siemens AG, Telecom Italia Lab e l'università di Parma.

Quando si implementano sistemi o framework per dispositivi mobili è sempre necessario prendere in considerazione tutti i limiti che questi portano per loro natura: nonostante infatti i risultati sempre migliori dei dispositivi portatili che portano alla riduzione notevole del gap con i computer desktop, si riscontrano dei limiti nella potenza di calcolo rispetto a quella dei computer da tavolo, nel numero minore di feature supportate dalla Java Virtual Machine dei dispositivi e dalla natura delle connessioni wireless utilizzate, come il GPRS o l'UMTS. Altri vincoli da tenere a mente sono posti dallo storage persistente dei dati che solitamente non presenta la disponibilità di file system e dalla limitata durata della batteria. Gli sviluppatori devono pertanto adattarsi a questi limiti gestendo al meglio la potenza di calcolo e la memoria, trattandole come risorse sempre più preziose per le applicazioni. Inoltre, essendo le interazioni container-container in JADE basate sul protocollo RMI, che non è molto efficiente in termini di byte trasmessi, reti come GPRS e UMTS non sono particolarmente indicate. Il progetto LEAP è nato raccogliendo tutte le considerazioni sopra citate con l'obiettivo primario di creare un middleware sufficientemente leggero per essere eseguito su dispositivi a limitate capacità come un cellulare. Il framework è stato progettato per supportare la scalabilità infatti JADE-LEAP può essere modellato in tre diversi modi che corrispondono ai tre tipi principali di ambienti Java (edizioni, configurazioni e profili):

- *J2SE*: per computer desktop e server in reti fisse;
- *pJava*: per dispositivi portatili che supportano J2ME CDC o Personal Java;

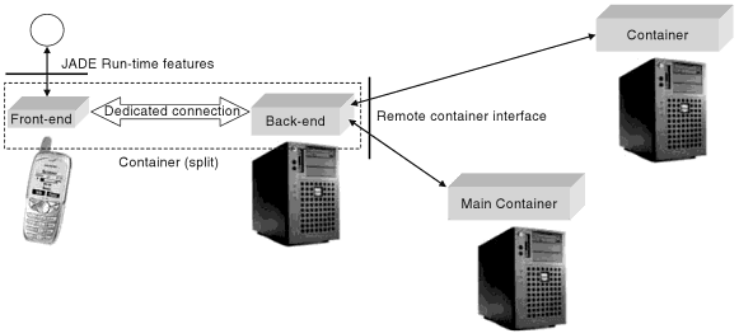


Fig. 3.4: Lo split execution mode in LEAP

- *MIDP*: per dispotivi portatili che supportano MIDP1.0 o successivi. Fanno parte di questa categoria la maggior parte dei cellulari che supportano Java.

Nonostante le diverse realizzazioni interne, le tre versioni di JADE-LEAP offrono un sottoinsieme di API che va a colmare le diversità dei dispositivi e dei tipi di rete. Si riesce così ad ottenere una trasparenza nell'utilizzo di un tipo piuttosto che un altro, dimostrato anche dal fatto che JADE-LEAP per J2SE e JADE coincidono esattamente in termini di API. Nonostante però dall'esterno JADE e JADE-LEAP siano praticamente uguali, guardando all'interno si notano alcune differenze sostanziali come ad esempio il protocollo di comunicazione che è basato sul *Jade Inter Container Protocol* (JICP), diversamente dalla versione nativa di JADE che si basa su RMI.

Grazie al modulo LEAP si riesce quindi ad ottimizzare la gestione dei dispositivi con risorse limitate e connesse attraverso una rete wireless. Con l'attivazione del modulo un container viene eseguito in *split mode* come rappresentato in figura 3.4 tratta da [12], ovvero vengono create due parti: il *front-end* e il *back-end* rispettivamente eseguiti nel terminale mobile e nella rete fissa. Il primo fornisce agli agenti le stesse caratteristiche di un container implementandone però solo alcune e delegando le rimanenti al back-end, un processo remoto che invece viene visto come un normale container da tutti i container della piattaforma. Ogni front-end è collegato al proprio back-end attraverso una connessione permanente e bi-direzionale. Questo tipo di approccio è conveniente per diversi aspetti, tra cui sicuramente il fatto che spostando parte del carico al lato back-end vengono richiesti minor potenza di calcolo e memoria nel dispositivo mobile. Si segnala inoltre che nel caso in cui il front-end rilevasse una temporanea perdita di connessione è in grado di ristabilirla appena possibile. Sia il front-end che il back-end implementano il meccanismo store-and-forward. Questo significa che in caso di messaggi non trasmessi a causa di una disconnessione temporanea questi vengono memorizzati in un buffer e consegnati non appena la connessione viene riattivata. Un ultimo punto da segnalare è che alcune informazioni

che si scambiano i container, come ad esempio la ricerca di un agente in esecuzione, sono curati solamente dai back-end: assieme alla codifica di comunicazione efficiente, si ha un'ottimizzazione dell'uso del canale wireless. Deve essere presente, inoltre, perchè tutto funzioni correttamente un meccanismo che in JADE è chiamato *mediator* che si occupa della gestione dei back-end e di tutti i front-end attivi sui dispositivi degli utenti.

3.1.3.1 JADE-LEAP per ANDROID

JADE ANDROID è un'estensione di JADE che fornisce il supporto per l'utilizzo di JADE-LEAP su una piattaforma Android, che verrà presentata in seguito. La versione utilizzata per lo sviluppo del progetto di tesi è la 1.2. In dettaglio, questo add-on consente di avere un'interfaccia che permette alle applicazioni Android di far partire un agente locale e dei behaviour oltre che la possibilità di scambiare degli oggetti con gli altri agenti della piattaforma. JADE ANDROID offre la compatibilità con la macchina virtuale di Android e fa fronte alle limitazioni dei dispositivi mobili e delle reti wireless. Nell'add-on si trovano due oggetti Java che risultano fondamentali per lo sviluppo di applicazioni che interagiscono con l'ambiente Android:

- `jade.android.MicroRuntimeService`: è un servizio per Android che è responsabile della configurazione dell'ambiente Jade e della gestione del Jade Runtime;
- `jade.android.JadeGateway`: fornisce un gateway tra l'applicazione Android e il sistema multiagente basato su JADE e utilizza al suo interno `jade.core.MicroRuntime` per aprire un container in split mode.

3.1.4 Gli add-on

Le estensioni alla piattaforma JADE di base sono numerose. Nel progetto di tesi sono stati utilizzati JADE-LEAP e JADE-ANDROID che sono stati presentati in precedenza. Si citano altri add-on che possono diventare utili al fine di realizzare applicazioni particolari, strumenti che però non sono stati utilizzati nel corso del progetto. Tra le varie estensioni si segnalano:

- JADE-Security: add-on che introduce uno strato aggiuntivo di sicurezza allo scopo di proteggere sia gli agenti che le piattaforme mediante meccanismi di autenticazione e di autorizzazione;
- Trusted Agents: questa estensione permette di assicurare il fatto che un agente che entra nel sistema possegga un token per la validazione;
- Java-WSIG (Web Service Integration Gateway): l'add-on consente una diretta esposizione dei servizi degli agenti registrati con il DF come Web Services.

3.2 ANDROID

Nello sviluppo del progetto di tesi è stato utilizzato Android, un sistema operativo per telefonia mobile, anche se il suo utilizzo ha coperto solo l'ultima parte dell'implementazione. In seguito vengono presentati i concetti basilari di questo sistema operativo con una breve analisi della struttura di un'applicazione.

3.2.1 Concetti base

Negli ultimi anni, oltre alla grande rivoluzione portata da Internet che è entrato in modo sempre più insistente nella vita di tutti noi, si riscontra un enorme cambiamento anche nell'utilizzo dei dispositivi mobili. Quello che inizialmente era un semplice telefono cellulare è diventato uno strumento che offre all'utilizzatore sempre più servizi che spaziano dalla localizzazione GPS all'invio e ricezione della mail, dalla navigazione in internet alla visualizzazione di file multimediali. Android, un prodotto firmato Google, si inserisce a pieno in questo contesto fornendo una piattaforma open source per il settore mobile.

Android nasce dall'esigenza di fornire una piattaforma aperta per la realizzazione di applicazioni mobili e può esser definito come uno stack software di strumenti e librerie per la realizzazione di tali applicazioni. Il termine *open* può ben rappresentare la posizione di questa piattaforma. Si può riscontrare infatti che:

- Android utilizza tecnologie open source, come ad esempio il kernel Linux;
- Le librerie e le API utilizzate per la sua realizzazione sono le stesse di quelle che vengono utilizzate dagli sviluppatori per creare le proprie applicazioni;
- Il suo codice è open source, quindi scaricabile e consultabile liberamente, secondo la licenza Open Source Apache License 2.0 che permette ai venditori di costruire le proprie estensioni anche proprietarie senza dover pagare alcuna royalty per l'utilizzo di Android nei propri dispositivi.

Per la realizzazione e lo sviluppo delle applicazioni, Android mette a disposizione un SDK (Software Development Kit) fornendo tutte le API e gli strumenti necessari. Il linguaggio di programmazione scelto da Google è Java, facilitando quindi il lavoro agli sviluppatori che avrebbero altrimenti dovuto impararne uno nuovo. Non è questa però la sola motivazione che ha spinto a scegliere Java. La definizione di un nuovo linguaggio avrebbe

obbligato, infatti, Google a un lavoro aggiuntivo per fornire specifiche, un compilatore, un debugger, almeno un IDE opportuno e librerie e documentazione idonee. Java non ha portato però solo vantaggi in quanto il suo utilizzo va in contrasto con la natura open di Android: i dispositivi che intendono utilizzare la virtual machine (VM) in ambiente J2ME, infatti, devono pagare una royalty. Questo stride con la licenza Apache citata precedentemente. Per risolvere questo problema, allora, il team di Android ha deciso di non eseguire direttamente il bytecode Java ma di utilizzare una propria VM, inizialmente proposta e sviluppata da Dan Bornstein, a cui è stato dato il nome di *Dalvik*, prendendo spunto da una località islandese. Questa virtual machine è ottimizzata per sfruttare la poca memoria presente nei dispositivi mobili, consente di far girare diverse istanze della macchina virtuale contemporaneamente e nasconde al sistema operativo sottostante la gestione della memoria e dei thread. In particolare viene eseguito il codice contenuto all'interno di file che hanno l'estensione `.dex` ottenuti nella fase di building a partire dai binari di Java, i tipici `.class`. La Dalvik Virtual Machine (DVM) soddisfa, quindi, l'esigenza di risparmiare più spazio possibile per la memorizzazione e l'esecuzione delle applicazioni attraverso il meccanismo di generazione del codice che è orientato all'utilizzo di registri, diversamente da quanto avviene invece per la Java VM in cui è presente una tecnica stack based. Il risultato che si ottiene consente, in prospettiva futura, una sensibile riduzione del numero di operazioni da eseguire. Risulta molto importante avere un compilatore in grado di eseguire le ottimizzazioni richieste riducendo allo stretto indispensabile lo sforzo a runtime, un aspetto, questo, necessario in ambiente mobile. Essendo in grado di eseguire file dex, la DVM non consente eventuali elaborazioni a runtime di informazioni disponibili all'interno del codice binario Java.

3.2.2 La struttura di un'applicazione

Una singola applicazione è composta da una parte di codice, da un insieme di risorse e da un file di configurazione chiamato `AndroidManifest.xml` nel quale vengono elencate tutte le dichiarazioni dei componenti che costituiscono l'applicazione e i requisiti che essa richiede.

Le parti principali che stanno alla base dell'architettura Android sono:

Activity Un'activity è una delle parti fondamentali dell'architettura in quanto rappresenta la gestione dell'interfaccia grafica e in particolare rende possibile l'interazione dell'utente con l'applicazione, gestendo le azioni tramite l'utilizzo delle `View`. In genere può essere vista come una singola schermata oppure un contenitore di componenti grafici che sono visibili o nascosti. Un'applicazione, in genere, è caratterizzata da più schermate che si possono alternare a video offrendo diverse funzionalità, dalla visualizzazione di informazioni o immagini all'interazione

con l'utente. Il passaggio da un'activity ad un'altra è gestita tramite una struttura a stack in cui il primo elemento è l'activity al momento visualizzata: al termine questa viene rimossa dallo stack con possibilità di inviare delle informazioni all'activity che la precedeva, la quale diventerà attiva a video. Le `Activity` sono alla base del meccanismo di ottimizzazione delle risorse adottato da Android e sono caratterizzate dal proprio *lifecycle*.

Intent e Intent Filter La classe `Intent` consente di riutilizzare le activity e altri componenti per eseguire operazioni che possono essere comuni a più applicazioni ed è caratterizzata dalle informazioni relative al comando da eseguire e da un meccanismo per l'identificazione del tipo di dati che verranno utilizzati. Per esempio, per visualizzare le informazioni contenute nel contatto di una persona, un'activity può creare un intent con l'azione `ACTION_GET_CONTENT` e l'informazione relativa al dato dell'azione rappresentato da un uri che individua la persona nella rubrica. L'intent creato può essere utilizzato poi dall'activity per l'avvio di una nuova attività in grado di gestirlo. Esiste inoltre un meccanismo che consente alle componenti di un'applicazione di dichiarare l'insieme degli intent che sono in grado di gestire attraverso gli intent filter.

Broadcast Intent Receiver Questo componente risulta necessario qualora lo sviluppatore volesse rendere l'applicazione reattiva a un dato evento esterno, come ad esempio l'arrivo di una telefonata oppure di un determinato orario, al fine di gestire delle informazioni associate all'evento stesso. L'attivazione di un intent receiver non implica obbligatoriamente l'esecuzione di un'activity: può accadere infatti che l'evento venga notificato direttamente, come nel caso di una telefonata, oppure visualizzi una icona o viene fatto vibrare il dispositivo sfruttando il `NotificationManager`.

Content Provider Un content provider è un sistema che consente la condivisione dei dati tra diverse applicazioni e rappresenta un aspetto fondamentale per la piattaforma Android, andando a realizzare la gestione dei dati. Le informazioni possono essere gestite attraverso diversi strumenti come l'utilizzo di file, di strumenti di configurazione e di un database SQLite. L'importanza di questo componente è sottolineata, ad esempio, dal fatto che i contatti, gli sms e l'insieme delle applicazioni sono informazioni esposte attraverso specifici Content Provider.

Service In Android non tutto è legato a un'interfaccia grafica, come accade ad esempio per un servizio. Per chiarire meglio il concetto di servizio viene proposto un esempio. Si consideri la riproduzione di un brano musicale mediante un apposito applicativo: com'è facile intuire un

```
public final class R {
    public static final class attr {
    }
    public static final class id {
        public static final int IPSpinner=0x7f050001;
        public static final int connectButton=0x7f050002;
        public static final int disconnectButton=0x7f050003;
        public static final int lblIP=0x7f050000;
        public static final int timer=0x7f050008;
        public static final int tvContent=0x7f050004;
        public static final int tvPP=0x7f050007;
        public static final int tvParkInfo=0x7f050006;
        public static final int tvWifi=0x7f050005;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
```

Fig. 3.5: Screenshot della classe `R`

player multimediale può essere costituito da diverse activity per le diverse funzionalità proposte, come la scelta del brano, la visualizzazione delle informazioni della canzone corrente o la barra di gestione del brano. Un requisito di un'applicazione come il player musicale è quello di garantire l'esecuzione dei brani nonostante si passi da una schermata all'altra. Per questo motivo non è corretto assegnare a un'activity la gestione della riproduzione stessa. Il player deve quindi avviare un service che consenta di mantenere la musica in sottofondo fino alla sua naturale terminazione.

Un ultimo aspetto da citare riguarda le risorse dell'applicazione e la loro gestione. Si segnala in particolare la classe `R` che viene autogenerata e gestita dal tool di building del progetto: questa classe contiene un insieme di costanti raggruppate in un insisme di classi statiche interne associate al particolare tipo di risorsa. In figura 3.5 viene riportata una parte della classe `R` dell'applicazione utilizzata nel progetto di tesi in cui è possibile osservare come ciascun elemento sia caratterizzato da una costante numerica. Se in un certo punto del programma, ad esempio, il programmatore volesse utilizzare uno specifico layout, nel nostro caso `main`, è sufficiente utilizzare la costante `R.layout.main` per riferirsi alla particolare risorsa definita nella cartella `res`. Questa cartella contiene alcune sottocartelle che corrispondono ai diversi tipi di risorsa possibili: possono essere presenti le icone del programma, i vari layout delle schermate oppure dei file che memorizzano dei dati, come nel caso di `strings.xml` che contiene un elenco delle stringhe utilizzabili.

In fase di progettazione di un'applicazione, uno sviluppatore si deve quindi focalizzare sulla definizione di un'eventuale GUI, sulla gestione dei dati, sulle operazioni in background e sulle notifiche verso l'utente.

CAPITOLO 4

Il progetto dell'infrastruttura

Indice

Dopo aver descritto i sistemi utilizzati e le tecnologie adottate per la realizzazione del progetto, in questo capitolo vengono presentate le scelte progettuali e le modalità implementative del software. Il progetto è stato interamente sviluppato in linguaggio Java utilizzando l'ambiente di sviluppo integrato Eclipse al quale è stato aggiunto l'SDK Android per poter realizzare l'applicazione che deve essere eseguita su un dispositivo dotato di Android 2.1. Come già accennato precedentemente si tratta dello smartphone HTC Desire A8181.

4.1 Introduzione

Il progetto nasce dalla necessità di verificare la fattibilità di utilizzo di sistemi ubiqui in ambiente urbano con particolare interesse alla problematica dei parcheggi e della viabilità in quel contesto. Più dettagliatamente la tesi ha lo scopo di progettare e di implementare questi concetti mediante un'applicazione software che simula l'infrastruttura e tutte le componenti che caratterizzano il sistema escluso il telefonino. Per questo elemento viene utilizzato infatti uno smartphone HTC Desire A8181 dotato di Android.

Come è stato visto nella sezione 2.1, l'utilizzo dei sistemi ubiqui spazia in diversi settori. Quello che si è voluto fare in questo lavoro è stato implementare un sistema che realizzasse il concetto di ubiquità per favorire l'utente umano in un determinato ambiente caratterizzato, nella fattispecie, dal contesto cittadino. Negli ultimi anni si è assistito ad un notevole aumento del traffico portando i guidatori ad avere crescenti difficoltà nell'attraversamento delle città e nel trovare un parcheggio pertanto la realizzazione di un sistema a supporto risulta più che mai necessario.

Il sistema che è stato implementato consente di rilevare, note le coordinate GPS di interesse, il parcheggio *più conveniente* al guidatore. La politica che sta alla base della convenienza può essere implementata in diversi modi. In questo lavoro di tesi, al fine di avere un pacchetto applicativo che simulasse un corretto funzionamento, è stato scelto di adottare come più conveniente il parcheggio più vicino in termini di distanza. Il sistema, inoltre, può essere esteso con ulteriori funzionalità come ad esempio la gestione della pianificazione del tragitto in modo dinamico a seconda della situazione corrente del traffico.

Per realizzare una tale infrastruttura è stato deciso di utilizzare un sistema ad agenti mobili in quanto offre delle funzionalità maggiori rispetto ad altre architetture, come è stato ampiamente spiegato nella sezione dedicata 2.2.2. La piattaforma scelta per implementare tale sistema è JADE, precedentemente presentata nella sezione 3.1, che ha permesso di integrare, con relativa semplicità, lo smartphone dotato di Android grazie all'utilizzo dell'add-on JADE-LEAP specifico, ovvero JADE-ANDROID, illustrato nella sezione 3.1.3.1. Avendo dovuto simulare elementi quali l'ambiente cittadino e l'automobile, si è reso necessario implementare anche alcune soluzioni atte alla fornitura di servizi quali la connettività bluetooth e wireless, che verranno presentate con maggior dettaglio nel corso del capitolo. Sono stati presi come riferimento per lo sviluppo i sistemi di parcheggio del comune di Treviso [22, 23] e di Verona [24] oltre al framework generico per sistemi ubiqui presentato nella sezione 2.1.2.

La realizzazione di questa infrastruttura di supporto alla viabilità in ambiente urbano è un esempio che mira a dimostrare come l'utilizzo delle più recenti tecnologie nell'implementazione di un sistema ubiquo, in particolare del paradigma ad agenti, consenta di raggiungere l'efficacia, l'efficienza e

l'affidabilità anche in situazioni critiche mantenendo sempre la trasparenza di utilizzo nei confronti dell'utente.

4.2 Gli elementi del sistema

La parte principale dell'intero lavoro di tesi è costituito dall'infrastruttura. In particolare si è reso necessario procedere con la simulazione di una città seguendo alcune scelte strutturali effettuate a priori. Gli elementi costitutivi dell'infrastruttura del progetto sono essenzialmente i seguenti:

- Area;
- Quartiere;
- System Manager;
- Park.

Al sistema si aggiungono, poi, le automobili e gli smartphone posseduti dai guidatori. Essendo questo un progetto che simula tutti i componenti ad esclusione dei telefoni cellulari è risultato necessario effettuare un'attenta analisi progettuale, prima di procedere con l'effettivo sviluppo. La decisione presa è stata quella di suddividere l'intera superficie cittadina in diversi quartieri o district che a loro volta rappresentano un insieme di parti più piccole, dette zone o aree. Ciò che si ottiene è quindi una struttura gerarchica che consente di raggiungere l'obiettivo di semplificare il più possibile la gestione dell'aspetto infrastrutturale della città, senza appesantire inutilmente il carico computazionale dell'intero sistema.

Per l'avvio e il caricamento dell'intera struttura è stato implementato un apposito agente, chiamato `SystemStarter`, che raccoglie tutte le informazioni necessarie da un file di inizializzazione strutturato nel seguente modo:

```
districts [tot <nrTot>]|[<nrRiga1> <nrRiga2> ... <nrRigan>]
latsx <val> longsx <val>
latdx <val> longdx <val>
d0 [tot <valD0>]|[<nrRiga1> <nrRiga2> ... <nrRigak0>]
d1 [tot <valD1>]|[<nrRiga1> ... <nrRigak1>]
...
d(nrTot-1) [tot <valD(nrTot-1)>]|[<nrRiga1> ... <nrRigak(nrTot-1)>]
```

Codice 4.1: Struttura del file di inizializzazione del sistema

L'ambiente urbano è stato pensato come uno spazio suddiviso in distretti disposti lungo righe e colonne. Ciascun quartiere, poi, è a sua volta caratterizzato dalle proprie aree o zone distribuite anch'esse secondo il concetto riga-colonna. Nello specifico, è stata ideata una lista di parole chiave che consentissero di rappresentare questi concetti in modo strutturato senza escludere la semplicità.

Analizzando con attenzione il file si osserva come innanzitutto vengano richiesti i dati riguardanti i quartieri che possono essere rappresentati in due formati: il primo indica il numero totale di district presenti nel sistema mentre il secondo esplicita il numero di distretti per ogni riga. Si sottolinea che, nel caso in cui venisse utilizzata la prima opzione, il numero di quartieri per ogni riga è rappresentato dalla parte intera della radice quadrata del numero totale e il numero di righe totali è pari allo stesso valore, andando quindi a costituire una griglia quadrata di $\lfloor\sqrt{nrTot}\rfloor$ elementi per lato. Le due righe successive del file prevedono l'inserimento delle coordinate GPS del sistema ipotizzando una copertura territoriale di forma rettangolare o quadrata: per questo motivo vengono richieste solamente le coordinate per due punti che rappresentano rispettivamente il vertice in alto a sinistra e quello in basso a destra. Le righe seguenti specificano la struttura interna di ogni singolo distretto. Le modalità per indicare il numero e la disposizione delle aree all'interno di ciascun quartiere sono le medesime di quanto già presentato per i district. Anche in questo caso è possibile, quindi, inserire il numero di aree per ogni riga del quartiere oppure il numero totale di aree del distretto indicando implicitamente, quindi, una griglia quadrata di $\lfloor\sqrt{valDistrict}\rfloor$ elementi per lato. Le coordinate GPS delle singole aree e dei quartieri vengono calcolate dividendo la lunghezza e l'altezza del sistema in base al numero di componenti rispettivamente per riga e per colonna. Se la lunghezza, ad esempio, del sistema è di valore x e in una determinata riga ci sono y quartieri allora la lunghezza di ciascun district di quella specifica riga è pari a x/y. Analogamente per il calcolo delle coordinate delle singole aree, nel quale si va a operare sulla lunghezza del quartiere di interesse utilizzando il numero di aree per riga. La numerazione dei distretti è da sinistra a destra e dall'alto in basso.

Nel codice 4.2 viene riportato un file di esempio di inizializzazione del sistema ampiamente utilizzato in fase di sviluppo del progetto. L'analisi e l'elaborazione di questo file porta alla realizzazione dell'infrastruttura rappresentata in figura 4.1. Dall'esempio si osserva come la definizione delle aree per il distretto 0 dia lo stesso risultato di quella per il distretto 1 essendo il valore 4 un quadrato perfetto: scrivere 4 oppure due righe da 2 aree ciascuna è la stessa cosa. L'immagine è stata appositamente colorata per mettere in evidenza la copertura territoriale dei singoli quartieri e delle loro aree di appartenenza.

```
districts tot 4
latsx 0 longsx 0
latdx 150 longdx 150
d0 2 2
d1 tot 4
d2 tot 4
d3 2 2
```

Codice 4.2: Esempio di file di inizializzazione

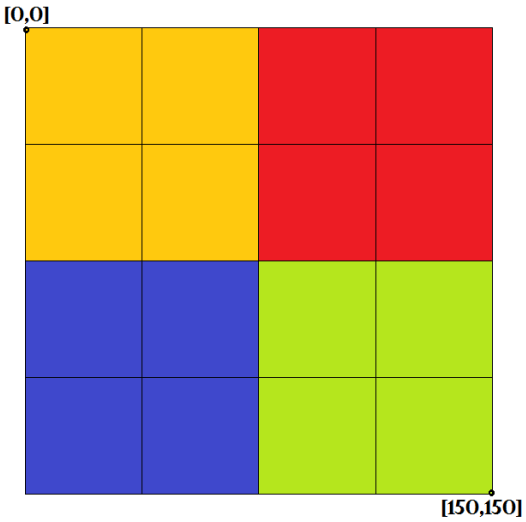


Fig. 4.1: Infrastruttura risultante dal file di inizializzazione nel codice 4.2

Da un punto di vista pratico, l'agente `SystemStarter` crea una serie di container secondari che vanno a rappresentare concettualmente la suddivisione geografica e i diversi ruoli amministrativi. Per questo viene generato il container `SystemManager`, che ospita l'omonimo agente, e uno per ogni quartiere, che va a contenere l'agente di gestione del distretto e quelli delle aree che lo compongono. A questi si aggiungono gli agenti che hanno in gestione i parcheggi, quelli che simulano l'automobile che si trova in quel determinato quartiere e l'agente che ha in gestione la sua connessione wifi con il sistema. Per i container dei district è stato necessario pensare a una nomenclatura che consentisse una rapida comprensione della loro distribuzione nella mappa. Per far questo si è deciso di utilizzare la forma `District_<riga>_<colonna>` in cui `riga` e `colonna` rappresentano due indici interi che rispettano le seguenti condizioni:

$$0 \leq \texttt{riga} < \texttt{totRigheSistema}$$
$$0 \leq \texttt{colonna} < \texttt{totColonneSistema}$$

Riprendendo l'esempio proposto in figura 4.1, si ha come risultato finale la generazione di un container chiamato `SystemManager` e di quattro contenitori relativi ai quartieri dal nome `District_0_0` (arancione), `District_0_1` (rosso), `District_1_0` (blu) e `District_1_1` (verde).

Il ruolo dell'agente `SystemStarter` non prevede però solo la creazione dei container ma permette anche l'avvio degli agenti di gestione dei quartieri. La nomenclatura scelta per tali agenti è stata, per comodità, quella dei rispettivi contenitori, pertanto il generico distretto `Discrict_x_y` sarà rappresentato da un agente di nome `Discrict_x_y` e situato nel container omonimo. A questi agenti vengono passati, in fase di creazione, i dati relativi alle proprie

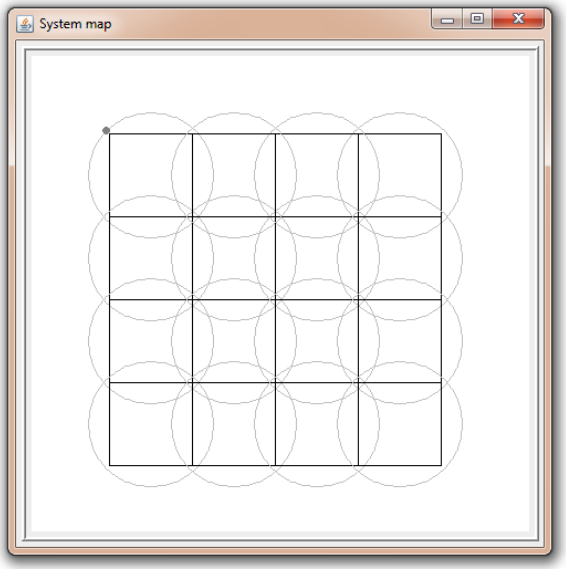


Fig. 4.2: Esempio di mappa del sistema con l'esempio nel listato 4.2

coordinate nel sistema e alle aree che costituiscono il quartiere di cui sono rappresentanti.

4.2.1 Il System Manager

Nel framework generico di sistema ubiquo presentato nella sezione 2.1.2 viene introdotto il concetto di administrator sottolineando l'importanza delle sue mansioni e della sua presenza. L'agente `SystemManager` svolge proprio questo ruolo avendo in gestione l'intero sistema. Futuri sviluppi del progetto prevedono l'implementazione di ulteriori funzionalità oltre a quelle di controllo già inserite.

In questa prima fase il `SystemManager` offre quindi solo alcune delle opzioni di controllo previste per un amministratore. Le funzionalità introdotte sono:

- *registrazione del servizio* `SYSTEM_MANAGER`: la registrazione di questo servizio simula l'offerta delle funzionalità dell'intero sistema alle automobili che si avvicinano o entrano in esso. Queste, infatti, hanno la possibilità di utilizzare o meno l'aiuto offerto dal sistema per operazioni quali la ricerca di un parcheggio nei pressi di un appuntamento. Nel caso in cui il guidatore volesse utilizzare il sistema non deve fare altro che cercare questo tipo di servizio inviando una richiesta al DF: se l'auto si trova in una posizione coperta dal segnale wifi e il sistema risulta registrato allora è possibile accedervi mediante un opportuno scambio di messaggi. La registrazione del servizio, nel caso specifico

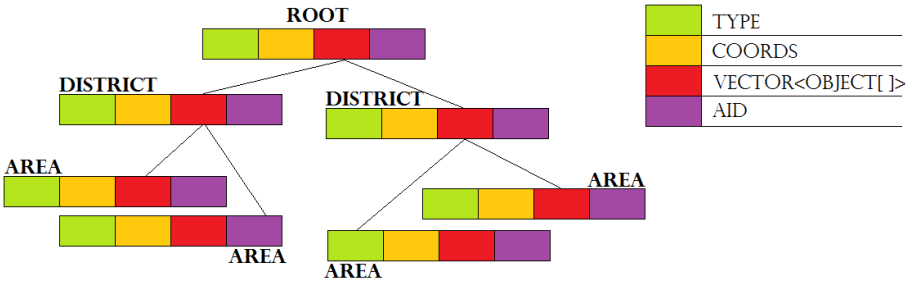


Fig. 4.3: Struttura dell'albero interno all'agente `SystemManager`

dell'implementazione, semplifica questo concetto offrendo a un'automobile la possibilità di capire qual è l'amministratore del sistema per poter iniziare la procedura di ingresso. Negli sviluppi futuri del sistema si può pensare di personalizzare tale servizio a seconda del contesto cittadino.

- *creazione della mappa del sistema*: l'agente `SystemManager` offre la possibilità di visualizzare la situazione corrente del sistema andando a rappresentare la mappa delle varie aree, dei distretti e dello spostamento delle automobili. La figura 4.2 rappresenta uno screenshot della finestra di visualizzazione utilizzando come file di inizializzazione il codice proposto nel listato 4.2: oltre all'infrastruttura appare anche un punto grigio in alto a sinistra che rappresenta in questo caso una macchina autenticata nel sistema. In questa immagine si trovano anche delle circonferenze che rappresentano la copertura del segnale wireless per ciascuna area: il significato e il loro utilizzo verranno spiegati nella sezione 4.2.3 riguardante le aree.

- *visione globale del sistema*: l'agente `SystemManager` memorizza al suo interno la struttura dell'intero sistema grazie all'utilizzo di una particolare struttura ad albero che consente di migliorare la ricerca in caso di interrogazioni da parte dell'utente. La creazione di tale albero viene effettuata all'avvio del sistema e risulta possibile grazie alla ricezione e all'elaborazione di particolari messaggi dagli agenti di tipo `Area`: questi messaggi consentono di aggiungere i nodi all'interno dell'albero a seconda del quartiere di appartenenza. Osservando la struttura presentata in figura 4.3 si osserva come il singolo nodo sia stato strutturato in quattro sezioni, ciascuna con un significato preciso:

  - *type*: la prima parte identifica il tipo di nodo e il valore che può assumere deve esser scelto tra `TYPE_ROOT`, `TYPE_DISTRICT` e `TYPE_AREA`;
  - *coords*: questo campo contiene le coordinate dell'entità di riferimento pertanto il nodo root conterrà le coordinate del sistema,

uno di tipo `TYPE_DISTRICT` quelle del quartiere e per i nodi caratterizzati da `TYPE_AREA` le coordinate di area; tali coordinate sono contenute in un array di quattro posizioni e sono costituite da latitudine e longitudine del punto in alto a sinistra e a seguire di quello in basso a destra, assumendo sempre che le entità siano di forma rettangolare o quadrata;

- *informazioni sui figli*: la terza parte del nodo è costituito da un `Vector` di `Object[]` in cui ciascun array inserito rappresenta alcune informazioni di un nodo figlio e in particolare il riferimento al nodo stesso e le sue coordinate GPS al fine di velocizzare la ricerca;
- *AID*: quest'ultima sezione contiene l'`AID` dell'agente di riferimento per la specifica entità quindi, nel caso del nodo radice, sarà contenuto l'`AID` del `SystemManager`, per i nodi di tipo `TYPE_DISTRICT` e `TYPE_AREA`, invece, rispettivamente l'AID dell'agente quartiere ed area associato.

La creazione dell'albero termina non appena il numero di nodi di tipo area inseriti coincide con il numero totale di aree inviate al `SystemManager` dall'agente `SystemStarter` mediante un apposito messaggio. Una volta terminato, il manager invia a tutte le aree un messaggio contenente le informazioni sui loro vicini (*neighbour*) al fine di aumentare il pacchetto informativo a disposizione e di facilitare eventuali ricerche o comunicazioni tra entità contigue, limitando il carico dell'intero sistema.

- *monitoraggio delle automobili e dei dispositivi*: una particolare finestra visualizza tutti i dispositivi collegati in wifi indicando l'area in cui si trovano e il timestamp all'ultimo keepalive ricevuto. All'interno del manager di sistema si trova implementata un'estensione della classe `TickerBehaviour` che funge da controllore: periodicamente questo behaviour elimina le connessioni che non risultano più attive ovvero quelle che non aggiornano il timestamp mediante l'invio di un messaggio di keepalive. Il `SystemManager` contiene le informazioni delle unità connesse in particolari strutture dati che semplificano le operazioni di ricerca e di aggiornamento.

- *gestione della messaggistica*: il `SystemManager` è un agente che riceve e deve processare un numero elevato di messaggi di diverso tipo. Questi vanno a costituire particolari tipi di interazione dell'agente administrator non solo con le entità dell'infrastruttura, quindi aree e district, ma anche con i dispositivi. Queste interazioni verranno presentate con maggior dettaglio in seguito nel corso del capitolo. Esempi di messaggi che il `SystemManager` riceve sono quelli di richiesta di accesso

al sistema, di ricerca di una specifica area in base a delle coordinate GPS di riferimento o del keepalive delle connessioni wifi dei dispositivi con il sistema. A questi, poi, si aggiunge una serie di tipologie di messaggi di servizio che consentono, soprattutto in fase di simulazione, di avere una miglior visione globale del sistema: fanno parte di questa categoria i messaggi che aggiungono elementi grafici alla mappa dell'infrastruttura, come nel caso dei parcheggi o delle automobili.

4.2.2 I quartieri

L'entità che sta al livello sottostante del `SystemManager` è il `District` o quartiere. Questo componente è stato inserito per aggiungere un controllo più localizzato rispetto a quanto effettuato dall'agente amministratore globale. In questa prima fase di sviluppo i vari agenti che caratterizzano i quartieri svolgono operazioni di base, lasciando, in visione futura, la possibilità di implementare altre funzionalità a supporto del sistema. Generati dall'agente `SystemStarter` e situati in un container secondario omonimo all'agente di quartiere, i `District` svolgono un ruolo molto importante nella generazione dell'infrastruttura in quanto sono i responsabili della creazione degli agenti di tipo `Area`. In fase di setup, infatti, un generico agente `District_x_y` utilizza gli argomenti passati alla creazione per generare tutte le aree che lo compongono andando a inserire gli agenti relativi nel container di sua appartenenza. La scelta della nomenclatura rispecchia complemente quella fatta per i quartieri: un'area appartiene a un preciso quartiere ed è situata in una specifica posizione al suo interno identificando riga e colonna. Per questo si ha che il nome generico assegnato a un'area è `Area_<rigaD>_<colonnaD>_<rigaA>_<colonnaA>` dove `rigaD` e `colonnaD` rappresentano gli indici del quartiere di interesse e `rigaA` e `colonnaA` quelli dell'area all'interno del distretto. Tali indici devono rispettare, come nel caso del nome dei quartieri, i seguenti vincoli:

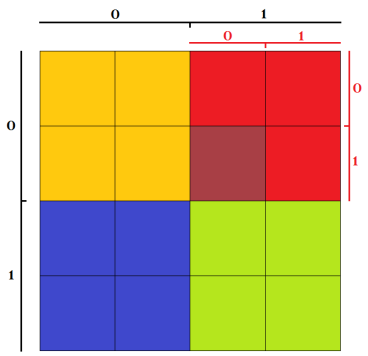


Fig. 4.4: Nomenclatura

$$0 \leq \texttt{rigaD} < \texttt{totRigheSistema}$$
$$0 \leq \texttt{colonnaD} < \texttt{totColonneSistema}$$
$$0 \leq \texttt{rigaA} < \texttt{totRigheDistretto}$$
$$0 \leq \texttt{colonnaA} < \texttt{totColonneDistretto}$$

Sfruttando la figura 4.4 si riporta un esempio per fissare meglio il concetto. Il quartiere rosso è chiamato `District_0_1` in quanto si trova nella prima riga, quindi indice 0, e seconda colonna, indice 1. Prendendo in considerazione poi l'area evidenziata contenuta in esso si ha che il nome che le viene

attribuito è `Area_0_1_1_0`: la dicitura `_0_1` iniziale identifica il distretto rosso di appartenenza, mentre l'ultima parte del nome, `_1_0`, rappresenta la localizzazione al suo interno, quindi seconda riga e prima colonna.

Analogamente al `SystemManager`, anche per i distretti è prevista la gestione delle connessioni wifi attive dentro i propri confini. Il monitoraggio avviene al medesimo modo utilizzando le stesse strutture dati e la visualizzazione delle entità connesse avviene tramite una finestra simile a quella presentata nella sezione precedente. L'unica distinzione è data dal fatto che, mentre per il `SystemManager` vengono indicate le unità connesse in tutte le aree del sistema con il relativo timestamp associato, in questo caso si visualizzano solamente quelle che rientrano nei confini dell'entità distrettuale di riferimento. I dati relativi alle varie connessioni vengono aggiornate mediante la ricezione di messaggi di tipo keepalive inviati dalle aree che costituiscono il quartiere e periodicamente viene effettuato il controllo di eventuali connessioni perse e scadute utilizzando un apposito behaviour. È da segnalare, inoltre, un ulteriore behaviour periodico che invia al `SystemManager` i dati aggiornati delle unità connesse sotto forma di keepalive: in questo modo l'amministratore del sistema è in grado di aggiornare i propri dati eseguendo le opportune operazioni di modifica, inserimento e cancellazione di *entry* nelle strutture interne.

4.2.3 Le aree

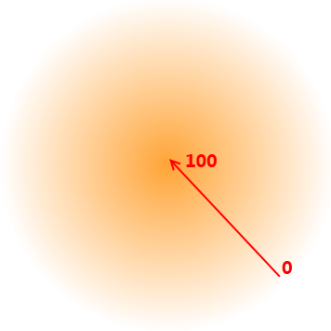


Fig. 4.5: Modello del segnale wifi

Le aree sono, all'interno del sistema, l'elemento strutturale più piccolo e sono dotate di funzionalità di controllo delle zone geografiche di interesse e di comunicazione con il sistema centrale per coordinare le attività. Come già detto nella sezione riguardante i quartieri, un agente di tipo `Area` ha come nome una stringa del tipo `Area_x_y_z_t` rappresentando l'appartenza al `District_x_y` e la localizzazione al suo interno alla riga di indice `z` e alla colonna di indice `t`. In fase di setup l'agente invia un messaggio opportunamente formattato al `SystemManager` che consente di aggiungerlo all'albero del sistema e, in seguito, crea i parcheggi di zona andando a processare degli opportuni file che contengono tutte le informazioni necessarie. Questa parte verrà trattata in maggior dettaglio nella sezione dedicata ai parcheggi.

Una delle caratteristiche principali dell'entità `Area` è fornita dalla copertura wifi che sta alla base del sistema ubiquo implementato: l'interazione dell'automobile con il sistema avviene infatti tramite la connessione wireless

alla rete locale. Per realizzare la copertura del segnale non è stato utilizzato un modello matematico preciso in quanto esula dagli obiettivi del lavoro di tesi. Per questo motivo è stato deciso di implementare un'irradiazione dal punto centrale dell'area con il segnale che va via via degradando in modo lineare all'allontanarsi dal centro. In figura 4.5 si osserva bene questo comportamento vedendo come nel centro si rilevi una copertura del 100% mentre al confine il segnale scenda allo 0%. Come già presentato nella figura 4.2, le coperture wireless si sovrappongo per l'ovvia necessità di non lasciare alcun punto del sistema scoperto. Per far questo è stato preso un valore ϵ a scelta congruo alle proporzioni del sistema per avere una sufficiente sovrapposizione dei segnali. Questo significa che, anche se le aree hanno delle determinate coordinate GPS in gestione, la copertura wireless che esse offrono esce da tali confini. In particolare, questa situazione può essere sfruttata da un'automobile nel momento in cui si sta avvicinando al sistema dall'esterno: percependo la presenza di un segnale wifi ha la possibilità di scoprire l'esistenza del sistema a supporto della viabilità e di iniziare così la procedura di accesso. Nel caso della realizzazione concreta dell'infrastruttura presentata in questo lavoro di tesi può risultare utile quindi prevedere una copertura wireless presistema per delle operazioni preliminari. Nella realtà la copertura wireless non può essere costituita da un'unica irradiazione centralizzata nell'area, soprattutto se la zona di interesse copre uno spazio geografico piuttosto ampio. In questo caso sarà necessario effettuare un'analisi preliminare della situazione per procedere poi con l'adeguata copertura della rete wireless del sistema.

Un altro compito di un agente `Area` è quello di fornire il supporto per aprire una connessione wifi a un dispositivo che la richiede tramite il `SystemManager`. Sfruttando un particolare scambio di messaggi, che verrà presentato nella sezione riguardante le interazioni, l'area consente di arrivare al termine della procedura creando un agente responsabile della gestione della connessione per il dispositivo. Questo nuovo agente viene generato con il nome `Wifi_<localnameDispositivo>` e rappresenta un elemento fondamentale per la connettività. L'importanza di questo agente verrà sottolineata nella sezione 4.2.5 riguardante le automobili.

Altre funzionalità offerte dall'agente `Area` sono la ricerca della disponibilità di parcheggio, la gestione dello stato delle connessioni attive e dei parcheggi di zona. Queste due ultime opzioni vengono realizzate mediante l'utilizzo di specifici behaviour che sono stati implementati ad hoc per ottimizzare il risultato in esecuzione: in abbinata vengono impiegate delle particolari strutture dati per contenere tutte le informazioni necessarie evitando un eccessivo carico di messaggi sulla rete. Come avviene per il `SystemManager` e per gli agenti `District`, c'è un meccanismo di controllo dei dispositivi nell'area e uno di invio periodico di keepalive al quartiere di interesse con le informazioni aggiornate.

4.2.4 I parcheggi

I parcheggi rappresentano un elemento molto importante nel sistema e sono dotati di diverse funzionalità a seconda del tipo. In fase realizzativa è stato scelto di assumere che i parcheggi siano di forma rettangolare caratterizzati, come per le componenti costituenti del sistema, dalle coordinate GPS dei punti in alto a sinistra e in basso a destra e come punto di riferimento, ai fini della ricerca, le coordinate dell'intersezione delle diagonali. Si possono distinguere diverse tipologie di parcheggio e quelle prese in esame in questa prima implementazione del sistema sono:

- Parcheggio a disco orario;
- Parcheggio a pagamento;
- Parcheggio gratuito;
- Parcheggio residenti;
- Parcheggio disabili;
- Parcheggio rosa.

Esistono altre tipologie come ad esempio il parcheggio ibrido, inteso come a pagamento ma con un primo periodo temporale gratuito visto come a disco orario, per taxi, per corriere o per carico/scarico merci: queste opzioni non sono al momento gestite ma nelle future versioni sarà eventualmente possibile aggiungerle nello sviluppo.

Ciascun parcheggio è costituito da un agente di tipo `Park` che funge da controllore ed è specifico a seconda della modalità che rappresenta. Una cosa che accomuna tutte le diverse tipologie è la registrazione del loro tipo come servizio nel DF per la realizzazione di funzionalità di sistema future. A questa registrazione si aggiunge poi la gestione dei sensori di parcheggio mediante un apposito agente che simula l'intera struttura di sensori di presenza dell'auto. Non appena un'automobile arriva nella postazione il sensore la rileva ed invia all'agente controllore del parcheggio la notifica; la comunicazione tra i due agenti avviene anche in fase di partenza della macchina non appena il sensore percepisce l'evento. L'agente dei sensori, in questa prima fase di simulazione, prevede anche un'interfaccia grafica che consente di andare a modificare manualmente la situazione delle postazioni libere ed occupate del parcheggio con anche l'opzione di renderlo *full* o *empty* a seconda delle esigenze di test. Questa funzionalità ha consentito di verificare il corretto funzionamento del sistema simulando l'andamento dei posti liberi dei parcheggi. Grazie alle notifiche immediate da parte dell'agente sensore, il controllore del parcheggio conosce in realtime la situazione dello stato: una funzione che caratterizza tutti i tipi di parcheggio è la presenza di un behaviour ciclico che invia all'area di appartenenza proprio lo stato

in un dato istante. Le informazioni contenute variano a seconda del tipo di parcheggio ma il primo campo dei dati è comune. Questa informazione rappresenta la situazione istantanea della struttura e può assumere un valore tra `PARK_FREE`, `PARK_QUITE_FULL` e `PARK_FULL`. L'opzione *quite full* indica una situazione di quasi completezza con una soglia scelta a piacere: nell'implementazione si è preso come threshold il 15% sul totale dei posti andando a identificare così il numero di posti occupati necessario per entrare nella soglia `PARK_QUITE_FULL`. Nulla vieta di modificare tale parametro a piacere o a seconda di certe politiche, come ad esempio posizione del parcheggio nel sistema, orario o statistiche.

Ciascun agente parcheggio viene avviato dall'area di appartenenza durante la sua fase di setup mediante l'analisi di un file formattato opportunamente. Successivamente viene comunicato al `SystemManager` l'evento in modo che questo possa andare ad inserirlo nelle componenti da raffigurare nella rappresentazione dell'intero sistema. Tale file è costituito da diverse righe, una per ogni parcheggio dell'area e la sua formattazione prevede un numero variabile di token a seconda della tipologia di parcheggio. Il numero di valori per riga varia da sei a sette: i primi sei sono di significato comune per tutti mentre il settimo parametro è opzionale e, a seconda del contesto, viene inteso diversamente. Le parti sono nello specifico:

1. type: indica uno dei tipi di parcheggio presentati in precedenza mediante un'apposita sigla scelta tra `PDO` (disco orario), `PP` (pagamento), `PF` (gratuito), `PR` (residenti), `PH` (disabili) e `PPW` (rosa);
2. latitudine *relativa* del punto in alto a sinistra del parcheggio;
3. longitudine *relativa* del punto in alto a sinistra;
4. latitudine *relativa* del punto in basso a destra del parcheggio;
5. longitudine *relativa* del punto in basso a destra;
6. numero posti del parcheggio.

In questa fase implementativa, il settimo parametro interessa solamente i parcheggi a disco orario e quelli a pagamento assumendo significato diverso. Mentre nel primo va a identificare la durata espressa in minuti del posteggio, nel secondo indica la tariffa oraria. Si sottolinea inoltre come le coordinate GPS siano *relative* pertanto i valori devono soddisfare i seguenti vincoli contemporaneamente:

$$0 \leq \texttt{latitudineSx} < \texttt{maxLatitudineArea}$$
$$0 \leq \texttt{longitudineSx} < \texttt{maxLongitudineArea}$$
$$\texttt{latitudineSx} < \texttt{latitudineDx} \leq \texttt{maxLatitudineArea}$$
$$\texttt{longitudineSx} < \texttt{longitudineDx} \leq \texttt{maxLongitudineArea}$$

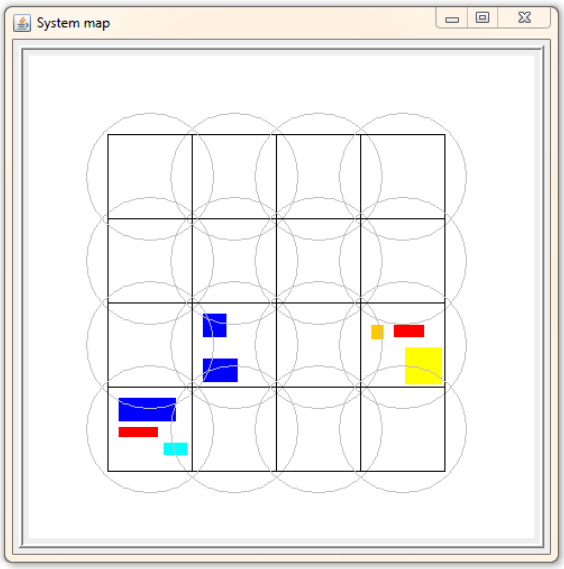


Fig. 4.6: Esempio di mappa del sistema con i parcheggi

dove `maxLatitudineArea` e `maxLongitudineArea` rappresentano la massima lunghezza di latitudine e longitudine dell'area. Si mette in evidenza che vanno quindi aggiunti rispettivamente `latitudineSxArea` ai valori `latitudineSx` e `latitudineDx` e `longitudineSxArea` a `longitudineSx` e `longitudineDx` per avere le coordinate nel sistema di riferimento globale. Il fatto di procedere in questo modo deriva da una questione di praticità per l'inserimento dei parcheggi in fase di sviluppo evitando di dover calcolare esattamente le coordinate del sistema ma limitandosi a valori più semplici da gestire. Un esempio di file processato dall'agente `Area` è proposto nel listato 4.3: in queste poche linee si vede come, per la specifica area presa in esame, ci siano tre parcheggi, uno di tipo disco orario di 400 posti e 50′ di durata, uno gratuito di 50 posti e uno a pagamento con tariffa oraria di 1.2€e di 30 postazioni.

```
PDO 5 5 15 30 400 50
PF 25 25 30 35 50
PP 18 5 22 22 30 1.2
```

Codice 4.3: Esempio di file di parcheggi

In figura 4.6 viene riportato un esempio della mappa dell'intero sistema al termine della creazione dei parcheggi: come si può osservare, le aree che contengono i parcheggi sono solamente tre e i vari `Park` sono di diverso tipo, come testimoniato dalla differente colorazione assunta.

I parcheggi offrono la possibilità di comunicare con l'automobile al momento del parcheggio tramite il modulo wifi sfruttando anche la mobilità

messa a disposizione dall'ambiente agent-based: la modalità con cui questo avviene verrà spiegata nella sezione relativa alle interazioni nel corso del capitolo.

Si sottolinea, infine, come le funzionalità dei diversi tipi di parcheggio vengano gestite singolarmente da classi specifiche. Nel caso particolare del parcheggio a pagamento si è assunto che il guidatore possegga una carta prepagata dalla quale vengono scalati i soldi in base alla tariffazione oraria e alla durata della permanenza potendo così utilizzare lo stesso modello pensato per il parcheggio a disco orario. Nei test effettuati sono stati presi in considerazione due tipi di parcheggio tra i sei proposti e sono precisamente il disco orario (`PDO`) e il pagamento (`PP`). I rimanenti sono stati implementati in parte anche se non offrono tutti i servizi e le funzionalità che li caratterizzano.

4.2.5 Le automobili

Le automobili in questo sistema sono gestite attraverso l'agente `Car` e la sua GUI, che viene riportata in figura 4.7. Da questa si possono capire alcune delle funzionalità che sono implementate per questo componente. Si osserva innanzitutto la possibilità di accendere e spegnere l'automobile e attivare e disattivare il modulo wifi. Nel caso specifico del progetto, avviare l'automobile non significa solo simulare la sua accensione ma rappresenta anche altri concetti. Primo di tutti la registrazione del servizio che simula l'attivazione del modulo bluetooth dell'auto, `BLUETOOTH_MOBILE`. A questo segue la richiesta di ingresso al sistema supponendo che l'automobile si trovi alle coordinate GPS di latitudine 0 e longitudine 0. Per procedere con l'accesso è necessario che l'automobile conosca il `SystemManager` pertanto viene effettuata una ricerca del servizio `SYSTEM_MANAGER` che ritorna l'`AID` del manager del sistema. A questo punto l'automobile inizia un processo di accesso al sistema e uno di acquisizione di una connessione wifi con l'area corretta. Nella realtà questo procedimento non è necessario in quanto i moduli wifi percepiscono direttamente il segnale ma, essendo questo un progetto che simula l'intero sistema, risulta indispensabile avere un meccanismo di identificazione dell'area e di instaurazione di un collegamento wifi. Una volta che questi processi sono andati a buon fine, l'automobile risulta entrata nel sistema ed è associata all'agente descritto dalla classe `WifiConnectionAgent`. La scritta *WIFI connected* che compare in figura 4.7 indica pertanto l'avvenuta connes-

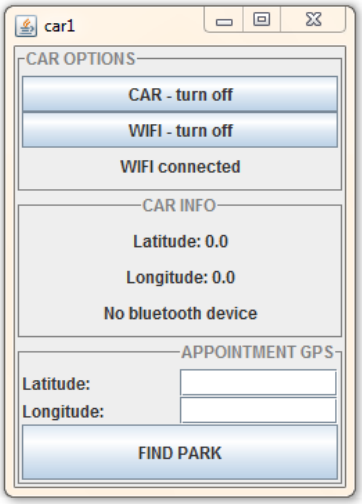


Fig. 4.7: La GUI di un agente `Car`

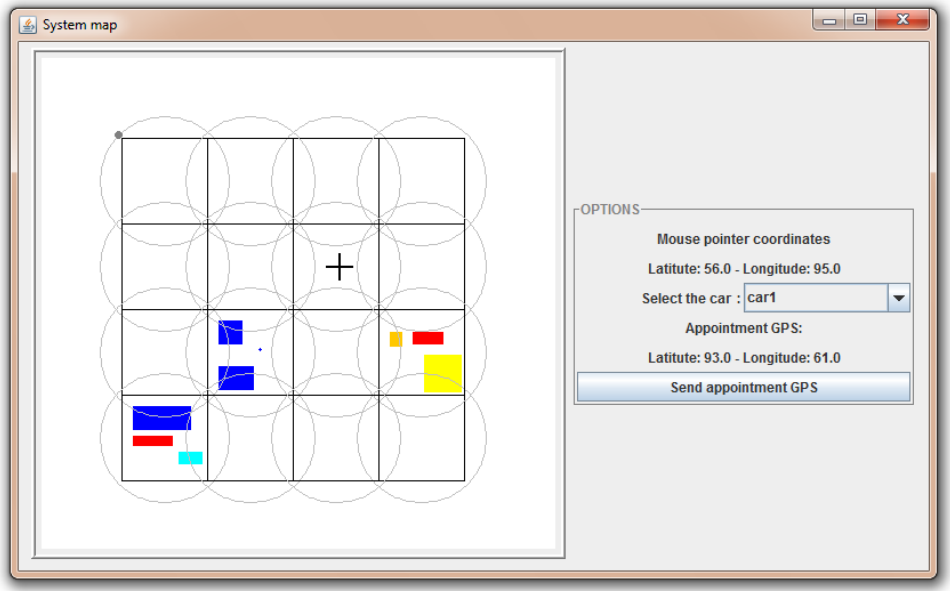


Fig. 4.8: Una modalità di inserimento dell'appuntamento

sione al sistema e la creazione del rispettivo agente di gestione. Sempre dalla figura è possibile vedere le altre funzioni della macchina: vengono presentate le coordinate GPS attuali oltre allo stato di connessioni bluetooth con altri dispositivi. Nel contesto dell'esempio presentato l'automobile non risulta collegata ad alcun elemento esterno, come può essere uno smartphone.

La ricerca del parcheggio più conveniente disponibile nei pressi di un appuntamento può essere effettuata in due modi anche se si possono immaginare in futuro altre realizzazioni come ad esempio l'interazione con l'agenda dello smartphone del guidatore. Una delle due modalità è rappresentato, come si osserva dalla figura 4.7, dall'inserimento manuale di latitudine e longitudine dell'appuntamento direttamente dalla GUI dell'automobile: questo tipo di inserimento può simulare l'interazione dell'utente mediante dispositivi integrati nell'auto come ad esempio un navigatore satellitare che inoltra le coordinate GPS di un punto selezionato dal guidatore. L'altra modalità è stata invece implementata per motivi pratici e consiste nella selezione del luogo di interesse mediante l'utilizzo del mouse nella schermata di gestione del manager di sistema. Questa realizzazione è possibile vederla in figura 4.8 consentendo di selezionare l'automobile di interesse e, con un click del mouse all'interno del sistema, le coordinate GPS dell'appuntamento. In questo modo vengono comunicate latitudine e longitudine all'agente `Car` che le utilizzerà poi come se fossero state inserite direttamente dalla sua GUI.

Per la gestione delle automobili è stato deciso di implementare un apposito agente che sfrutta una delle caratteristiche fondamentali dei sistemi agent-based, ovvero la mobilità. Essendo questo un progetto prevalente-

mente di simulazione è stato necessario avere degli strumenti che rendessero l'idea di uno spostamento all'interno del sistema anche per quanto riguarda le auto. L'agente `Car` sfrutta quindi queste possibilità muovendosi da un'area all'altra e in particolar modo da un distretto all'altro. Mentre per lo spostamento intradistrettuale viene interessato solamente l'agente che gestisce la connessione wifi dell'automobile, nella mobilità interdistrettuale viene interessato anche l'agente `Car` in modo diretto grazie alla sua migrazione dal container rappresentante il distretto lasciato a quello che identifica il quartiere d'ingresso.

Come già anticipato in precedenza, non appena viene instaurata una nuova connessione wifi con il sistema viene generato un nuovo agente di tipo `WifiConnection` che viene associato al dispositivo che si connette. In particolare la realizzazione effettiva prevede che il nome di questo agente contenga il nome locale dell'automobile che si connette: se c'è l'agente generico `MyCar` che si connette al sistema allora verrà creato il relativo gestore della connessione con nome `Wifi_MyCar`. Oltre ad essere associato al dispositivo vengono memorizzate anche altre informazioni quali l'area corrente in cui l'entità si trova e i suoi vicini, le coordinate GPS attuali del dispositivo e dell'area di riferimento.

Questo agente ha il compito di gestire la connessione wifi e per farlo va ad analizzare frequentemente la potenza del segnale. Poiché in questa fase di progetto ci si trova a trattare il problema in simulazione, è stato necessario simulare il comportamento del modulo wifi che percepisce automaticamente la presenza di altri segnali diversi da quello corrente e effettua il passaggio da una cella di servizio a un'altra in modo trasparente all'utilizzatore. In fase implementativa è stato quindi pensato di gestire questa questione sfruttando le informazioni che un agente `Area` possiede riguardo ai suoi vicini. Non appena l'agente gestore di connessione viene creato, gli vengono passati anche i *neighbour* dell'area associata cosicché, in fase di controllo del segnale, possa utilizzarne i dati. Il controllo, infatti, avviene internamente all'agente mediante un behaviour che estende la classe `TickerBehaviour` e viene effettuato sfruttando il modello scelto per il segnale wifi e le conoscenze di base di geometria. Ricordando che l'equazione di un cerchio di raggio r e di centro (x_0, y_0) è:

$$(x - x_0)^2 + (y - y_0)^2 \leq r^2$$

si può analizzare l'appartenenza o meno a un preciso cerchio noti il raggio e il centro andando a controllare se la disuguaglianza è verificata oppure no. Le circonferenze che definiscono le coperture wifi delimitano quindi i cerchi di interesse. L'agente `WifiConnection` mantiene traccia della potenza corrente del segnale andando ad aggiornarla ad ogni invocazione del controllo. Il procedimento prevede l'analisi dell'appartenenza ai cerchi associati all'area corrente e a quelli relativi ai vicini: tra tutte le disuguaglianze verificate

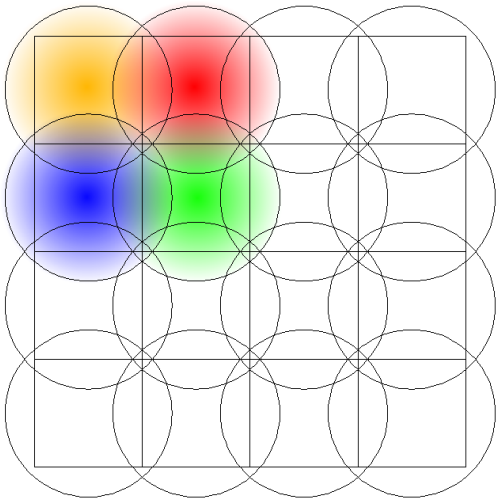


Fig. 4.9: La sovrapposizione del segnale wifi

viene preso come nuovo valore di potenza quello che rende massimo il valore percentuale:

$$\left(1 - \frac{(x - x_0)^2 + (y - y_0)^2}{r^2}\right) \cdot 100$$

in cui (x_0, y_0) e r rappresentano rispettivamente le coordinate GPS del centro area e il raggio visto come la distanza dal centro al vertice dell'area più l'ϵ scelto a piacere per la sovrapposizione di segnale. Se il segnale maggiore arriva da un'area diversa da quella corrente, significa che l'automobile ha cambiato zona. Si possono così identificare due diversi casi: lo spostamento può essere intradistrettuale oppure interdistrettuale. Mentre nella prima mobilità vengono aggiornati solo i riferimenti interni all'agente `WifiConnection` modificando informazioni quali l'area, le sue coordinate e i suoi vicini, nel secondo è prevista anche la sua migrazione nel nuovo distretto di interesse e l'invio di un messaggio all'agente `Car` associato per comunicare la necessità del suo spostamento. Da figura 4.9 si può capire come, per un sistema con le aree della stessa dimensione, il cambiamento di area venga rilevato al passaggio del confine. Si evidenzia che il segnale wifi di due aree contigue risulta essere coincidente sulla riga di separazione, pertanto uno scostamento δ piccolo a piacere in una direzione causa l'appartenza alla copertura di un'area piuttosto che a un'altra.

4.2.6 I dispositivi mobili

Tra i componenti del sistema emergono anche i dispositivi mobili, quali smartphone o cellulari di ultima generazione. Questi elementi, infatti, consentono di ampliare il pacchetto informativo a supporto dell'utente mediante l'utilizzo di un agente appositamente creato e di fornire ulteriori funzionali-

tà all'infrastruttura realizzata. Nello specifico è stato utilizzato uno smartphone dotato di Android 2.1 che ha reso necessario l'impiego di un add-on per JADE, JADE ANDROID. L'estensione di LEAP utilizzata, che risulta specifica per questo sistema operativo, offre allo sviluppatore la possibilità di ottenere l'integrazione di applicazioni per l'ambiente mobile con la piattaforma JADE.

L'agente implementato è inglobato in un'applicazione Android che è stata installata nel dispositivo ed interagisce, mediante l'utilizzo di particolari classi, con le `Activity`. Lo sviluppo dell'agente per il telefono è risultato più impegnativo in quanto, al fine di utilizzare le funzionalità offerte da JADE anche in ambiente Android, è stato necessario gestire i seguenti aspetti, come presentato in [34]:

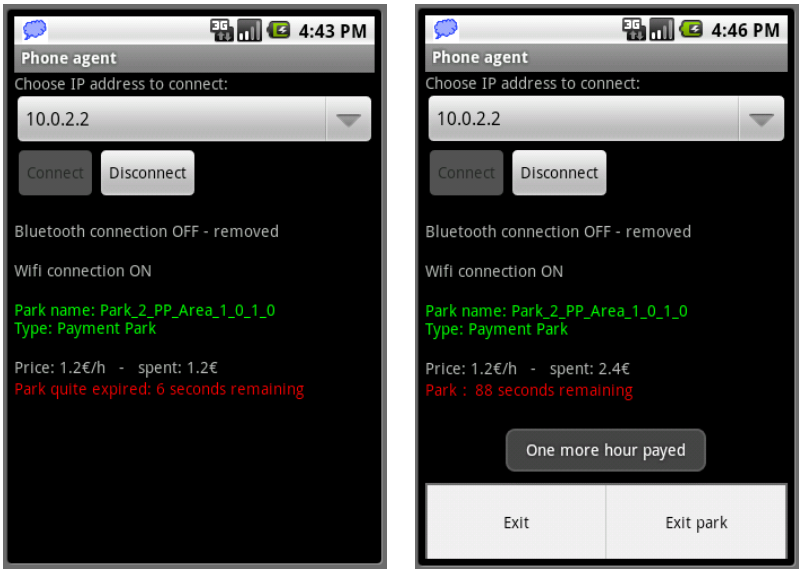
- L'agente, chiamato nel progetto `PhoneAgent`, deve estendere la classe `jade.wrapper.gateway.GatewayAgent`;
- Deve essere implementato il comportamento specifico dell'agente mediante i vari behaviour e deve esser sovrascritto il metodo `processCommand` inserendo la logica dell'applicazione;
- L'`Activity` deve implementare `jade.android.ConnectionListener`;
- L'`Activity` deve chiamare il metodo `JadeGateway.connect` passando i parametri opportuni;
- Nell'`Activity` deve essere implementato il metodo `onConnected` per ottenere l'istanza del `JadeGateway`;
- Per inviare un comando all'agente si deve utilizzare il metodo `execute` dell'istanza di `JadeGateway`;
- Per chiudere la connessione è necessario chiamare il metodo `disconnect`.

Oltre ai punti appena presentati risulta necessario effettuare anche un'aggiunta al file `AndroidManifest.xml` andando a inserire la riga che identifica un servizio utilizzato dall'add-on JADE ANDROID e che viene presentata nel codice 4.4.

```
<service android:name=``jade.android.MicroRuntimeService">
```

Codice 4.4: MicroRuntimeService nel file manifest

L'agente del telefonino, in questa prima fase del progetto, realizza solamente poche funzioni lasciando ampio margine di crescita nelle prossime versioni. All'attivazione dell'applicazione sul dispositivo, l'agente si registra alla piattaforma JADE e successivamente procede con la ricerca del servizio bluetooth dell'automobile del guidatore. Avendo nella realtà a disposizione



(a) Ora in scadenza **(b)** Timer reinizializzato

Fig. 4.10: Due situazioni dell'applicazione su Android

solamente uno smartphone è stato assunto che la macchina associata al telefonino diventa quella che ha registrato il servizio `BLUETOOTH_MOBILE` e che si trova nella prima posizione nel risultato della ricerca al DF. Dopo questa fase che simula una connessione bluetooth, l'agente `Phone` rimane in attesa mantenendo un controllo periodico sullo stato della connessione grazie a un apposito behaviour.

Uno degli eventi che possono interessare il telefono riguarda l'arrivo al parcheggio dell'automobile associata. Come detto precedentemente, l'attenzione è stata posta su due particolari tipi di parcheggio che sono quello a disco orario e quello a pagamento. Assumendo per quest'ultimo che il guidatore possegga una carta prepagata, la gestione dell'arrivo dell'auto nel posto del parcheggio può esser effettuata allo stesso modo ad alto livello per entrambe le tipologie. Non appena viene notificato l'arrivo della macchina, si procede con il caricamento delle informazioni del parcheggio nel telefonino. La connessione wifi dell'automobile viene trasferita al telefonino mentre la connessione bluetooth che collega auto e dispositivo mobile viene chiusa. In questo modo viene garantita la continuità del servizio anche quando il guidatore si allontana dall'automobile o questa è spenta. La logica dell'applicazione prevede che all'arrivo su un posteggio a disco orario venga trasferito un agente sul telefonino che consiste in un timer indicante il tempo rimanente possibile per il parcheggio dell'auto. Un analogo ragionamento viene fatto per il parcheggio a pagamento, assumendo però che il timer viene inizializzato ogni ora aggiornando il costo totale della fermata. Nella realtà,

il concetto di agente mobile che viene trasferito sul telefono cellulare non è stato possibile implementarlo a causa della diversità della virtual machine presente in ambiente Android rispetto a quella su computer. Per questo limite pratico, quindi, non è possibile far migrare l'agente che funge da timer: il problema è stato comunque aggirato simulando un comportamento di mobilità. All'arrivo nel posteggio, l'agente `WifiConnection` invia all'agente `Phone` un behaviour sfruttando la classe `LoaderBehaviour` appositamente modificata per gestire il problema in ambiente Android. I parametri inviati, diversi a seconda del tipo di parcheggio, consentono di avviare il timer in modo corretto nel telefonino simulando così la mobilità dell'agente con una mobilità di codice binario. Questo risulta comunque l'unica modalità realizzativa al momento di questa implementazione, come confermato nel forum dedicato agli sviluppatori JADE [35].

In figura 4.10 vengono presentate due schermate dello smartphone che rappresentano due istanti diversi nel caso di avvenuto parcheggio con tipologia a pagamento: prendendo in esame la figura 4.10a si osserva che il tempo equivalente alla prima ora si sta esaurendo, come viene confermato dalla scritta rossa. Al termine dei secondi rimanenti, l'importo pagato aumenta e il timer viene aggiornato e fatto ripartire: è questo il caso della figura 4.10b. Da queste immagini si riscontra l'effettiva chiusura della connessione bluetooth e dell'attivazione del wifi nel sistema, come confermato dalle scritte a video. Nella seconda figura, inoltre, è possibile osservare la presenza anche del menu costituito da due bottoni che consentono rispettivamente di chiudere l'applicazione e di simulare l'uscita dal parcheggio. Quest'ultima opzione è stata aggiunta solamente per scopi di test e di verifica del corretto funzionamento del processo di gestione dei parcheggi.

4.3 Le interazioni

Il sistema prevede un certo numero di scambi di messaggi che possono essere identificati come delle particolari interazioni mirate al raggiungimento di certi obiettivi specifici. Nel complesso, le entità coinvolte nello scambio dei messaggi sono tutte quelle presentate in precedenza sottolineando come l'intero sistema sia fortemente connesso e come le funzionalità fornite richiamino l'intervento di più di un'unità per essere concretizzate. Qualche scambio di messaggi avviene, come già anticipato nelle sezioni precedenti, solamente per consentire all'infrastruttura di funzionare correttamente in simulazione in quanto, nella realtà, esistono già dei moduli che realizzano tali concetti, come nel caso della connessione wifi. In seguito vengono presentate le interazioni di maggior interesse per il sistema entrando nel dettaglio dei singoli messaggi inviati.

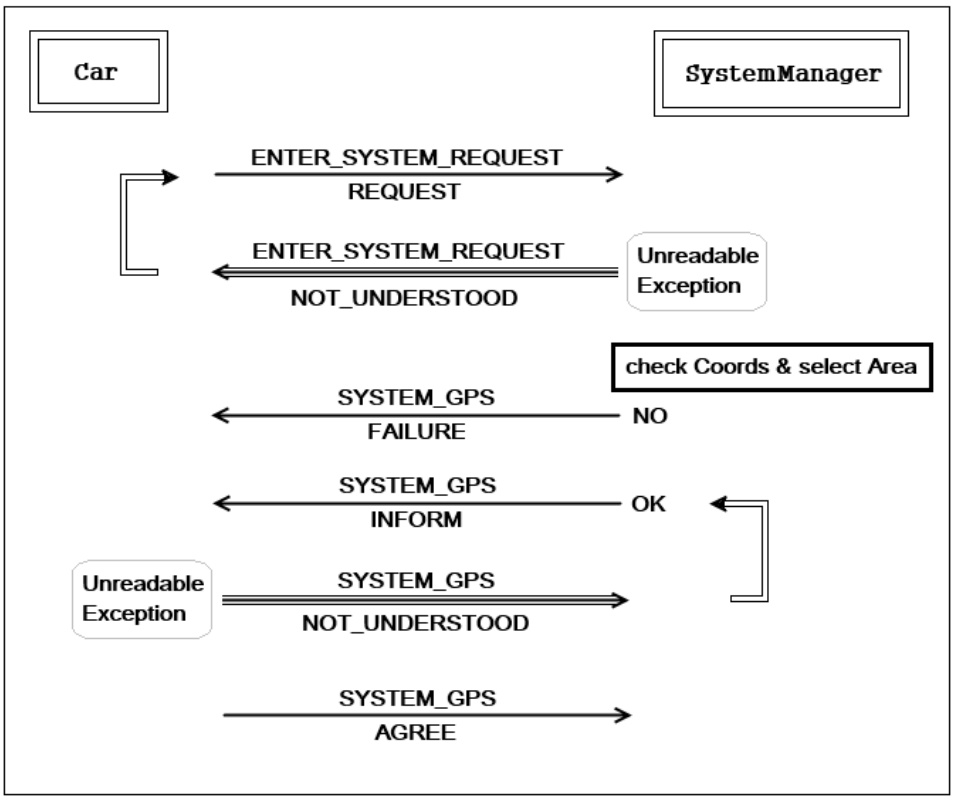


Fig. 4.11: L'interazione di accesso al sistema

4.3.1 L'accesso al sistema

Questa interazione avviene non appena un'automobile percepisce il sistema oppure quando essa viene accesa e si trova al suo interno. Lo schema dello scambio dei messaggi è presentato in figura 4.11 e le entità coinvolte sono l'agente `Car` e il `SystemManager`. L'iniziativa viene presa dall'automobile che invia un primo messaggio di richiesta d'accesso al sistema contenente le proprie coordinate GPS. Il manager di sistema analizza il pacchetto: nel caso in cui i dati ricevuti risultassero corrotti, l'agente risponde con un messaggio che ha come valore per il campo *performative* `NOT_UNDERSTOOD` comportando, alla ricezione di questo messaggio da parte della macchina, il reinvio del messaggio di richiesta. Se invece la lettura del messaggio va a buon fine, il `SystemManager` procede con la verifica di appartenenza al sistema delle coordinate ricevute: in caso di mismatch verrà notificato l'evento mediante un messaggio di tipo `FAILURE` all'auto che interromperà il processo di accesso al sistema altrimenti il manager di sistema inizia la ricerca dell'`Area` a cui le coordinate appartengono percorrendo l'albero memorizzato al suo interno. Al termine viene inviato un messaggio di tipo `INFORM` che comunica all'auto le coordinate del sistema e l'`AID` dell'area a cui appartiene.

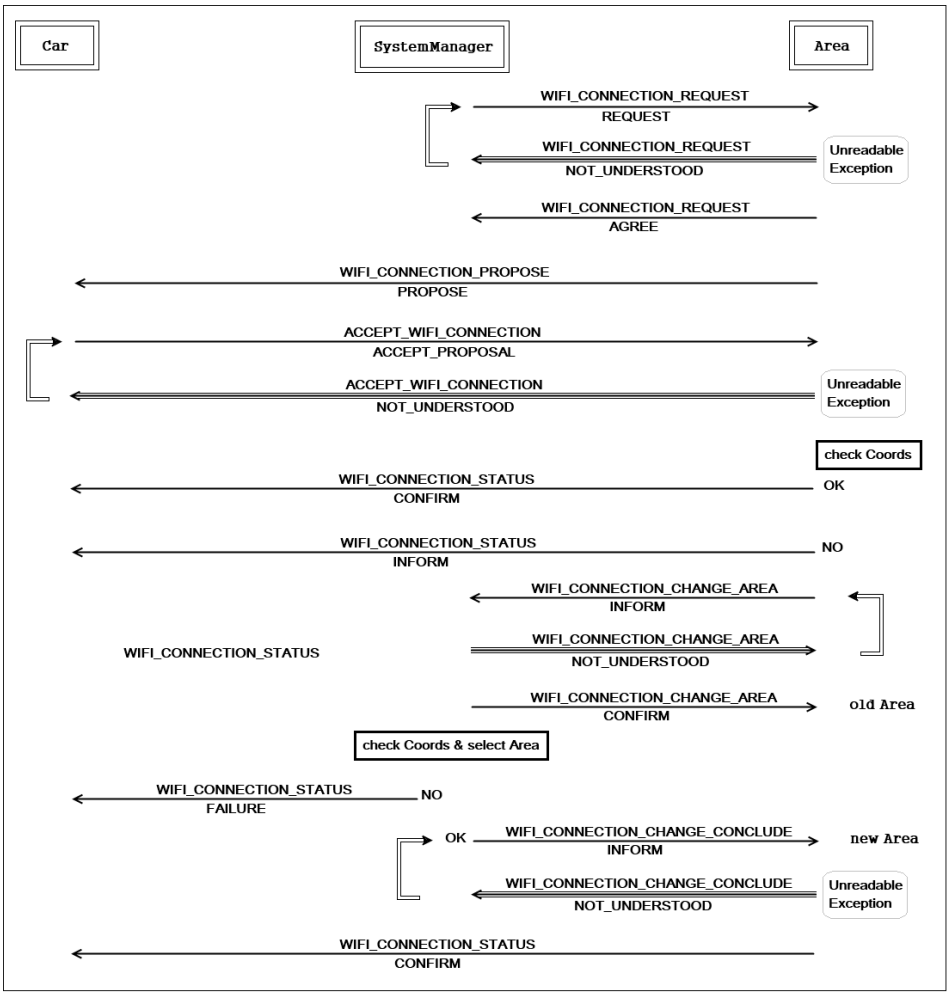


Fig. 4.12: La connessione wifi

Come in precedenza, se i dati ricevuti sono illeggibili, la macchina risponde al `SystemManager` con un messaggio `NOT_UNDERSTOOD`, richiedendo quindi il reinvio delle informazioni, altrimenti invia un ack di conferma andando così a chiudere il processo di accesso al sistema. Dopo il messaggio `AGREE`, infatti, l'automobile risulta autenticata nel sistema.

4.3.2 L'avvio di una connessione wifi

Il sistema prevede, oltre all'accesso ad esso, anche una connettività mediante l'instaurazione di una connessione wireless con un'area specifica. In particolare la connessione può avvenire quando si richiede l'accesso al sistema, funzionalità questa implementata di default nel progetto, oppure in fasi successive dopo ad esempio aver riattivato il modulo wifi del dispositivo.

Analizzando l'avvio di una connessione in fase di ingresso al sistema si può prendere in considerazione la figura 4.12 per esplicitare meglio le varie fasi dell'interazione. L'iniziativa viene presa dal `SystemManager` che invia una richiesta di connessione, contenente l'identificativo del dispositivo che si vuole connettere, all'area, che è il risultato della ricerca effettuata nella fase precedente di accesso al sistema. Analogamente a quanto presentato nella sezione 4.3.1, nel caso in cui i dati non fossero chiaramente leggibili viene inviato in risposta un messaggio `NOT_UNDERSTOOD` che richiede il reinvio delle informazioni di richiesta. Se invece la lettura va a buon fine, l'agente responsabile dell'area risponde con un `AGREE` al `SystemManager` ed inizia un'interazione diretta con l'automobile inviando una proposta di connessione mediante un messaggio `PROPOSE`. Questa accetterà la proposta rispondendo con un `ACCEPT_PROPOSAL` nel quale inserisce nuovamente le proprie coordinate GPS al fine di poter verificare la corretta appartenenza dell'auto alla specifica area. Viene gestita anche in questo caso l'eccezione `UnreadableException` mediante l'invio di un messaggio `NOT_UNDERSTOOD`. Se invece la lettura è corretta viene fatto un check sulle coordinate: a questo punto l'interazione per l'avvio della connessione può terminare positivamente mediante l'invio di un messaggio di tipo `CONFIRM` se le coordinate dell'automobile sono contenute nell'area geografica dell'area, altrimenti la connessione non risulta attiva ed inizia una seconda fase dell'interazione. La situazione in cui l'auto non appartiene più all'area cercata dal `SystemManager` in fase di accesso non è remota, in quanto la macchina può trovarsi in movimento e quindi, portando all'estremo il caso, trovarsi all'inizio della fase di accesso in un'area e alla ricezione della proposta di connessione in un'altra. Per questo è stato introdotto questo tipo di controllo che simula l'automatico spostamento di richiesta di connessione wifi da una cella ad un'altra. Nello specifico, questo comportamento è stato ottenuto mediante l'utilizzo di una serie di messaggi ulteriori. Innanzitutto l'area che ha verificato il mismatch delle coordinate invia all'automobile un messaggio di tipo `INFORM` nel quale è contenuta l'informazione di cambio area di riferimento. Contemporaneamente, manda anche un messaggio con l'ontologia `WIFI_CONNECTION_CHANGE_AREA` al `SystemManager` per comunicare la richiesta di cambio area di gestione per l'automobile il cui riferimento è contenuto in un parametro del messaggio stesso. Gestendo sempre l'eventuale eccezione in fase di lettura dei dati, il manager di sistema procede con la stessa verifica che fa in fase di accesso al sistema, ovvero controlla se le coordinate GPS dell'automobile appartengono all'area definita dal sistema e, in caso positivo, cerca la zona di appartenenza. Indipendentemente dall'esito di questo check, viene inviato un messaggio di conferma all'area che ha richiesto il cambio gestione al fine di aggiornare i dati di connessione interni. La motivazione di questa comunicazione, come anche dei messaggi precedenti di AGREE in fase di accesso al sistema e all'inizio della fase di connessione, è che le entità infrastrutturali mantengono all'interno delle strutture dati con le richieste pendenti associate a un parti-

colare id di conversazione, oltre che quelle definitive. Per questo è necessario avere uno strumento che aggiorni tali dati che, nello specifico, coincide con l'invio di questi messaggi. Se la funzione di verifica risponde negativamente allora significa che l'automobile è uscita dai confini del sistema e per questo le viene comunicato il `FAILURE` della connessione. Nel caso in cui, invece, la risposta risulta essere positiva, il `SystemManager` invia un messaggio di tipo `INFORM` per comunicare alla nuova area di interesse l'apertura di una connessione: alla ricezione, la nuova area non procede più con il controllo né con l'invio della proposta all'auto, ma le comunica direttamente la conferma di avvenuta connessione. A questo punto viene creato l'agente `WifiConnection` che va a gestire tutti gli aspetti della connessione, dalla mobilità alla comunicazione tra sistema e automobile.

È da sottolineare come questo tipo di interazione sia comunque utile in questa fase del progetto per simulare un comportamento wifi all'interno dell'infrastruttura. Nella realtà si possono utilizzare le tecnologie e i protocolli impiegati nei moduli wifi senza dover utilizzare completamente l'interazione descritta in questa sezione. Per questo motivo è stato implementato uno scambio di messaggi che realizzasse ad alto livello il concetto di connessione wifi, senza andare a gestire nel dettaglio elementi quali la crittografia ed elementi di sicurezza in genere.

4.3.3 La gestione delle connessioni: il keepalive

Simulare una connessione significa simularne anche il controllo: nel progetto è stato deciso di implementare un meccanismo di keepalive per mantenere traccia delle connessioni attive e per poter liberare risorse nel caso in cui qualche collegamento risultasse per troppo tempo inattivo. In particolare, l'agente `WifiConnection` invia periodicamente, con periodo scelto a piacere, un messaggio di keepalive sia al dispositivo ad esso associato, che può essere quindi uno smartphone o l'automobile, che all'area di interesse. Questo consente ai due agenti riceventi di aggiornare i propri dati, in particolare il timestamp, nelle strutture dati riservate e, mediante un apposito behaviour, andare a verificare se la connessione è ancora stabile oppure se non è stata rilevata alcuna attività in un arco temporale definito. Per quanto riguarda un agente `Area` generico, riceverà un numero elevato di keepalive che vanno ad aggiornare i dati al suo interno. Questi dati vengono poi inoltrati al distretto di appartenza attraverso un `TickerBehaviour` di periodo maggiore rispetto a quello dell'agente `WifiConnection`. In questo modo, ricevendo in un blocco solo tutti i dati di un'area, un quartiere ha la possibilità di procedere con l'aggiornamento dei propri dati eseguendo le eventuali operazioni di modifica, inserimento e cancellazione. I dispositivi connessi al distretto vengono infine comunicati anche al `SystemManager` sempre sfruttando un altro behaviour periodico di periodo ancora superiore rispetto

ai precedenti. Anche in questo caso l'agente riceverà tutte le informazioni relative ai vari distretti andando ad apportare gli opportuni aggiornamenti.

La struttura di questo meccanismo è stata pensata per alleggerire il carico sulla rete cercando di avere comunque un occhio di riguardo verso la sincronizzazione dei dati all'interno del sistema. È stato considerato ragionevole, quindi, incrementare il periodo di invio dei keepalive all'aumentare dell'entità di riferimento. Inoltre è stato deciso, a partire dagli agenti `Area`, di inviare con un unico messaggio tutte le informazioni relative alle connessioni sfruttando la possibilità di allegare un contenuto sotto forma di oggetto java, purché questo sia serializzabile.

4.3.4 L'aggiornamento di stato dei parcheggi

Il mantenimento di informazioni sincronizzate all'interno dell'intero sistema deve toccare necessariamente anche la gestione dello stato dei parcheggi. Per realizzare questa funzione, ogni agente di tipo `Park` tra i sei presentati nella sezione dedicata è dotato di un `TickerBehaviour` che periodicamente invia all'area di appartenenza le informazioni inerenti alla propria situazione attuale. Viene sfruttata anche in questo caso la possibilità di inglobare in un messaggio un contenuto sottoforma di oggetto serializzabile infatti al messaggio con ontologia `PARK_STATUS` e communicative act `INFORM` viene aggiunto un *content object* che consiste in una istanza di `Object[3]` e in particolare:

- *status*: questo campo assume un valore tra `PARK_FULL`, `PARK_FREE` e `PARK_QUITE_FULL`;
- *freePlaces*: indica il numero preciso di postazioni libere del parcheggio;
- *quiteFreePlaces*: questo campo risulta indispensabile per comunicare il numero di posti che si liberano in un arco temporale definito e personalizzabile. Viene utilizzato in particolare dai parcheggi di tipo disco orario e per quelli a pagamento se non utilizzati con la scheda prepagata: non si esclude però un'estensione di utilizzo a tutte le tipologie andando magari ad interessare il calcolo statistico e probabilistico per avere una stima su un preciso arco di tempo. Per questo motivo il campo può assumere un valore numerico oppure `null`.

Al lato ricevente, l'agente `Area` che riceve tale messaggio lo processa e ne utilizza i dati in esso salvato per andare ad aggiornare le informazioni memorizzate all'interno di un'apposita struttura dati che contiene, oltre ai valori ricevuti con il messaggio, anche dati relativi al nome, al tipo, alle coordinate, al suo centro e a valori tipici della tipologia, come ad esempio la durata del parcheggio del disco orario o la tariffa oraria di uno a pagamento. Questa particolare struttura dati viene utilizzata ampiamente durante la fase di ricerca di un parcheggio.

4.3.5 La richiesta di un parcheggio

La ricerca di un parcheggio da parte di un'automobile sta alla base della logica dell'infrastruttura implementata ed è stata realizzata mediante uno scambio di messaggi che vede coinvolte diverse entità, a partire dalla macchina e all'agente che gestisce la sua connessione wifi fino alle aree e al `SystemManager`.

In questa fase del progetto l'appuntamento che fa da punto di riferimento per la ricerca di un parcheggio viene inserito manualmente direttamente dalla GUI dell'agente `Car` oppure cliccando nella finestra della mappa del sistema appartenente al `SystemManager`. Sviluppi futuri prevederanno l'integrazione con l'agenda del telefonino Android che indicherà le coordinate GPS del luogo inserito.

La richiesta, in questa prima implementazione, viene quindi assunta come la lettura di un dispositivo integrato all'automobile, come può essere un navigatore satellitare, e viene inviata all'agente che gestisce la connessione wifi il quale la inoltra all'area collegata. A questo punto l'area assegna alla richiesta un particolare id che viene utilizzato per gestire eventuali richieste successive da parte dello stesso dispositivo: in questo modo, infatti, alla macchina verrà data risposta solo per l'ultima richiesta effettuata mentre tutte le altre fatte in precedenza e non ancora soddisfatte vengono scartate. L'area associata all'agente della connessione wifi verifica poi se l'appuntamento per cui si vuole trovare un parcheggio nei dintorni si trova entro i suoi confini oppure se appartiene ad un'area diversa. Nel primo caso viene controllato se esiste almeno un parcheggio libero al suo interno: se ce n'è almeno uno viene selezionato il *più conveniente*, conformemente alla politica di convenienza adottata, e comunicato alla macchina passando per l'agente `WifiConnection`; altrimenti la richiesta viene inoltrata per esser gestita da una delle aree vicine che viene scelta in modo random tra le possibili. Se questa trova un parcheggio restituisce i rispettivi dati che verranno poi comunicati all'agente wifi e, in secondo luogo, all'automobile, altrimenti viene ritornato il valore `null` che comporta l'inoltro della richiesta a un'altra area contigua all'area originale, scelta sempre in modo casuale. Questo procedimento continua fino a che non viene trovato un parcheggio disponibile oppure tutte le aree vicine sono state interrogate: in quest'ultima situazione verrà comunicato alla macchina, passando sempre per l'agente `WifiConnection`, l'indisponibilità del posto auto nei pressi dell'appuntamento scelto.

Nel caso in cui, invece, le coordinate GPS non appartengono all'area in cui si trova la macchina, la richiesta viene inoltrata al `SystemManager` che ha il compito di individuare l'area di interesse analizzando la struttura ad albero memorizzata al suo interno. La funzione di ricerca è la medesima di quella che viene utilizzata in fase di accesso al sistema quando viene localizzata l'automobile. Dopo aver trovato quindi l'`Area` di competenza le invia la richiesta di parcheggio comunicando anche i dati dell'agente `WifiConnec-`

`tion` in modo da tenere traccia di eventuali richieste multiple. Una volta ricevuta la richiesta, il procedimento è del tutto analogo a quanto presentato nel caso dell'appuntamento nell'area collegata: l'unica eccezione è che il risultato non può essere comunicato direttamente all'agente responsabile della connessione ma deve invece passare per il `SystemManager` che la inoltrà opportunamente all'area di interesse sfruttando una struttura dati apposita che memorizza le richieste pendenti. Nel caso in cui la risposta non risultasse essere `null`, il pacchetto informativo riguardante il parcheggio viene salvato sia dall'agente `Car` che da quello `WifiConnection`.

4.3.6 L'arrivo a un parcheggio

Quando un'automobile arriva nel parcheggio individuato dalla ricerca inizia uno scambio di messaggi e l'esecuzione di alcune operazioni che consentono di registrare l'evento e di comunicare all'utente, tramite il cellulare, alcune informazioni. Essendo un progetto in simulazione è stato necessario realizzare un meccanismo che simulasse l'arrivo dell'auto e questo è stato fatto attraverso l'invio, da parte dell'agente `Car`, di un messaggio al gestore della connessione indicando di essere arrivato nel parcheggio memorizzato al suo interno. Oltre a questa informazione viene comunicato anche l'eventuale dispositivo mobile collegato in bluetooth con la macchina. L'agente `WifiConnection`, avendo memorizzato al suo interno le informazioni dell'ultima ricerca del parcheggio, ha la possibilità di inviare direttamente al telefonino un particolare behaviour che simula la mobilità degli agenti non realizzabile per JADE ANDROID. Il behaviour che viene inviato è appositamente compilato per essere eseguito dalla Dalvik virtual machine di Android e contiene, a seconda del tipo di parcheggio, diversi dati. Nel caso specifico della tesi, avendo preso in maggiore considerazione i parcheggi di tipo disco orario e a pagamento, il comportamento inviato consiste nell'attivazione di un timer che gestisce il tempo rimanente utile al parcheggio. Ciò che viene fatto, poi, è trasferire la connessione wifi al sistema dall'automobile al telefonino, assicurando una continuità di servizio. Per questo, oltre a inviare il behaviour, l'agente `WifiConnection` viene associato al dispositivo mobile e la connessione bluetooth tra telefono e automobile viene chiusa. Non è stata prevista la creazione di un nuovo agente di tipo `WifiConnection` ma viene gestito solamente l'aggiornamento del dispositivo associato. In questo modo si può comunque simulare il movimento concreto del telefonino nello spazio mediante la migrazione da un container ad un altro secondo la stessa modalità di quando è associato all'automobile. Diversamente però da questa situazione l'agente `Phone` non può migrare per i limiti dovuti alla diversità di VM tra Android e la JVM classica. Il risultato di quanto accade è stato presentato nelle figure 4.10 nelle quali è possibile vedere le schermate dello smartphone al termine dello scambio di messaggi e dell'esecuzione delle operazioni all'arrivo dell'automobile nel parcheggio.

CAPITOLO 5

Conclusioni e sviluppi futuri

L'obiettivo principale della tesi era dimostrare come l'utilizzo delle più recenti tecnologie per realizzare i sistemi ubiqui non offre solo una possibilità implementativa ma anche che diventa un vero e proprio punto di forza in termini di efficienza, di affidabilità e di efficacia dei sistemi stessi. Questo è stato realizzato mediante il paradigma ad agenti che ha permesso di sfruttare a pieno tutte le caratteristiche e i vantaggi che esso mette a disposizione.

L'aver utilizzato una piattaforma ad agenti mobili come base per un sistema ubiquo ha consentito, quindi, di semplificare lo sviluppo dell'applicazione distribuita che è composta da entità autonome che necessitano di comunicare, di interagire, di collaborare e di negoziare tra loro al fine di raggiungere il corretto funzionamento dell'intero sistema. Ciascun agente controlla in particolare il proprio thread di esecuzione e, per questo, può essere programmato per avviare delle azioni che non richiedono un diretto intervento dell'utente. Questa caratteristica ha permesso di semplificare l'interazione machine-to-machine (M2M) e quindi di rendere questo aspetto comunicativo pressoché trasparente all'uomo.

Ciò che ha portato all'impiego degli agenti è comunque una delle proprietà fondamentali di queste entità: la mobilità. Ciascun componente ha la possibilità di migrare da un container ad un altro senza particolari vincoli, ad esclusione degli agenti da e verso dispositivi Android per i noti limiti dovuti alla differente VM. Il movimento degli agenti offre un nuovo modo di sfruttare la tecnologia a disposizione in quanto non è richiesto che un programma conosca tutto il pacchetto informativo per procedere con il calcolo, ma ha la possibilità così di migrare per raccogliere nei vari punti i dati di interesse. Il vantaggio non si ha solamente in termini di acquisizione di informazioni infatti si può sfruttare la mobilità anche per fare in modo che

un agente faccia proprie alcune azioni che vanno ad ampliare il pacchetto di servizi e funzionalità che può offrire. Oltre a questo, l'utilizzo degli agenti consente di avere a disposizione degli strumenti che permettono d'avere una migliore parcellizzazione del carico di lavoro e delle mansioni da svolgere.

Andando nello specifico, l'aver utilizzato JADE per lo sviluppo dell'infrastruttura di un sistema ubiquo in ambito della viabilità urbana ha permesso di sfruttare completamente tutte le caratteristiche presentate fornendo un valido supporto all'utilizzatore di tale sistema. L'esempio implementato, infatti, consente a un utente di essere guidato, secondo una particolare politica decisionale personalizzabile, verso un parcheggio di interesse nei pressi di un appuntamento o di un luogo da lui selezionato. La ricerca avviene in un modo del tutto trasparente e il guidatore ha la possibilità di accettare o meno il suggerimento dal sistema. Un'ulteriore prova dell'utilità di un tale sistema è offerto dai servizi che possono essere a disposizione dell'utente in modo non invadente: è questo il caso di popup o finestre che possono essere aperti nello smartphone all'accadere di determinati eventi, come ad esempio l'attivazione di un timer all'arrivo in un parcheggio. La realizzazione di questi servizi viene solitamente effettuata mediante agenti appositamente implementati oppure attraverso dei behaviour che gli agenti responsabili fanno propri.

Un ulteriore punto di forza offerto dall'utilizzo di una piattaforma agent-based, e nello specifico di JADE, è dato dalla versatilità. Viene fornito infatti un set di API omogeneo che risulta essere indipendente dalla rete sottostante e dalla versione Java a disposizione, coprendo gli ambienti J2EE, J2SE e J2ME. In particolare, considerando l'ambiente mobile, la piattaforma consente una facile integrazione anche con i dispositivi Android sfruttando l'add-on specifico. L'interazione con questi dispositivi risulta essere piuttosto semplice nonostante sia comunque da tener conto delle limitazioni dovute alla Dalvik VM. Questo rappresenta un ulteriore punto a favore di una piattaforma come JADE aprendo il sistema a dispositivi mobili dalle risorse limitate utilizzando gli specifici add-on.

Il progetto realizzato vede raggiungere gli obiettivi preposti andando a dimostrare, quindi, come l'utilizzo del paradigma ad agenti mobili nell'ambito dei sistemi ubiqui abbia una certa rilevanza dal punto di vista dell'efficienza, dell'efficacia e dell'affidabilità. Con l'infrastruttura d'esempio si è in grado, infatti, di sottolineare come le caratteristiche degli agenti vadano a facilitare la realizzazione del concetto di ubiquità.

Alcuni aspetti, però, sono ancora da trattare come ad esempio la sicurezza. In questa prima realizzazione questo concetto è stato toccato molto superficialmente lasciando agli sviluppi futuri un'analisi più dettagliata e approfondita mediante l'utilizzo, ad esempio, dell'estensione JADE-S. Questo add-on, infatti, offre una sorta di strato aggiuntivo allo scopo di proteggere gli agenti e le piattaforme da attacchi riguardanti l'autenticazione, l'autorizzazione, l'integrità dei messaggi e la loro riservatezza. In particolare questi

ultimi obiettivi possono essere raggiunti utilizzando dei Login Module che implementano diverse modalità di login per l'aspetto di autenticazione, dei particolari file contenenti le policy che autorizzano o meno l'esecuzione delle azioni di un agente e la crittografia e la firma digitale per l'integrità e la confidenzialità dei messaggi.

Nello specifico dell'esempio implementato, poi, è possibile estendere le funzionalità offerte mediante la realizzazione e la gestione di altri servizi. Tra questi si può pensare, ad esempio, all'estensione delle tipologie dei parcheggi, aggiungendo a quelli già esistenti gli ibridi, quelli di carico/scarico o quelli per taxi, oppure a un diverso modo di gestire il pagamento oltre alla scheda prepagata. Altri sviluppi futuri, che farebbero emergere l'utilità della piattaforma ad agenti mobili, potrebbero riguardare l'implementazione di un ulteriore strumento di supporto in termini di viabilità che si integra con il sistema già realizzato. Questo potrebbe consentire di ricalcolare il percorso dinamicamente a seconda delle informazioni sul traffico inviate da un agente che precede l'automobile al fine di evitare inutili ingorghi. Essendo questo lavoro di tesi prevalentemente di simulazione non è stato gestito il planning stradale, aspetto sicuramente da aggiungere nelle future realizzazioni. Un'ulteriore possibilità di estensione arriva dall'integrazione del sistema con gli enti commerciali andando, per esempio, ad implementare un sottosistema di gestione della ricerca di punti di ristorazione nei pressi di un appuntamento o di determinate coordinate GPS. Tutte queste ipotesi di sviluppo futuro mettono in luce la grande varietà applicativa a cui è soggetto un tale sistema. L'utilizzo del paradigma ad agenti risulta essere uno strumento necessario per realizzare tutto ciò richiedendo la progettazione e la suddivisione dei ruoli e delle mansioni tra gli agenti costituenti il sistema.

Non può essere comunque trascurato, in futuro, il problema dell'integrazione che si ha con Android. Nonostante risulti tutto sommato piuttosto semplice far interagire un cellulare che monta questo sistema operativo con la piattaforma JADE, non è risolto il problema della mobilità degli agenti. Sviluppi futuri, problemi di VM a parte, potrebbero consentire l'interazione diretta con l'agenda dello smartphone in modo da rendere ancor più automatica e trasparente all'utente la ricerca del parcheggio. In queste estensioni entra in gioco pesantemente anche l'aspetto sicurezza poiché vengono trattati i dati sensibili delle persone.

Ringraziamenti

Un affettuoso ringraziamento è rivolto ai miei genitori, a mio fratello Daniele e ai miei nonni che hanno reso possibile il raggiungimento di questo importante traguardo non solo per il sostegno economico ma anche per la costante presenza e comprensione.

Un grazie speciale va poi a Silvano che mi ha sempre aiutato nei momenti difficili sostenendomi con la sua vicinanza, incoraggiandomi in ogni circostanza e credendo sempre nelle mie capacità. Particolare riconoscenza è rivolta a Gioacchino, Walter e Martina per quanto fanno per me da quando ci conosciamo. Un ringraziamento generale, poi, a tutte quelle persone che mi sono state vicine in diverse occasioni e che hanno sempre creduto in me.

Voglio manifestare, infine, la mia gratitudine al mio relatore, il professor Carlo Ferrari, per la costante presenza e disponibilità durante tutto il periodo della tesi.

Bibliografia

[1] Andrew S. Tanenbaum, Maarten Van Steen, *Distributed Systems, principles and paradigms*, Pearson International Edition, 2nd edition, 2007.

[2] A. Hentout, B. Bouzouia, and Z. Toukal, *Multi-agent architecture model for riving mobile manipulator robots*, International Journal of Advanced Robotic Systems, 2005, 5(3), Pages 257-268.

[3] L. Kaddour el Boudadi, J. Vareille, P. Le Parc, and N. Berrached, *Remote control on internet, long distance experiment of remote practise works, measurements and results*, International Review on Computers and Software (IRECOS), ISSN 1828-6003, May 2007.

[4] J. Mayfield, Y. Labrou and T. Finin, *Evaluating KQML as an Agent Communication Language*, in M. Wooldridge, J. P. Müller and M. Tambe (eds), Intelligent Agents II (LNAI 1037), Springer Verlag, Heidelberg, Pages 347-360.

[5] Tim Kindberg and Armando Fox, *System Software for Ubiquitous Computing*, IEEE Pervasive Computing, Volume 1, Issue 1, January 2002, Pages 70-81.

[6] Seokhee Jeon, Jane Hwang, Gerard J. Kim and Mark Billinghurst, *Interaction with large ubiquitous displays using camera-equipped mobile phones*, Personal Ubiquitous Computing, Volume 14, Issue 2, February 2010, Pages 83-94.

[7] Amara Touil, Etienne Pardo, Jean Vareille, Philippe Le Parc, *Toward a generic framework for ubiquitous system*, Workshops at the Grid and Pervasive Computing Conference, 2009.

[8] R. He and M. Lacoste, *Applying component-based design to self-protection of ubiquitous systems*, Proceedings of the 3rd ACM workshop on Software engineering for pervasive services, February 2008, Pages 9-14.

[9] M. Fukase, H. Takeda and T. Sato, *Hardware/software co-design of a secure ubiquitous system*, 2006 International Conference on Computational Intelligence and Security, November 2006, 2: 1307-1310.

[10] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, *Weak and Strong Mobility in Mobile Agent Applications*, Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000), Manchester (UK), April 2000.

[11] Fabio Bellifemine, Agostino Poggi and Giovanni Rimassa, *Developing Multi-agent Systems with JADE*, Lecture Notes in Computer Science, 2001, Volume 1986, Intelligent Agents VII Agent Theories Architectures and Languages, Pages 42-47.

[12] Fabio Bellifemine, Giovanni Caire e Dominic Greenwood, *Developing multi-agent systems with JADE*, Wiley, 2007.

[13] Fabio Bellifemine, Giovanni Caire, Giosuè Vitaglione, Giovanni Rimassa and Dominic Greenwood, *The JADE Platform and Experiences with Mobile MAS Applications*, Whitestein Series in Software Agent Technologies and Autonomic Computing, 2005, Software Agent-Based Applications, Platforms and Development Kits, Pages 1-20.

[14] Fabio Bellifemine, Federico Bergenti, Giovanni Caire and Agostino Poggi, *Jade - A Java Agent Development Framework*, Multi-Agent Programming / Multiagent Systems, Artificial Societies, And Simulated Organizations, 2005, Volume 15, II, Pages 125-147.

[15] Fabio Bellifemine, Giovanni Caire, Agostino Poggi and Giovanni Rimassa, *JADE: A White Paper*, Exp in search of innovation, Telecom Italia Lab, Volume 3 n.3, March 2004.

[16] Stephen R. Tate, Ke Xu, *Mobile Agent Security Through Multi-Agent Cryptographic Protocols*, Proceeding of the 4th International Conference on Internet Computing, 2003.

[17] Eleni Mangina, Javier Carbo, José M. Molina, *Agent-Based Ubiquitous Computing*, Atlantis Ambient and Pervasive Intelligence, Ismail Khalil, 2009.

[18] Mark D. Weiser, *Hot topic: Ubiquitous computing*, IEEE Computer, October 1993, Pages 71-72.

[19] Massimo Carli, *Android - Guida per lo sviluppatore*, Apogeo, 2010.

[20] CAR 2 CAR Communication Consortium Manifest, version 1.1, August 2007.

[21] CAR 2 CAR Communication Consortium, `http://www.car-to-car.org/`

[22] Comune di Treviso, `http://www.comune.treviso.it/`

[23] Trevisosta, `http://www.trevisosta.it/`

[24] Portale comune di Verona, `http://portale.comune.verona.it/`

[25] Xerox Ubiquitous Computing, `http://sandbox.xerox.com/ubicomp/`

[26] FIPA foundation, `http://www.fipa.org`

[27] FIPA Agent Management Specification, `http://www.fipa.org/specs/fipa00023/XC00023H.html`

[28] Comunicato stampa JADE Governing Board, `http://jade.tilab.com/jboard/PressReleaseBoardItalian.pdf`

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

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

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

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

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

[34] JADE ANDROID add-on guide, `http://jade.tilab.com/doc/tutorials/JADE_ANDROID_Guide.pdf`

[35] JADE develop forum, `http://avalon.tilab.com/pipermail/jade-develop/`