Alma Mater Studiorum
Università degli Studi di Bologna

Campus di Cesena
Scuola di Ingegneria e Architettura

Corso di laurea magistrale in Ingegneria Informatica

Titolo dell'elaborato

Coordinazione space-aware per dispositivi mobili in TuCSoN

Elaborato in

Sistemi Multi-Agente

Relatore
Chiar.mo Prof. Andrea Omicini

Correlatore
Dott. Ing. Stefano Mariani

Presentata da
Michele Bombardi

Sessione II - Anno Accademico 2012-2013

Introduzione

Negli ultimi anni, in particolare durante l'ultima decade, si è verificata una straordinaria evoluzione e diffusione tecnologica, soprattutto per quanto riguarda i dispositivi mobili. Il progresso scientifico nel campo dell'ingegneria hardware ha portato allo sviluppo di dispositivi intelligenti con potenzialità computazionali sempre crescenti e dimensioni sempre più ridotte.

La rapida diffusione di queste tecnologie, utilizzate dalle persone nella vita di tutti i giorni, ha profondamente cambiato la prospettiva dell'ingegneria del software, inducendola a supportare la comunicazione tra dispositivi eterogenei in continuo movimento ed immersi in un ambiente, anch'esso eterogeneo, la cui *topologia* è in continuo mutamento. É dunque nato un nuovo scenario applicativo, nel quale i sistemi computazionali sono dovunque, integrati nell'ambiente, sempre connessi tra loro tramite la rete wireless od operatore telefonico e sempre attivi, allo scopo di portare a compimento i compiti richiesti dagli utenti. Questo ambito è riconosciuto come *Pervasive Computing* e si basa sull'idea per la quale devono essere *le macchine ad adattarsi all'ambiente umano, piuttosto che forzare gli umani ad entrare nel loro.* Seguendo questo principio, le persone si potrebbero connettere tra loro direttamente, o tramite l'ambiente, al fine di acquisire informazioni utili riguardanti i loro interessi personali.

Questa prospettiva costringe lo sviluppatore a progettare nuove tecnologie software per garantire la corretta interazione tra dispositivi mobili, nonché tra essi e l'ambiente nel quale sono immersi. Inoltre, implica la creazione di sistemi complessi caratterizzati da numerosi elementi interconnessi tra loro, arricchiti con capacità computazionali complesse, aventi la necessità di coordinarsi ed interagire allo scopo di raggiungere obiettivi globali, oltre a quelli personali. In particolare, lo sviluppo tecnologico ha portato alla comparsa di dispositivi mobili dotati di sensori e coprocessori di movimento, aspetto che ha fatto nascere la necessità di poter progettare sistemi che fossero consapevoli del contesto spaziale nel quale si trovano, allo scopo di fornire all'utente funzionalità relative a posizione e movimento, e per permettere ai dispositivi di dialogare e coordinarsi in accordo con la loro posizione.

In generale, il concetto di spazio non può essere considerato singolarmente ma dev'essere parte integrante di un insieme di aspetti riguardanti il tempo che scorre, l'ambiente nel quale un dispositivo è immerso e la posizione nella quale esso si trova. Questi danno origine, nel loro insieme, ad un nuova proprietà denominata *situatedness*.

Considerando l'ambito dei *Sistemi Multi-Agente (MAS)*, questi vengono definiti come un insieme di agenti (entità autonome) *situati* in un certo ambiente ed interagenti tra loro mediante un'opportuna organizzazione. La proprietà di *situatedness* si riferisce dunque alla capacità, che gli agenti devono possedere, di

essere strettamente correlati con l'ambiente che li circonda. Tale capacità viene tipicamente espressa in termini di abilità nel reagire ai cambiamenti nell'ambiente, i quali possono riferirsi a tempo, spazio o variazioni dell'ambiente stesso. Risulta chiaro quindi che questa proprietà rappresenta tutt'oggi il problema più critico da risolvere per permettere la corretta interazione e coordinazione tra agenti e ambiente all'interno dei Sistemi Multi-Agente.

In questo elaborato viene presa in considerazione l'infrastruttura di coordinazione TuCSoN (**Tu**ple **C**entres **S**pread **o**ver the **N**etwork), basata su spazi condivisi chiamati *centri di tuple*, tramite i quali è possibile progettare sistemi complessi caratterizzati da un numero dinamico di agenti collegati, distribuiti ed aventi in comune un unico spazio condiviso sul quale coordinarsi o semplicemente scambiare informazioni espresse sotto forma di *tuple*.

L'infrastruttura di coordinazione TuCSoN si basa sul linguaggio di specifica ReSpecT (**Re**action **Spec**ification **T**uples), tramite il quale è possibile specificare un determinato comportamento che il centro di tuple deve avere in risposta agli eventi e quindi anche alle variazioni nell'ambiente.

Con riguardo alla *situatedness*, l'infrastruttura TuCSoN e il linguaggio di specifica ReSpecT sono già stati estesi con i concetti di tempo e ambiente ma, allo scopo di poter definire questa tecnologia come *situata*, risulta necessario estenderla ulteriormente, iniettando in essa il concetto di spazio, cosicché sia possibile progettare sistemi complessi nei quali, agenti e centri di tuple che vi appartengono, siano in grado di dialogare a proposito di spazio e coordinarsi in relazione alla loro posizione.

L'elaborato è organizzato come segue. Nel primo capitolo vengono presentati l'infrastruttura TuCSoN, il linguaggio di specifica ReSpecT e il modello di programmazione sul quale essi si basano, prima ne vengono introdotti i concetti di base e successivamente viene definita la loro architettura generale, descrivendola anche in termini formali.

Nel secondo capitolo vengono analizzate le estensioni riguardanti la gestione del tempo e l'interazione con l'ambiente, descrivendone le principali caratteristiche ed infine fornendo una descrizione dell'estensione formale applicata all'architettura di base.

Nel terzo capitolo viene analizzato il concetto di Spatial Computing per comprendere alcuni aspetti chiave dei sistemi pervasivi che si stanno prendendo in considerazione. Successivamente viene introdotta la nozione di topologia, concentrandosi su quella fisica e descrivendo nel dettaglio il modello *st*ReSpecT come estensione da realizzare a quello di base per il supporto alla coordinazione basata sullo spazio.

Nel quarto capitolo viene definita una versione *space-aware* di TuCSoN e ReSpecT, estendendo quella originale con le astrazioni e i meccanismi per la gestione degli eventi spaziali. Prima di tutto vengono descritte le dinamiche di comunicazione e coordinazione già esistenti, dopodiché vengono descritti nel dettaglio gli aspetti implementativi che hanno permesso di poter definire l'infrastruttura TuCSoN come completamente *situata*.

Nel quinto capitolo viene presentato un nuovo livello logico con il quale è stato estesa l'infrastruttra TuCSoN allo scopo di fornire allo sviluppatore una piattaforma di geolocalizzazione generica per la creazione e il controllo dei servizi di localizzazione, nonché il loro interfacciamento con l'architettura già esistente.

Nel sesto capitolo viene presentata l'applicazione *Android* realizzata al fine di verificare concretamente il funzionamento dell'estensione implementata in questo elaborato e del nuovo livello logico fornito per la geolocalizzazione.

Infine, nel settimo capitolo viene mostrato un interessante caso di studio che mette in evidenza i risultati ottenuti grazie alle estensioni dell'infrastruttura TuCSoN e del linguaggio di specifica ReSpecT, progettate, implementate e descritte all'interno di questo elaborato.

Indice

Capitolo 1

Il Meta-Modello A&A

Nel contesto dei Sistemi Multi-Agente (MAS), la coordinazione ha sempre rappresentato uno dei problemi chiave per lo studio di nuovi linguaggi e modelli computazionali che incapsulassero le astrazioni necessarie per la progettazione di tale sistema.

Questo aspetto è diventato ancora più rilevante con la nascita e diffusione di sistemi complessi, distribuiti ed eterogenei, nei quali la coordinazione viene generalmente vista come un modo di integrare svariate attività o processi in maniera tale che il sistema nel complesso sia in grado di mostrare le caratteristiche desiderate e/o richieste dal progettista.

In questo capitolo si discuterà del *Meta-Modello A&A* come modello per la coordinazione nei MAS in quanto appositamente progettato ed esteso allo scopo di gestire le problematiche legate a questo aspetto.

1.1 Il meta-modello

Il meta-modello A&A si focalizza sul concetto di MAS come sistema computazionale composto da due astrazioni fondamentali: *agenti* e *artefatti*. Gli agenti, visti come le entità attive del sistema, incapsulano il controllo e hanno il compito di portare a termine obiettivi (o compiti) definendo il comportamento dell'intero sistema. Gli artefatti invece rappresentano entità passive e reattive, ed hanno il compito di fornire servizi e funzioni utili agli agenti necessarie per lavorare in sinergia e portare a termine i compiti a loro assegnati. Una caratteristica fondamentale del meta-modello A&A consiste nella possibilità di reinterpretare la natura e l'ambiente, infatti, considerando la società umana, gli agenti possono essere identificati nelle persone comuni mentre gli artefatti negli oggetti e utensili utilizzati da esse per migliorare le proprie abilità e interagire con l'ambiente sulla base di specifici criteri.

1.2 Agenti e Artefatti

Come introdotto nella sezione precedente, all'interno del meta-modello si possono distinguere due astrazioni, agenti e artefatti, delle quali si discuterà più approfonditamente in questa sezione.

1.2.1 Agenti

Gli agenti rappresentano una delle astrazioni fondamentali del meta-modello A&A e sono componenti di natura *pro-attiva*, progettati per svolgere una o più attività con lo scopo di raggiungere un determinato obiettivo. A tal proposito, necessitano di specifiche abilità e capacità di ragionamento, e sono in grado di sfruttare gli oggetti resi a loro disponibili (artefatti) per poter svolgere al meglio i compiti a loro assegnati.

All'interno del sistema, gli agenti sono *entità autonome* che incapsulano il loro flusso di controllo, sono completamente definiti e non possono essere invocati quindi non devono fornire ne interfacce ne metodi, in quanto per loro natura sono essi stessi che dovranno invocare ed utilizzare gli strumenti a loro necessari e incapsulare un criterio per *autogovernarsi*.

Inoltre, gli agenti sono *dinamici*, possono cioè decidere per la loro esecuzione (non più passivi, ne semplicemente reattivi, ma attivi), e *impredicibili* poiché si possono specificare criteri talmente articolati da non poterne prevedere l'esito dell'esecuzione oppure perché vengono specificati, ad esempio, algoritmi di apprendimento che ne cambiano la percezione dell'ambiente che li circonda.

Un'altra caratteristica importante risiede nel fatto che, essendo pensati come entità orientate al raggiungimento di un obiettivo e il cui operato determina il comportamento dell'intero sistema, gli agenti sono caratterizzati anche da *capacità sociali* ma essendo entità ingovernabili la comunicazione tra due agenti non può modificarne il comportamento in maniera assoluta. Infatti, essendo la comunicazione basata su scambio di dati, tali informazioni possono solo avere l'effetto di condurre l'agente a ragionare e di conseguenza *deliberare* in base agli strumenti che ha a disposizione.

Questo dialogo è una delle interazioni fondamentali all'interno dei sistemi multi-agente ed in particolare all'interno del meta-modello A&A, nel quale si considera il sistema multi-agente nel suo complesso piuttosto che i singoli agenti, quindi per garantire l'interazione agente-agente e agente-risorsa, dovranno essere resi disponibili artefatti modellati opportunamente.

1.2.2 Artefatti

Gli artefatti rappresentano la seconda astrazione fondamentale del meta-modello A&A. Essi sono componenti *passivi* e *reattivi* costruiti ed utilizzati dagli agenti durante le loro attività. In particolare, possono essere *individuali*, quando vengono utilizzati da un singolo agente, oppure *sociali*, quando vengono utilizzati in cooperazione da più agenti e quindi essere sfruttati come entità di coordinazione delle interazioni.

In accordo con la *Activity Theory (AT)* [5], ogni attività viene svolta da una o più componenti del sistema e non può essere concepita senza considerare gli strumenti o artefatti che mediano le azioni e interazioni tra i componenti. Seguendo questo principio, gli artefatti mediano, da un lato le interazioni tra i componenti individuali e l'ambiente in cui vivono, dall'altro incapsulano la porzione dell'ambiente progettata per supportare le attività dei componenti del sistema. Inoltre, come parte osservabile del sistema, gli artefatti possono essere monitorati a loro volta allo scopo, ad esempio, di valutare le prestazioni complessive del sistema stesso [11]. Si può dunque identificare anche una terza tipologia di artefatti, ovvero gli *artefatti ambientali* come mediatori tra l'am-

biente e gli agenti del sistema.

A differenza degli agenti gli artefatti non sono autonomi e non hanno capacità sociali ma presentano diverse proprietà utili agli agenti che lo utilizzano, in particolare:

inspectability – gli agenti devono poter sfruttare al meglio l'artefatto quindi si deve rendere il suo comportamento (lo stato, il contenuto, le funzioni), in parte o interamente, visibile e osservabile.

controllability – un artefatto può garantire il controllo sulla propria struttura, stato e comportamento. Unitamente alla proprietà precedente si garantisce il completo controllo e monitoraggio dell'artefatto.

malleability/forgeability – capacità di cambiare e/o adattare a runtime le funzionalità e il comportamento di un artefatto.

predictability – definite le funzionalità e il comportamento che un artefatto deve avere, esso diviene prevedibile dal punto di vista degli agenti che lo utilizzano.

formalisability – i sistemi complessi e/o con comportamenti emergenti non sono di base formalizzabili. In questi casi nasce il bisogno di avere, all'interno del sistema, componenti il cui comportamento risulti predicibile e quindi completamente formalizzabile.

linkability – capacità di collegare tra loro, a runtime, artefatti distinti, dando la possibilità di ottenere una composizione dinamica utile per poter scalare il sistema in accordo con la complessità della funzione fornita ed anche per il supporto al riuso dinamico.

distribution – capacità di un artefatto di essere distribuito. Questa proprietà unita alla precedente permette di avere un unico artefatto distribuito su più nodi collegati tra di loro.

situation – proprietà di essere immerso in un ambiente multi-agente e di essere reattivo agli eventi e ai cambiamenti dell'ambiente stesso.

Un artefatto dunque, non è altro che uno strumento senza alcuna attività che viene progettato solo allo scopo di fornire funzioni aggiuntive agli agenti che lo utilizzano. Per permettere ciò dovrà, quindi, essere caratterizzato da un'*interfaccia di utilizzo* che renda disponibili tutte le funzioni utilizzabili dagli agenti per interagire con l'ambiente o gli altri agenti.

Relativamente ai sistemi multi-agente, a questo punto si può dichiarare che l'astrazione primaria per la coordinazione all'interno di un sistema complesso è rappresentata dagli artefatti, più precisamente dagli *artefatti di coordinazione*. Tali artefatti vengono utilizzati nel contesto delle attività collaborative, mediando l'interazione tra gli attori coinvolti nello stesso contesto sociale, quindi sono artefatti sociali condivisi dagli agenti del sistema, pensati per abilitare e governare l'interazione agente-agente e agente-ambiente.

Il meta-modello A&A non fa altro che riformulare il concetto di spazio di interazione all'interno di un MAS in modo tale che i componenti di un MAS

possano interagire in tre modi differenti: gli agenti *parlano* con altri agenti, gli agenti *usano* gli artefatti e gli artefatti sono *collegati* ad altri artefatti [11].

1.3 TuCSoN e ReSpecT

Allo scopo di affrontare i problemi di interazione e coordinazione, una delle possibili alternative, e anche quella che si adatta meglio al modello concettuale descritto nelle sezioni precedenti, è rappresentata dall'utilizzo del modello di coordinazione TuCSoN e del linguaggio di coordinazione ReSpecT su cui esso si basa. In questa sezione verranno introdotti questi due elementi poiché insieme formano il framework concettuale di base per gli aspetti che verranno affrontati nei prossimi capitoli.

1.3.1 TuCSoN

TuCSoN (**Tu**ple **C**entres **S**pread **o**ver the **N**etwork) [14], è un modello e infrastruttura general purpose agent-oriented per la coordinazione nei Sistemi Multi-Agente (MAS). TuCSoN è basato su un modello di coordinazione che fornisce gli *spazi di tuple* come astrazione principale per progettare e sviluppare artefatti di coordinazione general purpose. A differenza dello spazio di tuple standard, TuCSoN adotta il linguaggio di specifica ReSpecT che permette la programmazione del comportamento che deve avere il centro di tuple.

Gli agenti interagiscono con il centro di tuple e si coordinano tramite lo scambio di *tuple*, attraverso un insieme di primitive LINDA-like (`in`, `rd`, `inp`, `rdp`). Questo approccio presenta tre caratteristiche principali:

comunicazione generativa – le informazioni inserite nel centro di tuple sono disaccoppiate dal ciclo di vita di chi le ha generate. Questo permette di ottenere il disaccoppiamento spaziale e temporale degli agenti.

accesso associativo – l'accesso alle informazioni è basato sul meccanismo di *tuple matching*, considerando la loro struttura e contenuto piuttosto che la loro posizione o il loro nome.

semantica sospensiva – la coordinazione è basata sulla disponibilità delle informazioni nel centro di tuple.

Modello

Un sistema basato su TuCSoN è una collezione di agenti che interagiscono tramite il centro di tuple ReSpecT (*media di coordinazione*), posizionato in un insieme di nodi che possono essere distribuiti nella rete. Il centro di tuple è composto da due elementi: uno spazio condiviso per la comunicazione basata su tuple (*tuple space*) e uno *spazio di specifica* che contiene la logica di comportamento del centro di tuple stesso.

Ogni centro di tuple viene identificato con l'ausilio di un *nome*:

```
tcname@netid:portno
```

Dove:

- `tcname`: rappresenta il nome del centro di tuple definito come un termine Prolog ground di prim'ordine.
- `netid`: rappresenta l'indirizzo IP del dispositivo che ospita il centro di tuple.
- `portno`: rappresenta il numero di porta sulla quale il servizio di coordinazione è in ascolto.

L'interazione tra gli agenti e il centro di tuple avviene tramite un *linguaggio di coordinazione* che permette di eseguire *operazioni di coordinazione*, ciascuna delle quali viene eseguita in due fasi. Innanzitutto un agente richiede l'esecuzione di un'operazione su uno specifico centro di tuple (*invocation*), successivamente, dopo che il centro di tuple ha elaborato il risultato, esso risponde con le informazioni richieste (*completion*).

La sintassi per l'invocazione di un'operazione (`op`) su un centro di tuple è `tcname@netid:portno?op`, tramite la quale è possibile invocare una primitiva anche su un centro di tuple remoto e non solo su quello in esecuzione sul nodo locale. In particolare, TuCSoN mette a disposizione degli agenti nove primitive di base [9], che sono:

- `out(Tuple)`: una nuova tupla `Tuple` viene inserita nel centro di tuple. Dopo che l'operazione è stata eseguita con successo, la tupla viene restituita.
- `rd(TupleTemplate)`: cerca una tupla `Tuple`, corrispondente al template `TupleTemplate`, nel centro di tuple. Se viene trovata almeno una tupla corrispondente al template l'esecuzione dell'operazione termina con successo e la tupla viene restituita. Altrimenti, l'esecuzione viene sospesa fino a che non è presente una tupla che corrisponda con il template specificato.
- `in(TupleTemplate)`: come la precedente ma, nel momento in cui la tupla viene trovata e restituita, essa viene anche rimossa dal centro di tuple.
- `rdp(TupleTemplate)`: versione con semantica non sospensiva dell'operazione `rd(TupleTemplate)`. Se non viene trovata alcuna tupla corrispondente al template specificato, l'esecuzione fallisce e viene restituito il template stesso.
- `inp(TupleTemplate)`: versione con semantica non sospensiva dell'operazione `in(TupleTemplate)`. Se non viene trovata alcuna tupla corrispondente al template specificato, l'esecuzione fallisce e viene restituito il template stesso.
- `no(TupleTemplate)`: cerca una tupla `Tuple` che corrisponda al template `TupleTemplate` specificato. Se non viene trovata alcuna tupla, l'esecuzione termina con successo e viene restituito il template stesso. Altrimenti, l'esecuzione viene sospesa fino a che la tupla corrispondente al template non viene rimossa dal centro di tuple, in tal caso viene restituito il template stesso.
- `nop(TupleTemplate)`: versione con semantica non sospensiva dell'operazione `no(TupleTemplate)`. Se viene trovata almeno una tupla corrispondente al template specificato, l'esecuzione fallisce e viene restituita la tupla stessa.

- `get`: legge tutte le tuple `Tuple` presenti nel centro di tuple e restituisce una lista che le contiene. Se non viene trovata alcuna tupla, viene restituita una lista vuota e comunque l'esecuzione termina con successo.
- `set(Tuples)`: sovrascrive tutte le tuple presenti nel centro di tuple con quelle specificate nella lista in ingresso. Quando l'esecuzione termina, viene restituita la lista `Tuples`.

La necessità di poter gestire più tuple con una singola operazione, ha portato alla definizione di nuove primitive chiamate *primitive bulk*: `out_all`, `rd_all`, `in_all`, `no_all`. Tali primitive restituiscono una lista contenente tutte le tuple che corrispondono al template specificato, altrimenti, se non ne viene trovata nessuna, una lista vuota.

Inoltre, TuCSoN mette a disposizione anche le *primitive uniformi*: `urd`, `urdp`, `uin`, `uinp`, `uno`, `unop`. Tali primitive sono state introdotte per poter definire comportamenti probabilistici all'interno del meccanismo di coordinazione degli agenti, infatti sostituiscono il determinismo tipico delle primitive LINDA-like con una distribuzione di probabilità uniforme.

Infine, è presente anche una primitiva `spawn(Op,TCId)`, la quale permette di attivare computazioni, Java o Prolog, localmente al centro di tuple nel quale viene invocata. Questa operazione è caratterizzata da una semantica non sospensiva e avvia un'attività (`Op`) di computazione parallela che viene eseguita in maniera asincrona all'interno del centro di tuple specificato da `TCId`.

Architettura

Un sistema basato su TuCSoN è costituito da un'insieme di *nodi*, interconnessi tramite la rete, che nel complesso formano lo *spazio di coordinazione*. Ogni nodo ospita i servizi TuCSoN ed è caratterizzato da un'interfaccia di rete (`netid`) e da una porta (`portno`) sulla quale il servizio resta in ascolto delle richieste in arrivo. Ogni nodo può quindi contenere un qualsiasi numero di centri di tuple, ai quali viene associato un *nome logico* che li identifica univocamente.

Un agente può comunicare con qualsiasi centro di tuple tramite la specifica del suo nome logico, il quale può essere *assoluto*, per i centri di tuple che risiedono in un nodo differente da quello dell'agente, oppure *relativo*, per i centri di tuple che risiedono invece nello stesso nodo che ospita l'agente. Si possono dunque identificare due spazi di coordinazione, uno *globale*, composto da tutti i centri di tuple presenti in tutti i nodi del sistema, e uno *locale*, composto dai centri di tuple presenti nel nodo che ospita l'agente. Ogni nodo inoltre definisce un centro di tuple (`default`) e una porta (20504) di default, quindi se non viene specificato alcun indirizzo IP, l'esecuzione delle operazioni avviene nello spazio di coordinazione locale.

Dal punto di vista degli agenti, l'idea di TuCSoN consiste nello strutturare i centri di tuple in modo tale da governare l'accesso associando ad ogni agente un *ruolo* specifico. Questo aspetto viene realizzato con l'ausilio del modello *Role-Based Access Control (RBAC)* che richiede la presenza di un centro di tuple adibito a contenere le regole di accesso. A questo scopo, è stata introdotta la nozione di *Agent Coordination Context (ACC)*, che consiste in un'interfaccia utilizzata dagli agenti per l'invocazione delle primitive sul centro di tuple. Questo componente regola l'interazione tra gli agenti e il

centro di tuple e lo fa tramite tre ACC di base, definiti sia in versione sincrona che asincrona, i quali fungono da interfaccia per, rispettivamente, tre insiemi di primitive: *Ordinary(Synch/Asynch)ACC*, *Bulk(Synch/Asynch)ACC*, *Uniform(Synch/Asynch)ACC*.

Utilizzando la versione sincrona di un ACC, un agente che invoca una primitiva, successivamente sospende la sua esecuzione in attesa del suo completamento. Contrariamente, utilizzando la versione asincrona, l'agente non sospende la sua esecuzione ma viene notificato in maniera asincrona del completamento dell'operazione.

Inoltre, TuCSoN fornisce un'altro ACC, *Specification(Synch/Asynch)ACC*, utilizzato allo scopo di permettere l'accesso alla specifica ReSpecT.

1.3.2 ReSpecT

Come già detto, i centri di tuple in TuCSoN vengono programmati attraverso il linguaggio di specifica ReSpecT (**Re**action **Spec**ification **T**uples).

Essendo nato come linguaggio di specifica dei comportamenti, ReSpecT permette la definizione di attività computazionali, dette *reazioni*, all'interno dei centri di tuple e associa queste ultime ad eventi che si verificano all'interno del centro di tuple stesso. Conseguentemente, si può affermare che ReSpecT risulta composto da due componenti, una *dichiarativa* in quanto è possibile dichiarare le reazioni associate a certi eventi tramite apposite *tuple di specifica*, ed una *procedurale* poiché permette di definire reazioni sotto forma di sequenze di obiettivi da raggiungere (*reaction goals*), ognuno dei quali può avere esito positivo (`success`) o fallire (`failure`).

In ReSpecT si possono dunque identificare due tipi di tuple [11],[14]:

tuple ordinarie – è un termine `Prolog` di prim'ordine, che può anche essere *ground*.

tuple di specifica – è un termine `Prolog`, nella forma `reaction(E, R)`. Dato un evento *Ev*, tale tupla di specifica associa un evento di comunicazione rappresentato dal termine `E` ad una reazione $R\theta$ se $\theta = mgu($`E`$, Ev)$[1]. *R* rappresenta i `ReactionGoals`, ovvero l'insieme di operazioni ReSpecT che devono essere eseguite in risposta agli eventi.

Le tuple di specifica quindi, definiscono il comportamento di uno spazio di tuple in termini di reazioni di interazione [13], specificando quindi come il centro di tuple debba reagire agli eventi di comunicazione che gli pervengono e modificandone il comportamento a tempo di esecuzione.

Ogni reazione viene eseguita *sequenzialmente* con una *semantica transazionale* e tutte le reazioni innescate da un evento di comunicazione vengono eseguite prima che ne venga servito un altro, quindi gli agenti percepiscono l'avvenuta elaborazione di un evento (e di tutte le reazioni associate) come una singola transizione. Questo fa si che gli effetti dell'invocazione di una primitiva possano essere complessi quanto serve. In particolare, quest'ultima caratteristica permette di realizzare qualsiasi tipo di computazione logica, rendendo quindi ReSpecT un linguaggio *general-purpose*.

[1] *mgu* è il most general unifier come definito nella programmazione logica

1.4 A&A ReSpecT

Nella prospettiva del meta-modello A&A si può affermare che, da un lato TuCSoN mette a disposizione svariati centri di tuple (artefatti distribuiti), contenenti sia informazioni che logica di coordinazione espressa sotto forma di tuple logiche. Dall'altro lato, il centro di tuple ReSpecT risulta essere un artefatto avente le proprietà di inspectability e malleability, inoltre, con successive estensioni, anche le proprietà di linkability e situatedness risultano essere parte integrante del linguaggio di coordinazione.

Insieme, TuCSoN e ReSpecT, formano quindi un framework concettuale che soddisfa al meglio le necessità del meta-modello A&A ReSpecT.

1.4.1 Proprietà

Come già detto in precedenza, gli artefatti sono entità passive mentre gli agenti pro-attive. Per questo motivo, all'interno di un MAS basato sul meta-modello A&A, saranno l'agente e l'ambiente le uniche sorgenti degli eventi che si verificano nel sistema. Nonostante questo, gli artefatti restano comunque entità reattive e collegate tra di loro, quindi, in linea di principio, possono influenzarsi a vicenda. Di conseguenza, l'invocazione da parte di un altro artefatto potrebbe essere la causa diretta di un evento.

Quanto detto conduce alla naturale necessità di avere una *nozione di evento generalizzata* nella quale viene specificata sia la causa primaria dell'evento che quella diretta, allo scopo di consentire la piena osservabilità di tutti gli eventi che avvengono all'interno del sistema.

Il meta-modello A&A promuove anche il *disaccoppiamento del controllo*. Questa proprietà richiede un adattamento dei linguaggi e modelli per garantire il completo disaccoppiamento di artefatti composti e lasciare all'agente la completa autonomia nella scelta del tipo di comunicazione, ovvero se eseguire operazioni in maniera sincrona o asincrona, e allo stesso tempo mantenere inalterato il comportamento degli artefatti.

Questo tipo di disaccoppiamento viene garantito elaborando ciascun evento in due fasi, in particolare si può affermare che l'esecuzione di ogni operazione presenta una struttura di tipo *request/response* nella quale si distinguono una fase di *invocation* (request), attuata nel momento in cui la richiesta per l'esecuzione di un'operazione viene ricevuta, ed una conseguente fase di *completion* (response), attuata nel momento in cui tale richiesta viene servita e nella quale, se necessario, vengono forniti anche i risultati.

Nel caso di artefatti composti (*linked*) risulta necessario che queste due fasi vengano gestite in maniera del tutto asincrona, allo scopo di garantire il completo disaccoppiamento del controllo dei singoli artefatti.

1.4.2 Estensioni di ReSpecT

In relazione alle proprietà descritte sopra, relative alla definizione di un descrittore generalizzato degli eventi ed al supporto al disaccoppiamento del controllo, il linguaggio di coordinazione ReSpecT è stato esteso in [11] definendo una nuova sintassi per il meta-modello A&A.

In questa rivisitazione del linguaggio è stata definita un'estensione per la specifica delle *guardie*, quindi, a quella che era la struttura delle reazioni, viene aggiunto un nuovo termine `G` rappresentate la guardia. Le tuple di specifica in A&A ReSpecT assumono quindi la forma `reaction(E,G,R)`. Dato un evento *Ev*, tale tupla di specifica associa un evento di comunicazione rappresentato dal termine `E` ad una reazione `R`θ se $\theta = mgu($`E`, *Ev*$)$[2] e se le condizioni espresse dalla guardia `G` sono soddisfatte.

Più nello specifico:

- `E` è l'evento al quale la reazione è associata.
- `G` è la guardia associata all'evento, ovvero l'insieme di condizioni che devono essere soddisfatte affinché tutte la reazioni presenti in `R` possano essere eseguite. Una guardia è una sequenza di predicati (di guardia) definiti nella sintassi del linguaggio dal predicato ⟨*GuardPredicate*⟩ [11].
- `R` è il corpo della reazione espressa sotto forma di primitive ReSpecT. In particolare, come già detto nella Sottosezione 1.3.2, rappresenta l'insieme dei `ReactionGoals`, contenente tutte le operazioni da eseguire (atomicamente) in risposta agli eventi.

Grazie alla definizione dei predicati di guardia, è possibile controllare un ampio numero di condizioni allo scopo di decidere se una reazione associata ad un certo evento debba essere innescata all'interno del centro di tuple.

Seguendo la stessa linea di principio, i predicati di osservazione sono stati generalizzati in accordo con il nuovo modello. In particolare, i predicati ⟨*ObservationPredicate*⟩ [11] assumono la forma ⟨*EventView*⟩`_`⟨*EventInformation*⟩ , dove ⟨*EventView*⟩ può essere:

- `event_`: termine che si riferisce alla causa diretta di un evento.
- `start_`: termine che si riferisce alla causa primaria di un evento.

Un'altra importante estensione al modello consiste nell'introduzione del predicato ⟨*TCStatePredicate*⟩ [11] grazie al quale un agente ha la capacità di invocare le primitive di comunicazione. Tale predicato può essere utilizzato all'interno delle reazioni per agire sullo stato di un centro di tuple (locale o remoto).

Nella versione originale di ReSpecT era già stata definita una prima estensione *time-aware* con l'introduzione, in [12], di *reazioni temporizzate*, *eventi temporali* e vari predicati per la gestione del tempo.

Infine, relativamente alla *situatedness*, in [2] il modello viene esteso con l'introduzione del predicato ⟨*EnvPredicate*⟩, grazie al quale è possibile definire reazioni che rispondono ad eventi generati dall'ambiente esterno. Inoltre, l'architettura di ReSpecT è già stata estesa con nuovi predicati e guardie per poter includere le risorse ambientali tra le sorgenti e/o destinazioni degli eventi.

Entrambe queste estensioni verranno analizzate più approfonditamente nel prossimo capitolo.

[2] *mgu* è il most general unifier come definito nella programmazione logica

L'unico aspetto non ancora trattato approfonditamente e discusso formalmente in [8], riguarda la coordinazione *space-aware* per la gestione della topologia dei MAS e della mobilità dei centri di tuple. Questo elaborato si propone proprio di affrontare tale aspetto e l'estensione al linguaggio verrà trattata nel dettaglio in seguito.

1.4.3 Architettura

L'architettura di un centro di tuple ReSpecT ha come elemento fondamentale il *ReSpecT Engine*, che consiste in una macchina a stati avente il compito di elaborare le richieste e generare le risposte. Collegato a questo componente si hanno il *ReSpecT Specification*, che contiene le specifiche di comportamento del centro di tuple, il *ReSpecT Interpreter* che si occupa di generare gli eventi di uscita, e il *Tuple Set* ovvero lo spazio di tuple vero e proprio, contenente le tuple informative e di coordinazione.

1.4.4 RespectVM

Essendo un *reaction specification language*, ReSpecT fornisce il supporto per l'esecuzione di reazioni, innescate da eventi che si verificano all'interno del centro di tuple, seguendo uno specifico *comportamento*.

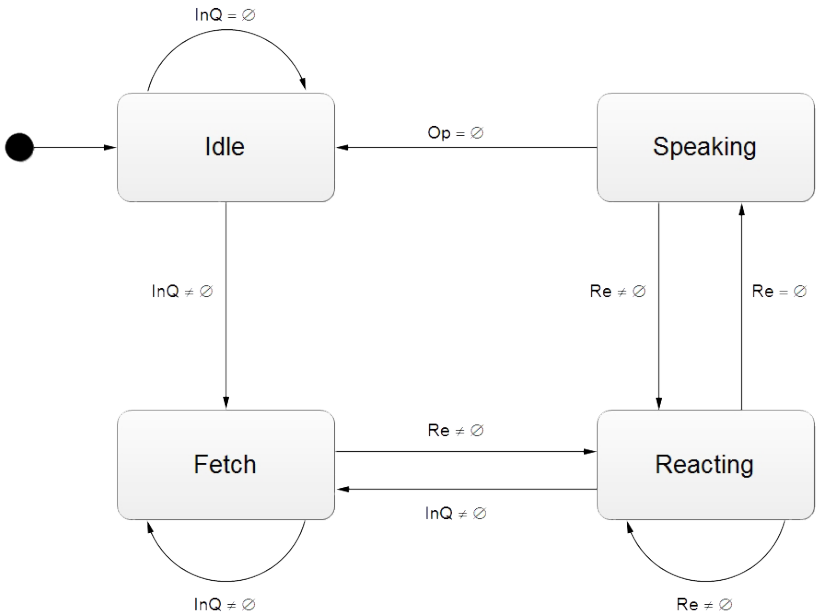


Figura 1.1: ReSpecT - Ciclo di lavoro della macchina virtuale

In termini non formali, alla ricezione di una richiesta, il *ReSpecT Engine* reagisce seguendo un ciclo di lavoro a stati. Ogni qualvolta avviene l'invocazione di una primitiva da parte di un agente (*operation*) o di un artefatto (*link*), viene generato un evento A&A ReSpecT *ammissibile*, ovvero che rispetta la sintassi definita in [11]. L'evento generato viene automaticamente elaborato ed inserito nella coda *InQ* di eventi in input al centro di tuple.

Nel caso in cui non vi siano reazioni in esecuzione o da eseguire, quindi se la macchina virtuale si trova nello *stato di idle*, viene estratto il primo evento ϵ della coda *InQ*, secondo la logica FIFO, e viene inserito in un multiset *Op* rappresentate le operazioni da eseguire (*fase di request*): questo causa la transizione allo *stato di fetch*.

Tutte le reazioni innescate dall'evento ϵ vengono inserite nella lista *Re* delle reazioni in attesa di essere elaborate, le quali, con il passaggio allo *stato di reacting*, vengono eseguite sequenzialmente (in maniera non-deterministica) e con semantica transazionale. Ogni reazione può generarne a sua volta altre che vengono inserite allo stesso modo nella lista *Re*, oppure generare eventi di output che vengono inseriti nel multiset *Out* degli eventi di uscita e poi spostati nella coda *OutQ* al termine dell'esecuzione della reazione associata.

Questo processo continua fino a che la lista *Re* non risulta vuota. A questo punto, la macchina virtuale transita allo *stato di speaking* in cui le richieste in attesa di essere servite, presenti in *Op*, vengono eseguite dal centro di tuple e i risultati della operation (o del link) vengono restituiti a chi ha effettuato la richiesta. Questo può portare all'innescamento di altre reazioni associate alla *fase di response*, le quali vengono eseguite con la stessa modalità appena descritta per la fase di request.

Al termine di quest'ultima fase il ciclo di lavoro risulta concluso e il *ReSpecT Engine* torna allo stato di idle.

Implementazione della RespectVM

Analizzando come ReSpecT gestisce l'elaborazione degli eventi e delle code a livello software, si può notare la presenza della classe `RespectVM` che rappresenta l'entità principale dell'intero ciclo di elaborazione. Questa classe è un processo incaricato di rimanere in ascolto degli eventi generati in risposta alle richieste effettuate dagli agenti, ed il cui ruolo principale consiste nell'assicurare la continuità del ciclo di lavoro del ReSpecT Engine, inserendo gli eventi nella coda di ingresso *InQ*.

La macchina a stati descritta formalmente sopra, viene definita a livello software da sei stati, che sono:

- `ResetState`: rappresenta lo stato iniziale nel quale la macchina virtuale si troverà solo all'avvio del sistema.
- `IdleState`: rappresenta la condizione di assenza di eventi da gestire o di reazioni da elaborare.
- `ListeningState`: stato raggiunto quando il sistema è in attesa di nuovi eventi generati dagli agenti esterni.
- `ReactingState`: stato raggiunto quando sono presenti reazioni innescate a seguito del verificarsi degli eventi.

- `FetchEnvState`: stato raggiunto quando il sistema percepisce gli eventi ambientali.
- `SpeakingState`: stato fondamentale incaricato di gestire l'esecuzione delle primitive invocate dagli agenti.

Ad ogni stato, inoltre, corrisponde una classe omonima utilizzata per l'implementazione del suo comportamento. Di seguito viene riproposto il diagramma mostrato in Figura 1.1, specificando gli stati e le transizioni appena descritti.

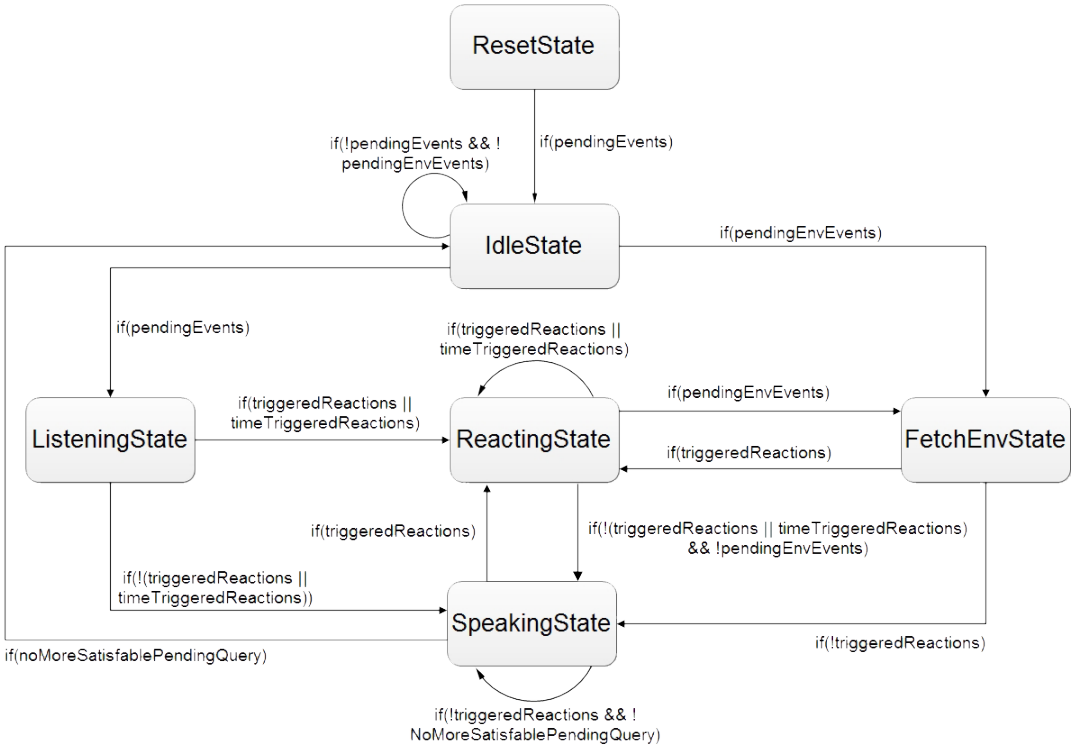


Figura 1.2: ReSpecT - Ciclo di lavoro completo della macchina virtuale

Lo stato `FetchEnvState`, mostrato in Figura 1.2 è stato inserito con l'estensione di ReSpecT per a situatedness, trattata più approfonditamente nel Capitolo 2. Questo stato si occupa della gestione degli eventi ambientali in attesa nella coda *EnvQ*. Questi ultimi vengono considerati meno prioritari rispetto a quelli ordinari, in particolare, solo nel caso in cui la coda *InQ* risulti vuota verranno considerati eventuali eventi presenti nella coda *EnvQ*. Ogni evento ambientale viene trattato esattamente come un qualsiasi evento ordinario, infatti esso può innescare reazioni all'interno del centro di tuple elaborate successivamente nel `ReactingState`, dal quale è possibile tornare nuovamente nello stato `FetchEnvState` nel caso in cui vi fossero ulteriori eventi ambientali da gestire.

Un'altra classe principale è rappresentata da `RespectVMContext` che si occupa di fornire un'interfaccia tra TuCSoN e le operazioni ReSpecT. Questa classe è incaricata di adempiere a due compiti; innanzitutto deve gestire inserimento, rimozione e lettura delle tuple nel centro di tuple, in secondo luogo deve fornire le funzioni utili per verificare se la primitiva invocata porta alla generazione di un evento che, a sua volta, implica l'innescarsi di reazioni.

La reale esecuzione del corpo di una reazione, viene delegata in parte al motore Prolog, per quanto riguarda le computazioni Prolog, e in parte alla classe `Respect2PLibrary`, che rappresenta una libreria tuProlog attraverso la quale è possibile definire il comportamento delle primitive ReSpecT. Questa libreria viene utilizzata lato core, all'interno della `RespectVM`.

Analogamente, la classe `Tucson2PLibrary` rappresenta una libreria tuProlog che permette agli agenti di interagire con il sistema TuCSoN. Attraverso questa libreria, utilizzata lato agente, gli agenti Java, tuProlog o umani sono in grado di accedere a tutte le primitive definite all'interno del media di coordinazione.

Formalizzazione del comportamento

In termini formali, lo stato di un centro di tuple A&A ReSpecT viene espresso dalla quadrupla seguente:

$$InQ\langle\, Tu, \Sigma, Re, Op\rangle_{n}^{OutQ}$$

Dove:

- Tu ed Σ sono multiset contenenti rispettivamente le tuple ordinarie e le tuple di specifica;
- Re è il multiset delle reazioni innescate dagli eventi e in attesa di essere eseguite.
- Op è il multiset delle richieste in attesa di una risposta.
- InQ e $OutQ$ sono rispettivamente le code degli eventi in ingresso e uscita. Queste due code vengono rispettivamente espanse (con nuovi eventi in ingresso) e svuotate (tramite l'emissione degli eventi in uscita) a tempo di esecuzione.
- n è il tempo locale del centro di tuple.

Il comportamento formale di un centro di tuple, il cui stato è quello appena descritto, può essere quindi modellato come un sistema di transizione composto da quattro transizioni:

reaction – se $Re \neq \emptyset$, le reazioni innescate presenti in Re vengono eseguite tramite una transizione di tipo *reaction* ($\longrightarrow_r$).

time – se $Re = \emptyset$ e $\texttt{timed}(n, \Sigma) \neq \emptyset$, le reazioni temporizzate in attesa di essere elaborate possono innescare altre reazioni tramite una transizione di tipo *time* ($\longrightarrow_t$).

service – se $Re = \texttt{timed}(n, \Sigma) = \emptyset$ e $\texttt{sat}(Op, Tu, \Sigma) \neq \emptyset$[3], le richieste in attesa di una risposta possono essere servite tramite una transizione di tipo *service* ($\longrightarrow_s$).

log – se $Re = \texttt{timed}(n, \Sigma) = \texttt{sat}(Op, Tu, \Sigma) = \emptyset$ e $InQ \neq \emptyset$, le richiese in coda nella InQ possono essere recuperate (*logged*) sequenzialmente da un centro di tuple tramite una transizione di tipo *log* ($\longrightarrow_l$).

[3] $\texttt{sat}(Op, Tu, \Sigma)$ è il sottoinsieme di Op delle richieste in attesa di una risposta che possono essere servite dato lo stato corrente del centro di tuple.

Capitolo 2

Tempo e Ambiente in ReSpecT

Nel corso degli anni il linguaggio di coordinazione ReSpecT è stato esteso in maniera incrementale allo scopo di poter affrontare tutte le problematiche legate alla coordinazione nei sistemi multi-agente e distribuiti. Tali aspetti riguardano principalmente i concetti di tempo, spazio e ambiente, per i quali risulta necessaria un'opportuna gestione all'interno di un sistema che deve essere situato e i cui componenti hanno la necessità di coordinarsi a fronte del passare del tempo e delle eventuali variazioni nella topologia del sistema stesso.

In questo capitolo verranno analizzate in generale le estensioni riguardanti la gestione del tempo e l'interazione con l'ambiente, mentre l'estensione relativa alla gestione dello spazio e quindi alla dinamicità della topologia del sistema verrà discussa più in dettaglio nel prossimo capitolo in quanto oggetto di questo elaborato.

2.1 Coordinazione Time-Aware

L'estensione di ReSpecT per la coordinazione *Time-Aware* è stata trattata in [12].

All'interno del ciclo di lavoro degli artefatti di coordinazione è presente la nozione di *evento temporale*, definito come un evento innescate nuove computazioni per la gestione del tempo. Gli artefatti di coordinazione sono quindi arricchiti con la capacità di catturare gli eventi temporali e di reagire in maniera opportuna: un artefatto in grado di fare questo viene detto *artefatto di coordinazione temporizzato*. Analogamente anche il centro di tuple, che in questo caso viene chiamato *centro di tuple temporizzato*, adotta questo framework temporale, infatti è presente la nozione di *tempo corrente*, definito come un valore discreto monotono crescente espresso in millisecondi di attività del centro di tuple, da quando questo viene creato.

Grazie all'introduzione degli elementi appena descritti, ReSpecT è stato esteso con la nozione di tempo introducendo, adottando la stessa semantica transazionale tipica del modello A&A, alcuni predicati temporali per recuperare le informazioni relative al centro di tuple e agli eventi temporali, e rendendo possibile la specifica di reazioni innescate al verificarsi degli eventi temporali. In

particolare, sono stati definiti un nuovo predicato di osservazione e un nuovo evento. É possibile, infatti, definire eventi `time(`⟨*Time*⟩`)`, innescati nel momento in cui il valore del tempo in millisecondi diviene pari o supera quello indicato da *Time*, e recuperare informazioni temporali sugli eventi tramite il predicato di osservazione ⟨*EventView*⟩`_time`, dove ⟨*EventView*⟩ specifica con quale vista temporale recuperare le informazioni tramite i termini `current`, `event` e `start`. Nello specifico:

- `current_time(`⟨*Time*⟩`)`: ha successo se *Time* (variabile) unifica con il tempo corrente del centro di tuple.
- `event_time(`⟨*Time*⟩`)`: ha successo se *Time* unifica con il tempo del centro di tuple, nel momento in cui si è verificato l'evento innescante la reazione.

Inoltre, sono definite anche alcune guardie per il controllo preliminare dell'esecuzione di una reazione temporale:

- `before(`⟨*Time*⟩`)`: ha successo se il tempo corrente del centro di tuple ha valore minore di quello specificato in *Time*.
- `after(`⟨*Time*⟩`)`: ha successo se il tempo corrente del centro di tuple ha valore maggiore di quello specificato in *Time*.
- `between(`⟨*MinTime*⟩, ⟨*MaxTime*⟩`)`: ha successo se il tempo corrente del centro di tuple ha un valore compreso tra *MinTime* e *MaxTime*.

L'approccio descritto sopra e i predicati introdotti nel linguaggio di coordinazione ReSpecT si rivelano essere sufficientemente generali ed espressivi da permettere la descrizione di un vasto insieme di pattern di coordinazione basati sulla nozione del tempo.

2.2 Interazione con l'Ambiente

L'estensione di ReSpecT per la coordinazione basata sull'interazione con l'ambiente (*situatedness*) è stata trattata in [2].

La *situatedness* è la proprietà del sistema multi-agente di essere strettamente accoppiato con l'ambiente nel quale è situato, ovvero la sua abilità di reagire ai cambiamenti dell'ambiente stesso [2]. Il problema principale deriva dal conflitto tra il comportamento pro-attivo degli agenti e la reattività che viene loro richiesta nei confronti dei cambiamenti che avvengono nell'ambiente che li circonda.

L'infrastruttura ha dunque la responsabilità di tradurre gli eventi ambientali in eventi ReSpecT. A questo scopo, il linguaggio mette a disposizione appropriati componenti, chiamati *trasduttori*, in grado di elaborare eventi generati dall'ambiente e tradurli opportunamente in accordo con il modello ad eventi di ReSpecT. In particolare, ogni trasduttore è specializzato nella gestione di una specifica porzione dell'ambiente che circonda il sistema, la quale viene tipicamente identificata da una risorsa (*sensore* o *attuatore*) collegata al sistema stesso come, ad esempio, un sensore di temperatura.

Oltre ai trasduttori, ReSpecT mette a disposizione vari predicati, guardie e predicati di osservazione per affrontare il tema della situatedness. La variazione

più importante effettuata al modello di base consiste nell'estensione del predicato ⟨*SimpleTCEvent*⟩, al quale viene aggiunto ⟨*EnvPredicate*⟩, grazie al quale è possibile definire nuove reazioni in risposta agli eventi generati dall'interazione con l'ambiente e, più nello specifico, è possibile catturare due tipologie di evento:

- evento scatenato dall'interazione con sensori che percepiscono le proprietà dell'ambiente, gestito dal predicato `get(`⟨*Key*⟩, ⟨*Value*⟩`)`.
- evento scatenato dall'interazione con attuatori che modificano le proprietà dell'ambiente, gestito dal predicato `set(`⟨*Key*⟩, ⟨*Value*⟩`)`.

Poiché vi è la possibilità di inviare e ricevere eventi da e verso risorse ambientali, ReSpecT mette a disposizione l'identificatore ⟨*EnvResIdentifier*⟩ associato all'entità ambientale esterna. Inoltre, è possibile osservare le proprietà di un evento ambientale tramite il predicato di osservazione `env(`⟨*Key*⟩, ⟨*Value*⟩`)`, che permette di osservare proprietà specifiche di un particolare evento ambientale sotto osservazione tramite la specifica del nome (⟨*Key*⟩) e del valore (⟨*Value*⟩) della proprietà stessa.

Per supportare la comunicazione tra centro di tuple e risorse ambientali, ReSpecT mette a disposizione il predicato ⟨*TCEnvPredicate*⟩ definito come composizione tra ⟨*EnvResIdentifier*⟩ e ⟨*EnvPredicate*⟩. Tramite questo predicato è possibile specificare reazioni il cui body può assumere la forma:

- ⟨***EnvResIdentifier***⟩ `?  get(`⟨***Key***⟩, ⟨***Value***⟩`)`: permette di recuperare il valore ⟨*Value*⟩ della proprietà ⟨*Key*⟩ relativa alla risorsa ambientale ⟨*EnvResIdentifier*⟩.
- ⟨***EnvResIdentifier***⟩ `?  set(`⟨***Key***⟩, ⟨***Value***⟩`)`: permette di impostare il valore ⟨*Value*⟩ della proprietà ⟨*Key*⟩ relativa alla risorsa ambientale ⟨*EnvResIdentifier*⟩.

A questi predicati, vengono affiancate due guardie relative alla direzione degli eventi e utili per filtrare le reazioni innescate dagli eventi ambientali:

- `from_env`: ha successo se l'evento proviene da una risorsa ambientale.
- `to_env`: ha successo se l'evento è destinato ad una risorsa ambientale.

Con l'introduzione dei concetti appena descritti, il ciclo di lavoro della macchina virtuale A&A ReSpecT, descritto nella Sottosezione 1.4.4, risulta modificato per garantire il supporto alla proprietà di *situatedness* dei centri di tuple. In questo frangente, la semantica appare estesa con l'introduzione di una *environment queue (EnvQ)*, nella quale inserire tutti gli eventi ambientali generati da sorgenti esterne e/o dal trascorrere del tempo. Conseguentemente, anche tutte le reazioni innescate da eventi ambientali vengono inserite nella lista *Re* delle reazioni in attesa di essere eseguite.

Formalizzazione del comportamento con la Situatedness

In termini formali, lo stato di un centro di tuple A&A ReSpecT, considerando anche la sua estensione per la situatedness, viene espresso dalla quadrupla seguente:

$$InQ,EnvQ\langle Tu, \Sigma, Re, Op\rangle_n^{OutQ}$$

In cui l'unica differenza è data dall'introduzione della *EnvQ* che rappresenta la coda degli eventi ambientali in ingresso.

Analogamente a quanto descritto nella Sottosezione 1.4.4, il comportamento formale di un centro di tuple, il cui stato è quello appena descritto, può essere modellato come un sistema di transizione composto da quattro transizioni:

reaction – se $Re \neq \emptyset$, le reazioni innescate presenti in Re vengono eseguite tramite una transizione di tipo *reaction* ($\longrightarrow_r$).

environment[1] – se $Re = \emptyset$ e $EnvQ \neq \emptyset$, gli eventi ambientali in attesa di essere serviti possono innescare altre reazioni tramite una transizione di tipo *environment* ($\longrightarrow_{env}$).

service – se $Re = EnvQ = \emptyset$ e $\texttt{sat}(Op, Tu, \Sigma) \neq \emptyset$[2], le richieste in attesa di una risposta possono essere servite tramite una transizione di tipo *service* ($\longrightarrow_s$).

log – se $Re = EnvQ = \texttt{sat}(Op, Tu, \Sigma) = \emptyset$ e $InQ \neq \emptyset$, le richiese in coda nella InQ possono essere recuperate (*logged*) sequenzialmente da un centro di tuple tramite una transizione di tipo *log* ($\longrightarrow_l$).

Grazie ai predicati descritti sopra e alla modifica apportata al ciclo di lavoro del meta-modello A&A, è possibile affermare che tale modello è in grado di gestire le problematiche legate alla situatedness. É dunque possibile modellare artefatti situati che permettano agli agenti di essere reattivi ai cambiamenti delle proprietà dell'ambiente e che fungano da ponte tra l'ambiente fisico e software, rendendo possibile ad un artefatto di essere utile laddove la gestione della coordinazione agente-ambiente lo necessiti.

2.3 Ciò che manca: lo Spazio

A valle di quanto discusso in questo capitolo, il linguaggio che ne deriva risulta in grado di gestire eventi legati sia al tempo che all'ambiente. Nonostante questo, il meta-modello A&A presenta ancora grosse limitazioni che risiedono nell'assenza di un supporto esplicito alla topologia del sistema e, conseguentemente, nell'inabilità dello stesso di esplicitare e gestire direttamente gli eventi spaziali.

La proprietà spaziale, così come quella temporale, dovrebbe essere un'entità principale incapsulata nel medium di coordinazione, il quale si ritrova immerso in un ambiente tanto quanto gli agenti che popolano il sistema. Tuttavia, l'interfaccia per l'interazione con le risorse ambientali messa a disposizione dall'estensione per la situatedness descritta nella sottosezione precedente, risulta essere un elemento essenziale per estendere ulteriormente il meta-modello A&A verso il supporto alla *coordinazione space-aware*.

[1] Prima dell'estensione relativa alla situatedness questa transizione era identificata da *timed* e dall'insieme $\texttt{timed}(n,\Sigma)$; ora si ha una transizione *environment* ma di fatto è analoga alla precedente.

[2] $\texttt{sat}(Op, Tu, \Sigma)$ è il sottoinsieme di Op delle richieste in attesa di una risposta che possono essere servite dato lo stato corrente del centro di tuple.

Come già accennato in precedenza nella Sottosezione 1.4.2, lo scopo di questo elaborato è proprio quello di estendere il meta-modello proposto, introducendo nuove primitive di coordinazione per la gestione di eventi spaziali, come ad esempio l'ingresso o l'uscita da una stanza o il raggiungimento di una data distanza da un determinato punto. Questa estensione verrà trattata nel dettaglio nei prossimi capitoli.

Capitolo 3

Coordinazione Space-Aware in ReSpecT

Nell'ambito del pervasive computing, la complessità dei sistemi computazionali rende sempre maggiori i requisiti che il middleware di coordinazione deve soddisfare. In particolare, la comparsa di numerosi dispositivi mobili dotati di sensori e coprocessori di movimento, ha portato alla necessità di poter progettare sistemi e computazioni che supportino la *space-awareness*, consapevoli del contesto spaziale, allo scopo di stabilire gli obiettivi da raggiungere e come raggiungerli.

In questo ambito nasce dunque la necessità di linguaggi e modelli di coordinazione in grado gestire gli aspetti topologici, sia virtuali che fisici, e di aumentare il livello di astrazione da tupla a *struttura spaziale di tuple*, tenendo conto che questa, oltre ad essere distribuita, potrebbe risultare dinamica e mobile. Modelli di coordinazione di questo tipo sono già stati trattati in svariati articoli come [3], [4], [7] e [15]. Questi modelli arricchiscono lo spazio di tuple classico con primitive per la distribuzione delle tuple, dai nodi ai vicini, permettendo anche che esse influiscano sul contesto del vicinato, o che ne vengano influenzate.

In questo capitolo verrà analizzato il concetto di Spatial Computing per comprendere alcuni concetti chiave dei sistemi pervasivi che si stanno prendendo in considerazione. Inoltre verrà introdotta la nozione di topologia (virtuale o fisica) e verranno discussi i modelli di coordinazione per la topologia fisica, con particolare riguardo al modello *st*ReSpecT, come estensione da realizzare al meta-modello A&A ReSpecT per il supporto alla coordinazione space-aware.

3.1 Spatial Computing

Negli ultimi anni si è verificata la nascita di svariate tipologie di reti applicabili nell'ambito della computazione distribuita: *sensor networks*, *ubiquitous networks* e *global networks*. Nonostante le differenze, in struttura e comportamento, tra queste reti è stato possibile estrarre alcune caratteristiche generali che le distinguono da quelle tradizionali:

larga scala – il numero di nodi, quindi di componenti, coinvolti nelle applicazioni distribuite risulta essere elevato a causa della decentralizzazione del controllo.

dinamicità della rete – le attività dei componenti vengono svolte in una rete la cui struttura cambia nel tempo ed in maniera del tutto impredicibile a causa di fallimenti nelle computazioni e/o a causa della mobilità dei nodi.

situatedness – le attività dei componenti risultano essere fortemente relazionate con la loro locazione in un ambiente che può essere sia fisico che virtuale.

In particolare, le prime due caratteristiche richiedono che il sistema sia in grado di auto-organizzarsi ed auto-adattarsi (proprietà di *self-**), mentre la terza richiede un approccio che ponga l'ambiente, la sua distribuzione spaziale e la sua dinamicità, come elementi chiave per la modellazione del sistema. Risulta chiaro quindi che queste caratteristiche sono strettamente correlate tra loro poiché la proprietà di self-* non può astrarre dalla capacità del sistema di essere *context-aware*.

Allo scopo di affrontare questi aspetti, è stato introdotto il concetto di *Spatial Computing*, definendo la rete come uno spazio continuo e modellando le attività in termini di osservazione delle proprietà spaziali del sistema. Con questo approccio si è in grado di gestire la dinamicità della rete nei sistemi su larga scala, effettuare l'integrazione delle varie proprietà di self-* nei sistemi distribuiti e modellare sistemi le cui attività risultino situate nell'ambiente.

Più nello specifico, con riferimento alle caratteristiche descritte sopra, lo Spatial Computing risolve molte problematiche legate alle proprietà dei sistemi distribuiti [6]:

larga scala – la dimensione della rete non influenza i modelli, che restano gli stessi sia per reti piccole che per reti su larga scala.

dinamicità della rete – il dinamismo della rete non viene percepito direttamente dai componenti del sistema, ma resta nascosto dietro una struttura spaziale che viene mantenuta invariata nonostante le variazioni che si verificano nella topologia della stessa.

situatedness – lo spazio viene relazionato concettualmente con il concetto di ambiente. Questo permette di gestire le problematiche relative a quei sistemi le cui attività risultano strettamente correlate con l'ambiente nel quale il sistema è immerso.

Infine, lo spatial computing supporta e promuove la proprietà di self-*.

Quanto descritto, permette di affermare che il concetto di Spatial Computing è un'astrazione necessaria per la definizione di un nuovo modello che sia in grado di risolvere le problematiche di sistemi complessi e al tempo stesso distribuiti, eterogenei e la cui topologia cambia costantemente, poiché composti da dispositivi mobili, che sono in grado di comunicare con l'ambiente che li circonda e che rendono il sistema fortemente dinamico e variabile in dimensioni.

In merito a quanto detto, in [1] viene definito uno *Spatial Computer* come *"una collezione di dispositivi computazionali distribuiti in uno spazio fisico, nel quale la difficoltà di spostare le informazioni tra due qualsiasi dispositivi è fortemente dipendente dalla distanza tra essi, e gli obiettivi funzionali del sistema sono generalemente definiti in termini di struttura spaziale del sistema."*

I linguaggi per lo Spatial Computing (Spatial Computing Languages, SCL) di occupano quindi di collegare il comportamento aggregato di molteplici dispositivi indipendenti e situati al modo in cui essi vengono programmati. Inoltre, gli SCL dovrebbero fornire ai programmatori opportuni costrutti per programmare l'assemblaggio di dispositivi, mentre il compito di tradurre tali costrutti in funzioni eseguibili localmente, dovrebbe essere lasciato all'interprete in esecuzione su ogni dispositivo [8].

3.2 Topologia Virtuale e Fisica

Come già accennato all'inizio di questo capitolo, vi è la necessità di gestire le problematiche legate alla topologia del sistema, che sia essa virtuale o fisica. Da questa affermazione si deduce l'esistenza di due approcci possibili, uno orientato alla topologia virtuale ed uno a quella fisica.

L'*approccio virtuale* si affida ad un'infrastruttura di comunicazione e localizzazione, nella quale l'obiettivo è quello di creare un modello virtuale del mondo fisico appoggiandosi ad una piattaforma di servizio. Per agenti e artefatti mobili, questo implica l'obbligo di comunicare regolarmente alla piattaforma gli aggiornamenti della loro posizione, allo scopo di mantenere il modello consistente con lo stato del mondo fisico.

Questo approccio può causare la nascita di problemi di scalabilità, infatti la piattaforma di servizio fornisce funzionalità a tutti gli utenti, quindi, se il numero di artefatti ed agenti mobili risulta troppo elevato, la piattaforma diventa un collo di bottiglia per la comunicazione.

Con riferimento al linguaggio di coordinazione ReSpecT, applicando un approccio virtuale, le tuple e le operazioni esistono solo in uno specifico nodo della rete e gli agenti, spostandosi, non trasportano anche il centro di tuple, quindi esso deve essere mantenuto visibile qualora gli agenti lo necessitino.

L'*approccio fisico* si basa invece su un protocollo di coordinazione tra agenti e artefatti dotati di dispositivi wireless, tramite i quali essi sono in grado di comunicare solo con i dispositivi presenti nel loro raggio di comunicazione. In questo caso dunque la coordinazione avviene solo nel *vicinato* e la variazione della topologia può essere rilevata anche senza un modello globale del mondo fisico.

Basandosi sulla coordinazione locale tra i dispositivi, questo approccio non presenta problemi di scalabilità in quanto il numero di agenti e artefatti mobili risulta localmente limitato.

Con riferimento al linguaggio di coordinazione ReSpecT, applicando un approccio fisico, anche il centro di tuple può spostarsi all'interno del sistema, quindi tuple ed operazioni possono esistere in qualsiasi nodo della rete ed il centro di tuple deve essere considerato a tutti gli effetti come un'entità mobile.

Nel seguito, l'attenzione verrà focalizzata sulla topologia fisica del sistema, definendo in breve i modelli già esistenti e concentrandosi sull'introduzione di un nuovo modello per gestire le problematiche relative a questo aspetto, tramite la specifica di nuove primitive e operazioni all'interno del linguaggio ReSpecT.

3.3 Modelli per la Topologia Fisica

La necessità di linguaggi e modelli utili per gestire le problematiche relative agli aspetti topologici del sistema, ha portato all'introduzione, in letteratura, di svariati modelli tramite i quali si fosse in grado di supportare l'adattatività e la reattività del sistema ai cambiamenti che avvengo in ambienti distribuiti e fortemente eterogenei quali sono le reti esistenti oggi.

In questa sezione si discuterà in breve di alcuni modelli orientati alla topologia fisica del sistema quali: $\sigma\tau$LINDA, GEO-LINDA e TOTA.

3.3.1 $\sigma\tau$LINDA

Il modello $\sigma\tau$LINDA (Space-Time Linda) [15], estensione del modello LINDA, è nato allo scopo di colmare il divario tra la coordinazione e l'idea di spatial computing. In questo modello, gli agenti sono entità situate che interagiscono tramite processi, chiamati *space-time activities*, che manipolano la configurazione spazio-temporale delle tuple nella rete.

L'infrastruttura di coordinazione è composta da nodi che ospitano gli agenti e uno spazio di tuple; ogni nodo è in grado di interagire con i nodi nel vicinato e segue un *ciclo di computazione* ben definito e descritto in [15]. La vicinanza può essere vista come una proprietà fisica oppure virtuale, caratteristica che rende $\sigma\tau$LINDA adatto sia per gestire problemi legati alla topologia fisica che quelli associati a quella virtuale.

Differentemente da LINDA, in questo caso gli agenti interagiscono tramite l'iniezione di *space-time activities* e non più tramite azioni atomiche (in, out, read). Queste attività sono processi facenti parte di azioni atomiche Linda-like con costrutti addizionali che permettono all'attività stessa di diffondersi nello spazio e nel tempo.

Infine, $\sigma\tau$LINDA introduce nuovi operatori spaziali e temporali:

- `neigh` P: consente di inviare un messaggio contenente P a tutto il vicinato, il quale eseguirà l'attività P al ciclo ci computazione successivo.
- `$distance`: valuta la distanza tra il nodo che invia il messaggio e quello che lo riceve.
- `$orientation`: utilizzato per stimare la direzione relativa tra il mittente e il destinatario.
- `next` P: schedula il processo (attività) P per l'esecuzione al prossimo ciclo di computazione.
- `$delay`: valuta l'ammontare di tempo trascorso tra il ciclo di computazione corrente ed il precedente.
- `$this`: può essere utilizzato per accedere all'identificatore dello spazio di coordinazione corrente.

3.3.2 Geo-Linda

Il modello Geo-Linda [4], altra estensione di Linda, propone di gestire gli aspetti spaziali riflettendo le proprietà fisiche in uno spazio di tuple mobile presente su ogni dispositivo. Lo spazio di tuple visibile da ogni dispositivo corrisponde quindi all'unione degli spazi di tuple locali dei dispositivi inclusi nel raggio di comunicazione.

Con Geo-Linda è possibile rilevare precisamente differenti pattern di movimento dei dispositivi. A questo scopo, il modello associa un volume geometrico ad ogni tupla, chiamato *tuple's shape*, ed uno ad ogni operazione di lettura, chiamato *addressing shape*. Una qualsiasi operazione di lettura viene eseguita quando la "shape" della tupla, che corrisponde al tuple template specificato, interseca l'addressing shape dell'operazione. Grazie all'utilizzo dei volumi è possibile quindi definire una *configurazione geometrica* di un pattern di movimento.

Poiché le operazioni fornite da Linda non permettono di rilevare specifici pattern di movimento [4], Geo-Linda propone due nuove operazioni di lettura: `lostOne` e `readOnce`. Più nello specifico, il modello fornisce quattro operazioni di lettura:

- `read(s,p)`: restituisce una tupla che corrisponde al pattern `p` e la cui shape interseca l'addressing shape `s`.
- `take(s,p)`: restituisce una tupla che corrisponde al pattern `p` e la cui shape interseca l'addressing shape `s`. La tupla viene rimossa dallo spazio di tuple.
- `readOnce(s,p)`: restituisce una *nuova* tupla che corrisponde al pattern `p` e la cui shape interseca l'addressing shape `s`. Questa tupla non è stata già letta con l'operazione `readOnce`.
- `lostOne(s,p)`: restituisce una tupla che corrisponde al pattern `p`. La shape della tupla restituita deve intersecare l'addressing shape `s` e dev'essere scomparsa dallo spazio di tuple visibile all'entità che ha invocato l'operazione. La tupla corrispondente al pattern deve essere stata letta in precedenza con l'operazione `readOnce`.

Tramite queste quattro operazioni di lettura, con Geo-Linda è possibile rilevare svariati pattern di movimento (descritti in [4]), quali l'incontro tra due dispositivi, tramite l'operazione `read`, i cambiamenti di stato di un dispositivo, tramite l'operazione `take` e l'arrivo o la partenza delle entità, tramite le operazioni `readOnce` e `lostOne`.

3.3.3 TOTA

Il middleware TOTA (Tuples On The Air) [7] si basa su uno spazio di tuple distribuito nel quale ogni tupla è caratterizzata, oltre che dal suo contenuto, da due ulteriori elementi: *propagation rule* e *maintenance rule*.

Le tuple in TOTA sono quindi caratterizzate da un contenuto `C`, da una propagation rule `P`, e da una maintenance rule `M`, ed assumono la forma [7]:

```
T = (C,P,M)
```

Dove:

- `C`: è un insieme ordinato di campi tipati rappresentante le informazioni trasportate dalla tupla e tramite il quale un agente è in grado di accedere alle tuple utilizzando il tipico meccanismo di pattern-matching.
- `P`: è la *propagation rule* che determina come la tupla dovrebbe essere distribuita e propagata sulla rete. Questa regola include anche la specifica dello "scope" della tupla e come la propagazione possa essere influenzata dalla presenza o dall'assenza di altre tuple nel sistema. Inoltre, specifica anche come il contenuto della tupla possa variare.
- `M`: è la *maintenance rule* che determina come la tupla dovrebbe reagire al trascorrere del tempo e/o agli eventi che si verificano nell'ambiente. Da un lato, questa regola è in grado di preservare la struttura spaziale delle tuple (come specificato dalla propagation rule) a prescindere dalle dinamiche della rete, dall'altro le tuple posso essere rese variabili con il passare del tempo per poter specificare, ad esempio, tuple temporanee o tuple che "evaporano" lentamente.

Infine, per garantire il mantenimento della struttura spaziale delle tuple, il middleware TOTA supporta la propagazione attiva e adattativa delle stesse, infatti, monitorando la topologia della rete e l'inserimento di nuove tuple, esso è in grado di ri-propagare automaticamente le tuple in base alle necessità e in risposta cambiamenti avvenuti nell'ambiente.

3.4 Modello *st*ReSpecT

I modelli descritti nella Sezione 3.3 si rivelano in grado di gestire alcune problematiche relative alla topologia fisica ma, essendo orientati alla risoluzione di un problema specifico o di un singolo dominio applicativo, risultano essere carenti di un'analisi esaustiva dei meccanismi necessari per garantire e promuovere una Coordinazione Space-Aware che sia general-purpose.

A questo scopo, in questa sezione viene considerato nuovamente il linguaggio di coordinazione ReSpecT, già esteso per la gestione del tempo (Sezione 2.1) e il supporto alla situatedness (Sezione 2.2), con l'intenzione di estenderlo ulteriormente introducendo nuovi elementi, primitive di comunicazione e meccanismi fondamentali, richiesti per gestire le problematiche spaziali.

3.4.1 Problematiche

La coordinazione spaziale richiede che il media di coordinazione supporti entrambe le proprietà di *spatial situatedness* e *spatial awareness*. Allo scopo di definire opportune regole di coordinazione space-dependent, è dunque necessario considerare entrambi questi aspetti e specificare quali sono le principali problematiche da affrontare.

Spatial Situatedness

Innanzitutto, la *situatedness* richiede che un'*astrazione di coordinazione space-aware* risulti sempre associata ad un posizionamento assoluto, che può essere

fisico, ad esempio la posizione nello spazio del dispositivo che ospita il media di coordinazione, o virtuale, come ad esempio il nodo della rete sul quale esso è in esecuzione.

Questo aspetto riguarda sia la *posizione* che qualsiasi tipo di *movimento*, che includa qualunque tipo di variazione spazio-temporale dipendente anche dalla natura dello spazio stesso. Infatti, le astrazioni software, potrebbero muoversi all'interno di uno spazio *virtuale* e *discreto* (la rete), oppure *fisico* e *continuo* (lo spazio tridimensionale), inoltre, esse potrebbero essere ospitate da un dispositivo mobile e condividere con esso la loro posizione.

Risulta chiaro quindi che un'astrazione di coordinazione deve obbligatoriamente avere la capacità di muoversi all'interno sia dello spazio fisico continuo che di quello virtuale discreto. Questo mette in evidenza due tipi di posizionamento:

posizionamento fisico – può essere sia *assoluto* (e.g. latitudine, longitudine, altitudine), *geografico* (e.g. Via Sacchi 3, Cesena, Italia), o *organizzativo* (e.g. Ufficio 2 del DISI, sede di Cesena).

posizionamento virtuale – possibile tramite un servizio di rete e può essere sia *assoluto* (e.g. indirizzo IP del nodo), che *relativo* (e.g. localizzazione del dominio tramite DNS).

L'*ontologia del linguaggio* di coordinazione dovrà includere tutti questi concetti, che risultano necessari per gestire gli aspetti spaziali in un sistema distribuito. Più in generale, con riferimento a quattro nozioni chiave, si può affermare che [8]:

località – lo spazio può essere definito sia tramite un sistema di coordinate globali, conosciuto da ogni artefatto di coordinazione, che da uno locale.

relatività – il sistema di coordinate può essere assoluto, sfruttando alcune convenzioni come i dati GPS, oppure relativo, descrivendo soltanto le relazioni spaziali tra i vari artefatti di coordinazione.

topologia – le proprietà spaziali possono essere descritte considerando sia il mondo fisico in se (e.g. "Via Genova 181, Cesena"), che un'interpretazione logica di esso (e.g. "Workspace di Andrea Omicini"), oppure una sua *visione organizzativa* (e.g. "Ufficio di Omicini in Via Sacchi 3").

granularità – le misurazioni spaziali possono essere espresse sia su una scala continua (dati GPS) che su scala discreta (numero di hop nella rete).

Spatial Awareness

Il requisito principale della *spatial awareness* è identificato dal fatto che l'ontologia del medium di coordinazione debba incapsulare la nozione di spazio. Ciò significa che le leggi di coordinazione contenute in esso, devono poter accedere alla sua posizione allo scopo di essere in grado di *ragionare sullo spazio* e quindi permettere la definizione di *leggi di coordinazione space-aware*, le quali verranno utilizzate da artefatti ed agenti per interagire e coordinarsi tenendo conto degli aspetti spaziali.

Il linguaggio di coordinazione dovrà quindi fornire un *insieme di funzioni/-predicati* per consentire l'accesso alle informazioni spaziali associate a qualsiasi

evento che si verifica nel medium di coordinazione, e per effettuare computazioni, semplici o complesse, su tali informazioni.

La nozione di spazio deve essere incapsulata anche all'interno del ciclo di lavoro del media di coordinazione, infatti il modello ad eventi dovrebbe includere gli *eventi spaziali*, i quali influenzano le attività di coordinazione innescando computazioni aggiuntive relative allo spazio.

Infine, associare informazioni spaziali agli eventi non è sufficiente, in quanto, per ottenere la massima espressività, è necessario che gli eventi spaziali vengano *generati all'interno del media di coordinazione*. In questo modo esso sarebbe in grado di catturare anche eventi relativi al movimento (come la partenza o l'arrivo in un certo luogo), utilizzati per attivare leggi di coordinazione space-aware all'interno del normale ciclo di lavoro dell'astrazione di coordinazione.

3.4.2 Centri di Tuple Spaziali

Analogamente a come i centri di tuple temporizzati potenziano i centri di tuple con l'abilità di incapsulare leggi di coordinazione temporizzate [12], i *centri di tuple spaziali* devono estendere i centri di tuple allo scopo di affrontare le problematiche spaziali enunciate nella Sottosezione precedente.

Per definire la posizione di un centro di tuple si deve introdurre la nozione di *current place*. Questa può essere relativa alla posizione assoluta nello spazio del dispositivo nel quale il media di coordinazione è in esecuzione, al nome di dominio del nodo TuCSoN che ospita il centro di tuple, oppure ad una locazione sulla mappa. In questo modo, il movimento viene rappresentato da due tipi di eventi spaziali: spostamento da una posizione di partenza e fermata ad una posizione di arrivo.

Con riferimento al modello formale definito nella Sezione 1.4 ed esteso nella Sezione 2.2, questo viene ottenuto estendendo la coda degli eventi ambientali in ingresso affinché divenga un multiset *SitE* di eventi temporali, ambientali e spaziali. Ogniqualvolta si verifica un movimento, viene quindi generato un evento spaziale che viene inserito nel multiset *SitE* e gestito da una nuova transizione chiamata *situation* ($\longrightarrow_s$).

Analogamente agli eventi temporali, è possibile specificare reazioni innescate da eventi spaziali, chiamate *reazioni spaziali* che sono caratterizzate dalla medesima semantica vista per le altre tipologie di reazioni. Grazie a questo nuovo elemento, un centro di tuple spaziale risulta in grado di reagire al movimento sia nello spazio fisico che in quello virtuale.

Infine, viene fornita anche una nozione di *località*, infatti quando viene invocata una primitiva di coordinazione senza la specifica del nodo, essa viene gestita implicitamente come riferita allo spazio di interazione locale che il nodo ospita; quando viene specificato l'identificativo di un nodo, la primitiva viene gestita come riferita allo spazio di interazione globale [14].

3.4.3 Estensione del modello

Definite le problematiche principali da affrontare per supportare la coordinazione space-aware in ReSpecT, con riferimento ad esse, nel seguito verrà discusso come tali problematiche dovranno essere gestite dall'estensione per la coordinazione space-aware, chiamata *st*ReSpecT (*Space-Time Aware* ReSpecT).

Per quanto riguarda l'ontologia del linguaggio ed in particolare la *topologia*, in quanto aspetto chiave della coordinazione space-aware, vi è l'interesse a supportare qualsiasi politica di coordinazione. Poiché la maggior parte dei sistemi software e hardware odierni hanno facile accesso ai dati GPS, anche nei sistemi distribuiti risulta possibile rendere disponibile una descrizione fisica, globale e assoluta di qualsiasi proprietà spaziale. Questo differisce dagli aspetti temporali per i quali viene spesso preferita una nozione locale di tempo.

Allo scopo di consentire la definizione di leggi di coordinazione che permettano ad agenti e artefatti di "discutere" a proposito di spazio, vi è la necessità di introdurre nuovi predicati di osservazione e nuove guardie.

In altre parole, per permettere ai singoli componenti del sistema di dialogare riguardo lo spazio fisico, devono essere definiti nuovi *eventi ammissibili*:

- `from(`⟨*Place*⟩`)`: rappresenta la reazione innescata dall'evento spaziale generato dal movimento di un'entità che *lascia* una data locazione fisica rappresentata da ⟨*Place*⟩.
- `to(`⟨*Place*⟩`)`: rappresenta la reazione innescata dall'evento spaziale generato dal movimento di un'entità che *raggiunge* una data locazione fisica rappresentata da ⟨*Place*⟩.

Questi eventi vengono definiti estendendo il predicato ⟨*TCEvent*⟩ [11], la cui nozione, così estesa, permette di fornire al medium di coordinazione il supporto intrinseco alla gestione degli eventi spaziali, dando risoluzione all'ultima problematica descritta nella Sottosezione 3.4.1. Infatti, il predicato ⟨*TCEvent*⟩, rappresenta l'abilità dell'astrazione di coordinazione di generare pro-attivamente eventi spaziali.

Unitamente a questi eventi, devono essere definiti anche nuovi *predicati di osservazione* utili per l'accesso alle proprietà spaziali e la loro valutazione. In generale tali predicati assumeranno la forma ⟨*EventView*⟩`_place` e possono essere i seguenti:

- `current_place`: valuta la locazione del centro di tuple che sta eseguendo la reazione corrente.
- `event_place`: valuta dove si è verificata la *causa diretta* scatenante la computazione corrente.
- `start_place`: valuta dove si è verificata la *causa primaria* scatenante la computazione corrente.

Inoltre, è necessario introdurre nuove *guardie* per offrire controlli sulla locazione fisica e quindi per consentire la valutazione dei reaction goals (Sottosezione 1.3.2) solo a valle della verifica di una data condizione spaziale:

- `at(`⟨*Place*⟩`)`: innesca una reazione quando il centro di tuple si trova nella locazione fisica definita da ⟨*Place*⟩.
- `near(`⟨*Place*⟩, ⟨*Radius*⟩`)`: innesca una reazione quando il centro di tuple si trova in prossimità della locazione fisica definita da ⟨*Place*⟩ a patto che essa sia entro un raggio definito da ⟨*Radius*⟩.

Tutti i costrutti definiti sopra potenziano il medium di coordinazione permettendogli, una volta programmato opportunamente, di essere a tutti gli effetti space-aware e quindi in grado di riconoscere eventi, accedere alle informazioni e valutare condizioni facenti parte della dimensione spaziale in un sistema coordinato e *situato*.

Inoltre, forniscono gli elementi essenziali per la:

valutazione delle proprietà spaziali – tramite i predicati di osservazione `current_place`, `event_place` e `start_place`.

manipolazione dello spazio – tramite i costrutti per la situatedness discussi nella Sezione 2.2.

computazione delle informazioni spaziali – tramite le guardie `at(`⟨*Place*⟩`)` e `near(`⟨*Place*⟩, ⟨*Radius*⟩`)`.

percezione degli eventi di movimento – tramite gli eventi `from(`⟨*Place*⟩`)` e `to(`⟨*Place*⟩`)`.

In particolare, per quanto riguarda la percezione degli eventi di movimento, che siano essi generati autonomamente oppure on demand dal dispositivo fisico, si può affermare che il predicato `from(`⟨*Place*⟩`)` rappresenta l'evento spaziale generato quando il dispositivo che ospita il centro di tuple *comincia a muoversi*, mentre il predicato `to(`⟨*Place*⟩`)` rappresenta l'evento spaziale generato quando lo stesso dispositivo *termina il movimento*. Quindi, risulta chiaro che questi due eventi sono pensati per reificare un *cambiamento di stato* nella dimensione spaziale della computazione.

Per completare questa trattazione, nella Tabella 3.1 viene descritta formalmente la sintassi completa del meta-modello A&A ReSpecT, che riporta già le estensioni relative a tempo e ambiente trattate nel Capitolo 2, estesa con i costrutti introdotti dall'estensione *st*ReSpecT.

Infine, per quanto concerne il concetto di *Spatial Computing*, trattato nella Sezione 3.1, si può affermare che questa estensione del linguaggio di coordinazione ReSpecT risulta adatta ad essere utilizzata come Spatial Computing Virtual Machine, sulla quale progettare ed eseguire SCL domain-specific [8].

*st*ReSpecT e Topologia Virtuale

Pur non essendo argomento di questo elaborato, è opportuno dire che l'estensione *st*ReSpecT è già orientata anche al supporto delle problematiche legate alla topologia virtuale. In questo frangente, sono stati definiti un nuovo evento ed una nuova guardia per gestire gli eventi spaziali virtuali:

- `node(`⟨*Node*⟩`)`: rappresenta la reazione innescata dall'evento spaziale generato dalla presenza di un'entità in una data locazione virtuale rappresentata da ⟨*Node*⟩.
- `on(`⟨*Node*⟩`)`: guardia che innesca una reazione quando il centro di tuple si trova nella locazione virtuale definita da ⟨*Node*⟩.

Tabella 3.1: SINTASSI DI A&A ReSpecT ESTESO CON *st*ReSpecT

⟨*TCSpecification*⟩	::=	{⟨*SpecificationTuple*⟩.}
⟨*SpecificationTuple*⟩	::=	`reaction(`⟨*TCEvent*⟩`,[`⟨*Guard*⟩`,]`⟨*Reaction*⟩`)`
⟨***TCEvent***⟩	::=	⟨*SimpleTCPredicate*⟩`(`⟨*Tuple*⟩`)` | `time(`⟨*Time*⟩`)` | ⟨*EnvPredicate*⟩ | `from(`⟨***Place***⟩`)` | `to(`⟨***Place***⟩`)`
⟨*Guard*⟩	::=	⟨*GuardPredicate*⟩ | `(`⟨*GuardPredicate*⟩ {, ⟨*GuardPredicate*⟩}`)`
⟨*Reaction*⟩	::=	⟨*ReactionGoal*⟩ | `(`⟨*ReactionGoal*⟩ {, ⟨*ReactionGoal*⟩}`)`
⟨*ReactionGoal*⟩	::=	⟨*TCPredicate*⟩ | ⟨*ObservationPredicate*⟩ | ⟨*Computation*⟩ | `(`⟨*ReactionGoal*⟩ `;` ⟨*ReactionGoal*⟩`)`
⟨*TCPredicate*⟩	::=	⟨*SimpleTCPredicate*⟩ | ⟨*TCLinkPredicate*⟩ | ⟨*TCEnvPredicate*⟩
⟨*EnvPredicate*⟩	::=	`get(`⟨*Key*⟩`,`⟨*Value*⟩`)` | `set(`⟨*Key*⟩`,`⟨*Value*⟩`)`
⟨*SimpleTCPredicate*⟩	::=	⟨*TCStatePredicate*⟩`(`⟨*Tuple*⟩`)` | ⟨*TCForgePredicate*⟩`(`⟨*SpecificationTuple*⟩`)`
⟨*TCLinkPredicate*⟩	::=	⟨*TCIdentifier*⟩ `?` ⟨*SimpleTCPredicate*⟩
⟨*TCEnvPredicate*⟩	::=	⟨*EnvResIdentifier*⟩ `?` ⟨*EnvPredicate*⟩
⟨*TCStatePredicate*⟩	::=	`in` | `inp` | `rd` | `rdp` | `out` | `no`
⟨*TCForgePredicate*⟩	::=	⟨*TCStatePredicate*⟩`_s`
⟨*ObservationPredicate*⟩	::=	⟨*EventView*⟩`_`⟨*EventInformation*⟩`(`⟨*Tuple*⟩`)` | `env(`⟨*Key*⟩`,`⟨*Value*⟩`)`
⟨*EventView*⟩	::=	`current` | `event` | `start`
⟨***EventInformation***⟩	::=	`predicate` | `tuple` | `source` | `target` | `time` | `place`
⟨***GuardPredicate***⟩	::=	`request` | `response` | `success` | `failure` | `endo` | `exo` | `intra` | `inter` | `from_agent` | `to_agent` | `from_tc` | `to_tc` | `from_env` | `to_env` | `before(`⟨*Time*⟩`)` | `after(`⟨*Time*⟩`)` | `at(`⟨***Place***⟩`)` | `near(`⟨***Place***⟩ ⟨***Radius***⟩`)`
⟨*Computation*⟩	è	un goal Prolog-like che esegue computazioni logiche e/o aritmetiche
⟨*Time*⟩	è	un intero non negativo
⟨*Tuple*⟩, ⟨*Key*⟩, ⟨*Value*⟩	sono	termini Prolog
⟨***Place***⟩	**è**	**un termine ground rappresentante un punto in uno spazio assoluto**
⟨***Radius***⟩	**è**	**un intero non negativo rappresentate la distanza**

3.4.4 Rivisitazione sintattica

Recentemente in [10] è stata effettuata una rivisitazione sintattica del modello descritto nella Sezione precedente e la cui sintassi formale è mostrata nella Tabella 3.1.

Questa rivisitazione non va a modificare la semantica del modello definito in precedenza ma fornisce solo una sintassi ridotta e più intuitiva allo scopo di semplificarne la comprensione, anche relativamente alla gestione degli eventi spaziali che si verificano nel centro di tuple.

Innanzitutto, le *reazioni* assumeranno d'ora in avanti la forma

`reaction(`*`Activity`*`,`*`Guards`*`,`*`Goals`*`)`

che, analogamente a prima, specifica la lista di operazioni (*`Goals`*) da eseguire quando si verifica un dato evento, causato da una *`Activity`*, e alcune condizioni risultano soddisfatte (*`Guards`*).

Per quanto riguarda i *predicati di osservazione* `current_place`, `event_place` e `start_place`, utili per il recupero delle informazioni spaziali relative ad un evento che si verifica all'interno di una reazione, la loro nuova sintassi è la seguente:

- `current_place(@`*`S`*`,?`*`P`*`)`: ha successo se *`P`* unifica con la posizione del nodo al quale il centro di tuple appartiene.
- `event_place(@`*`S`*`,?`*`P`*`)`: ha successo se *`P`* unifica con la posizione del nodo nel quale è stato originato l'evento scatenante la computazione corrente (*causa diretta*).
- `start_place(@`*`S`*`,?`*`P`*`)`: ha successo se *`P`* unifica con la posizione del nodo nel quale è stata originata la catena di eventi che porta all'evento scatenante la computazione corrente (*causa primaria*).

In questo caso la locazione *`P`* del nodo può essere specificata come:

- `S=ph`: posizione fisica assoluta.
- `S=ip`: indirizzo IP.
- `S=dns`: nome di dominio.
- `S=map`: posizione geografica (tipicamente definita da servizi come Google Maps).
- `S=org`: posizione organizzativa (posizione all'interno di una topologia virtuale organizzativa).

Relativamente ai *predicati di guardia*, utilizzati per selezionare le reazioni da innescare sulla base delle proprietà degli eventi spaziali, la loro nuova sintassi è la seguente:

- `at(@`*`S`*`,@`*`P`*`)`: ha successo quando il centro di tuple è in esecuzione alla posizione *`P`*, specificata in accordo con *`S`*.
- `near(@`*`S`*`,@`*`P`*`,@`*`R`*`)`: ha successo quando il centro di tuple è in esecuzione in una posizione contenuta nella regione spaziale avente centro *`P`* e raggio *`R`*, specificata in accordo con *`S`*.

Infine, per quanto riguarda gli *eventi ammissibili*, generati in risposta agli eventi spaziali, la loro nuova sintassi è la seguente:

- `from(?S,?P)`: corrisponde ad un evento spaziale generato quando il dispositivo che ospita il centro di tuple comincia a muoversi dalla posizione `P`, specificata in accordo con `S`.
- `to(?S,?P)`: corrisponde ad un evento spaziale generato quando il dispositivo che ospita il centro di tuple termina il movimento e raggiunge la posizione `P`, specificata in accordo con `S`.

Grazie all'introduzione del parametro `S`, i predicati specifici per la topologia virtuale, definiti nella Sottosezione precedente, non sono più necessari in quanto con tale parametro è possibile specificare la posizione del nodo sia in termini fisici che virtuali a seconda delle necessità.

Nel seguito di questo elaborato si farà riferimento alla sintassi appena descritta e riportata formalmente nella Tabella 3.2. Nonostante questo, si tenga presente che l'implementazione corrente di ReSpecT utilizza la sintassi mostrata nella Tabella 3.1, motivo per il quale è stato deciso di riportarle entrambe.

Tabella 3.2: NUOVA SINTASSI DI A&A ReSpecT *(senza forgeability)*

⟨*Specification*⟩	::=	{⟨*Reaction*⟩.}
⟨*Reaction*⟩	::=	`reaction(`⟨*Activity*⟩,[⟨*Guards*⟩,]⟨*Goals*⟩`)`
⟨*Activity*⟩	::=	⟨*Operation*⟩ | ⟨*Situation*⟩
⟨*Operation*⟩	::=	`out(`⟨*Tuple*⟩`)` | `(in | inp | rd | rdp | out | no | nop) (`⟨*Template*⟩ [, ⟨*Term*⟩]`)`
⟨*Situation*⟩	::=	`time(`⟨*Time*⟩`)` | `env(`⟨*Key*⟩,⟨*Value*⟩`)` | `to(`⟨*Space*⟩,⟨*Place*⟩`)` | `from(`⟨*Space*⟩,⟨*Place*⟩`)`
⟨*Guards*⟩	::=	⟨*Guard*⟩ | `(`⟨*Guard*⟩ {, ⟨*Guard*⟩}`)`
⟨*Guard*⟩	::=	`request | response | success | failure | endo | exo | intra | inter | from_agent | to_agent | from_tc | to_tc | from_env | to_env | before(`⟨*Time*⟩`) | after(`⟨*Time*⟩`) | at(`⟨*Space*⟩,⟨*Place*⟩`) | near(`⟨*Space*⟩,⟨*Place*⟩,⟨*Radius*⟩`)`
⟨*Goals*⟩	::=	⟨*Goal*⟩ | `(`⟨*Goal*⟩ {, ⟨*Goal*⟩}`)`
⟨*Goal*⟩	::=	[⟨*TupleCentre*⟩ `?`] ⟨*Operation*⟩ | ⟨*EnvRes*⟩ `(<- | ->)env(`⟨*Key*⟩,⟨*Value*⟩`)` | ⟨*Observation*⟩ | ⟨*Computation*⟩ | `(`⟨*Goal*⟩ ; ⟨*Goal*⟩`)`
⟨*Observation*⟩	::=	⟨*Selector*⟩_⟨*Focus*⟩
⟨*Selector*⟩	::=	`current | event | start`
⟨*Focus*⟩	::=	`(activity | source | target) (` ⟨*Term*⟩ | `time(`⟨*Time*⟩ `) | place(`⟨*Space*⟩,⟨*Term*⟩`)`
⟨*Space*⟩	::=	`ph | ip | dns | map | org`
⟨*Computation*⟩	è	un goal Prolog-like che esegue computazioni logiche e/o aritmetiche
⟨*Time*⟩	è	un intero non negativo
⟨*Tuple*⟩,⟨*Key*⟩,⟨*Value*⟩,⟨*Term*⟩	sono	termini Prolog
⟨*Place*⟩	è	un termine ground rappresentante un punto in uno

Tabella 3.2: continua nella prossima pagina

Tabella 3.2: continua dalla pagina precedente

		spazio assoluto
⟨*Radius*⟩	è	un intero non negativo rappresentate la distanza

3.4.5 Semantica formale

Formalizzando la semantica introdotta con l'estensione space-aware di ReSpecT, devono essere considerati due principali cambiamenti. Innanzitutto, si deve definire un nuovo modello degli eventi generalizzato che includa, per ogni tipo di evento, sia eventi che informazioni spaziali. Inoltre, la transizione *environment*, già definita nella Sezione 2.2 come in grado di gestire sia eventi ambientali che temporali, deve essere estesa per includere anche gli eventi spaziali.

Modello degli Eventi

La prima estensione al modello degli eventi consiste nell'introduzione della nozione di ⟨*Activity*⟩ *spaziale*. In particolare, il concetto di ⟨*Situation*⟩ viene esteso con due attività spaziali quali `from(`⟨*Space*⟩, ⟨*Place*⟩`)`, rappresentante la traiettoria di movimento iniziale, e `to(`⟨*Space*⟩, ⟨*Place*⟩`)`, rappresentante quella finale.

Inoltre, allo scopo di incapsulare le proprietà spaziali all'interno di tutti gli eventi ReSpecT, è stata estesa anche la loro struttura introducendo il termine ⟨*Place*⟩, con il quale sarà possibile effettuare una *qualificazione spaziale* degli eventi. In questo modo tutti gli eventi ReSpecT risultano qualificati sia in termini di tempo che di spazio. L'estensione appena descritta viene mostrata nella Tabella 3.3.

Tabella 3.3: Estensione della sintassi degli Eventi ReSpecT

⟨*Event*⟩	::=	⟨*StartCause*⟩, ⟨*Cause*⟩, ⟨*Evaluation*⟩
⟨*StartCause*⟩, ⟨*Cause*⟩	::=	⟨*Activity*⟩, ⟨*Source*⟩, ⟨*Target*⟩, ⟨*Time*⟩, ⟨*Space*⟩
⟨*Source*⟩, ⟨*Target*⟩	::=	⟨*AgentId*⟩ | ⟨*TCId*⟩ | ⟨*EnvResId*⟩ | ⊥
⟨*Evaluation*⟩	::=	⊥ | {⟨*Result*⟩}
⟨*Place*⟩	::=	⟨*GPCCoordinates*⟩, ⟨*IPAddress*⟩, ⟨*DomainName*⟩, ⟨*MapLocation*⟩, ⟨*VirtualPosition*⟩

Sistema di Transizione

Analogamente a quanto discusso nella Sezione 2.2, lo stato di un centro di tuple A&A ReSpecT, considerando anche l'estensione space-aware, viene espresso dalla quadrupla seguente:

$$OpE,SitE\langle Tu, \Sigma, Re, Op\rangle_{n}^{OutE}$$

Dove:

- *Tu* ed Σ sono multiset contenenti rispettivamente le tuple ordinarie e le tuple di specifica;

- *Re* è il multiset delle reazioni innescate dagli eventi e in attesa di essere eseguite.
- *Op* è il multiset delle richieste in attesa di una risposta.
- *OpE* è il multiset degli eventi $\langle Operation \rangle$ in ingresso. Questo multiset viene automaticamente esteso ogni volta che un nuovo evento viene ricevuto dal centro di tuple.
- *OutE* è il multiset degli eventi in uscita. Questo multiset viene automaticamente svuotato emettendo gli eventi di uscita.
- *SitE* è il multiset degli eventi $\langle Situation \rangle$ in ingresso che include anche eventi temporali, spaziali e ambientali. Analogamente ad *OpE*, questo multiset viene automaticamente esteso ogni volta che un nuovo evento viene ricevuto dal centro di tuple.
- n è il tempo locale del centro di tuple.

In particolare, *SitE* rappresenta tutti gli eventi relativi alla *situatedness*, infatti verranno aggiunti eventi a questo multiset ogni volta che si verifica un evento ambientale, generato da un trasduttore, temporale, generato dal passare del tempo, o spaziale, generato da qualsiasi tipo di movimento.

Analogamente a quanto fatto in precedenza, anche in questo caso si può definire il comportamento formale di un centro di tuple come un sistema di transizione composto da quattro transizioni:

reaction – se $Re \neq \emptyset$, le reazioni innescate presenti in *Re* vengono eseguite tramite una transizione di tipo *reaction* ($\longrightarrow_r$).

situation – se $Re = \emptyset$ e $SitE \neq \emptyset$, le reazioni ambientali, temporali e spaziali in attesa di essere elaborate possono innescare altre reazioni tramite una transizione di tipo *situation* ($\longrightarrow_s$).

operation – se $Re = SitE = \emptyset$ e $\texttt{sat}(Op, Tu, \Sigma) \neq \emptyset$[1], le richieste in attesa di una risposta possono essere servite tramite una transizione di tipo *operation* ($\longrightarrow_o$).

log – se $Re = SitE = \texttt{sat}(Op, Tu, \Sigma) = \emptyset$ e $OpE \neq \emptyset$, le richiese in coda nella *OpE* possono essere recuperate (*logged*) sequenzialmente da un centro di tuple tramite una transizione di tipo *log* ($\longrightarrow_l$).

Tramite la transizione *situation*, che innesca reazioni in risposta agli eventi spaziali, anche questi possono essere catturati ed elaborati dal centro di tuple allo stesso modo degli eventi temporali e ambientali. Quindi, tale transizione risulta essere fondamentale e, a questo punto, completa di tutti gli elementi necessari per supportare appieno la situatedness.

[1] $\texttt{sat}(Op, Tu, \Sigma)$ è il sottoinsieme di *Op* delle richieste in attesa di una risposta che possono essere servite dato lo stato corrente del centro di tuple.

Capitolo 4

Space-Aware ReSpecT

Nei capitoli precedenti sono stati descritti gli aspetti concettuali fondamentali da affrontare per estendere il linguaggio di specifica ReSpecT verso la *space-awareness* e quindi verso il supporto completo alla *situatedness*. In questo capitolo vengono innanzitutto spiegate le motivazioni che risiedono alla base di tale estensione e, dopo aver mostrato il funzionamento implementativo di TuCSoN e della RespectVM allo scopo di comprenderne le dinamiche di funzionamento, verranno spiegati in dettaglio la progettazione e l'implementazione dei concetti discussi del capitolo precedente. Questi ultimi dovranno estendere il comportamento stesso del media di coordinazione affinché esso risulti in grado di gestire gli eventi spaziali legati alla posizione e al movimento delle entità all'interno di sistemi distribuiti, dinamici ed eterogenei, la cui topologia varia continuamente.

4.1 Motivazioni

Le problematiche spaziali sono tuttora aspetti critici nelle nuove classi di sistemi software complessi, come ad esempio quelli pervasivi, multi-agente e auto-organizzanti. Comprendere a fondo i meccanismi di base della coordinazione basata sullo spazio risulta essere un aspetto chiave per i modelli e linguaggi di coordinazione allo scopo di riuscire, con tali sistemi, a gestire l'interazione situata in topologie aventi una struttura spazio-temporale dinamica. L'infrastruttura di coordinazione TuCSoN e il linguaggio di specifica ReSpecT risultano essere adatti per gestire questi aspetti. Tramite i centri di tuple TuCSoN, infatti, gli agenti possono coordinarsi ed interagire mantenendo sempre il disaccoppiamento temporale e spaziale, tramite l'inserimento delle informazioni all'interno dello spazio di tuple. Inoltre, i centri di tuple TuCSoN possono essere programmati con uno specifico comportamento in risposta agli eventi che vengono generati all'interno del sistema; questo aspetto risulta molto importante per definire computazioni specifiche da effettuare in risposta ad eventi spaziali e temporali, che siano essi generati internamente o esternamente alla macchina virtuale ReSpecT. Infine, viene fornito intrinsecamente anche il supporto alla comunicazione distribuita, permettendo l'interazione tra centri di tuple e agenti, localizzati in diversi nodi della rete. Quest'ultima caratteristica è proprio quella che rende necessaria un'estensione del middleware di coordinazione per far si che agenti e

artefatti possano dialogare a proposito di spazio in sistemi complessi aventi una topologia dinamica ed eterogenea.

Ispirandosi alle considerazioni effettuate nei capitoli precedenti, nel seguito si vuole definire una versione *space-aware* di TuCSoN e ReSpecT estendendo quella originale con le astrazioni e i meccanismi per la gestione degli eventi spaziali. In particolare, si vuole fornire il completo supporto alla *situatedness*, affiancando l'estensione spaziale, alle estensioni temporale e ambientale già presenti ed analizzate nel Capitolo 2.

4.2 RespectVM e TuCSoN

Prima di mostrare l'estensione spaziale di ReSpecT dai punti di vista di requisiti, progettazione e implementazione, è necessario analizzare il comportamento di TuCSoN e della RespectVM, per meglio comprendere come essi reagiscono all'invocazione delle primitive e come agiscono durante l'elaborazione delle richieste, degli eventi e delle reazioni. In questa sezione si analizzeranno il flusso delle chiamate, la loro dinamica e, in generale, il contesto che si sta andando a considerare.

Dinamica della comunicazione

In generale, un'agente TuCSoN, posto in uno specifico nodo della rete, effettua una richiesta che viene pervenuta da un centro di tuple TuCSoN, situato nello stesso nodo o in remoto. Quest'ultimo elabora la richiesta effettuando le relative operazioni ed infine risponde al chiamante restituendo il risultato.

Da un lato, per effettuare una richiesta, l'agente TuCSoN invoca la funzione corrispondente alla primitiva desiderata e quindi attende una risposta dal centro di tuple contattato tramite un thread di controllo. Dall'altro lato, il centro di tuple TuCSoN attende che gli pervengano le richieste dagli agenti attraverso un altro thread specifico e adibito a tale scopo.

Sia che il centro di tuple si trovi nello stesso nodo dell'agente che su un nodo remoto, il protocollo utilizzato per stabilire la connessione tra un agente A ed uno specifico centro di tuple T è basato su TCP ed è suddiviso in tre fasi, che sono:

I. A deve autenticarsi nel sistema; se ha successo, viene creata un'entità chiamata `ACCProxyAgentSide`, che ha il compito di mantenere la comunicazione. L'autenticazione è valida a livello di sistema ed è quindi necessaria solo la prima volta che l'agente effettua una richiesta.

II. L'entità `ACCProxyAgentSide` ha il compito di comunicare con un processo, chiamato `WelcomeAgent`, in esecuzione lato nodo. Tale processo attende che gli pervengano richieste, le quali vengono poi direzionate verso un'altra entità chiamata `ACCProvider`. Quest'ultima ha il compito di analizzare tipo, contenuto e chiamante della richiesta e, nel caso tutto sia corretto, si occupa di creare una `ACCProxyNodeSide` avente il compito di mantenere attiva la comunicazione con l'agente A. In questa fase viene quindi stabilito un canale di comunicazione tra `ACCProxyAgentSide` e `ACCProxyNodeSide` che verrà utilizzato per l'interazione tra l'agente ed il centro di tuple.

III. Se *A* avesse la necessità di interagire con un altro centro di tuple, utilizzerà lo stesso `ACCProxyAgentSide`, ripetendo la procedura descritta sopra a partire dalla comunicazione con l'entità `WelcomeAgent` e mantenendo attivo un canale per ogni centro di tuple contattato.

Per quanto riguarda il comportamento interno di un nodo TuCSoN, esso ha il compito di controllare se una richiesta è indirizzata a lui stesso, oppure ad un altro nodo. Nel secondo caso, la richiesta viene indirizzata al centro di tuple corretto mentre, nel primo caso, l'operazione corrispondente viene inserita nella coda degli eventi in ingresso e viene successivamente recuperata dal *ReSpecT Engine*, il quale ne estrarrà l'evento da gestire. Il risultato viene poi inserito nella coda degli eventi in uscita, dalla quale l'`ACCProxyNodeSide` recupererà l'informazione con cui creare il messaggio di risposta. La presenza delle entità `ACCProxyNodeSide` e `ACCProxyAgentSide` permette al centro di tuple di gestire più primitive sospensive contemporaneamente.

Flusso delle operazioni

Considerando stabilita la connessione tra l'agente e il centro di tuple, e quindi tra `ACCProxyAgentSide` e `ACCProxyNodeSide`, dal punto di vista dell'agente, la richiesta di eseguire una specifica primitiva viene realizzata tramite l'invio di un messaggio, `TucsonMsgRequest`, direttamente al centro di tuple da contattare. Questo messaggio viene costruito partendo dall'istanza della classe `TucsonOperation` relativa alla primitiva richiesta. Le principali operazioni che permettono l'invio di un messaggio sono incapsulate nel metodo `doOperation`, definito all'interno della classe `ACCProxyAgentSide`.

Inviato il messaggio, l'agente attende una risposta dal centro di tuple contattato, `TucsonMsgReply`, grazie ad un thread di controllo. Non appena viene pervenuta una risposta, questo processo controlla se l'esecuzione è stata terminata con successo e se l'informazione ottenuta risulta coerente con la richiesta effettuata. In tal caso, lo stesso processo si occupa di segnalarlo all'`ACCProxyAgentSide`, che è ancora sospeso in attesa della risposta, il quale restituisce il risultato dell'operazione all'agente.

Dal punto di vista del nodo, la gestione delle primitive può essere suddivisa in due fasi che vengono elaborate principalmente dall'`ACCProxyNodeSide` e dalla `RespectVM`.

L'`ACCProxyNodeSide` legge ciclicamente sul canale d'ingresso rimanendo in attesa di richieste da parte degli agenti o dei centri di tuple. Pervenuta una richiesta, esso si occupa di controllarne il tipo e di invocare una funzione per inserire l'evento corrispondente nella coda degli eventi in ingresso. Sinteticamente, il flusso delle operazioni dal punto di vista dell'`ACCProxyNodeSide` risulta il seguente:

$$ACCProxyNodeSide \rightarrow TupleCenterContainer \rightarrow OrdinaryAsynchInterface \rightarrow RespectTC \rightarrow RespectVMContext$$

Nel complesso il flusso delle operazioni, a partire dalla richiesta fino alla creazione dell'evento corrispondente, è riportato nella Tabella 4.1.

Tabella 4.1: FLUSSO DELLE OPERAZIONI (NODO)

`AbstractTucsonAgent`	Recupero del contesto e invocazione della primitiva. Viene generato un messaggio di richiesta.
↓	
`ACCProxyAgentSide`	Viene eseguito il metodo `doOperation` nel quale avviene la creazione della sessione di comunicazione e del messaggio di richiesta (`TucsonMsgRequest`) relativo all'operazione invocata dall'agente. Tale messaggio viene inviato tramite protocollo TCP all'`ACCProxyNodeSide`.
↓	
`ACCProxyNodeSide`	Ricezione del messaggio di richiesta ed invocazione dell'operazione corrispondente, la quale può essere sia bloccante che non bloccante. Tale distinzione viene effettuata invocando rispettivamente i metodi `doBlockingOperatione` e `doNonBlockingOperation`, definiti all'interno dell'entità `TupleCentreContainer`.
↓	
`TupleCenterContainer`	Recupero dell'interfaccia relativa all'operazione corrispondente. A seconda che si tratti di operazioni ordinarie oppure di specifica, bloccanti (sincrone) o non bloccanti (asincrone), viene recuperata l'interfaccia che ne definisce i metodi: `OrdinarySynchInterface`, `OrdinaryAsynchInterface`, `SpecificationSynchInterface`, `SpecificationAsynchInterface`.
↓	
`(Ord|Spec)(Syn|Asyn)Inferface`	Tramite il metodo `getCore()`, si accede all'entità `RespectTC`, nella quale è incapsulata la logica applicativa di tutte le operazioni effettuabili sul centro di tuple.
↓	
`RespectTC`	Creazione dell'operazione (corrispondente alla richiesta) tramite i metodi di *make* definiti nella classe `RespectOperation` ed invocazione del metodo `doOperation` sulla `RespectVM`, la quale viene informata di chi ha effettuato la richiesta e qual'è l'operazione da eseguire.
↓	
`RespectVM`	Invocazione del metodo `doOperation` sul contesto della macchina virtuale (`RespectVMContext`) e segnalazione della presenza di nuovi eventi in ingresso.
↓	
`AbstractTupleCentreVMContext`	Incapsula la logica vera e propria del metodo `doOperation`. Nel momento in cui questo viene invocato dall'entità `RespectVM`, si procede alla creazione di nuovo evento (`InputEvent`) che viene immediatamente inserito nella coda degli eventi in ingresso. Solo dopo che questo è avvento, la `RespectVM` si occuperà di segnalare la presenza di nuovi eventi.

Parallelamente, la `RespectVM` esegue ciclicamente il metodo `execute` definito all'interno della classe `SpeakingState`. Il suo compito è quello di recuperare, uno per volta, gli eventi dalla coda di ingresso e, a seconda del tipo di operazione

richiesta, invocare le relative funzioni per ottenere i risultati. Questo flusso coinvolge nuovamente la classe `RespectVMContext` che a sua volta sfrutta le funzioni definite in `TupleSet`. Quest'ultima rappresenta la classe principale nella quale è definita la logica applicativa e quindi il comportamento di tutte le primitive. Analogamente a prima, il flusso delle operazioni dal punto di vista della `RespectVM` risulta il seguente:

$$RespectVM \rightarrow AbstractTupleCentreVMContext \rightarrow SpeakingState \rightarrow RespectVMContext \rightarrow TupleSet$$

Una descrizione più dettagliata del flusso delle operazioni dal punto di vista della `RespectVM` è riportata in Tabella 4.2.

Tabella 4.2: FLUSSO DELLE OPERAZIONI (RESPECTVM)

`RespectVM`	Viene eseguito ciclicamente il metodo `execute` definito all'interno dello `SpeakingState`, dopodiché viene controllata la presenza di eventi pendenti (tramite il metodo `pendingEvents` definito in `AbstractTupleCentreVMContext`) e, nel caso non ve ne fossero, si resta in attesa.
↓	
`SpeakingState`	All'interno del metodo `execute` avviene il recupero della coda degli eventi pendenti e, se presenti, vengono elaborati uno per volta. Viene innanzitutto controllato il tipo di operazione da effettuare e quindi vengono eseguite le computazioni corrispondenti. Infine, per la valutazione di eventuali reazioni innescate dall'evento appena elaborato, viene invocato il metodo `fetchTriggeredReactions`, definito nella classe `RespectVMContext`.
↓	
`RespectVMContext`	All'interno del metodo `fetchTriggeredReactions` viene coinvolto il motore *tuProlog* per il recupero delle reazioni innescate dall'evento appena elaborato. Se ne vengono trovate, e solo dopo aver controllato il soddisfacimento delle condizioni specificate dai predicati di guardia, viene creato un oggetto di tipo `TriggeredReaction` che viene aggiunto al multi-set (`TupleSet`) delle reazioni innescate in attesa di essere eseguite e la cui elaborazione avverrà all'interno del `ReactingState`.

4.3 Iniezione dello spazio

Un altro passaggio che risulta necessario effettuare prima di poter procedere con l'implementazione della space-awareness, comprende la definizione delle entità rappresentati lo spazio e l'iniezione di questo concetto all'interno degli eventi e della RespectVM.

4.3.1 Posizione

Prima di trattare un qualunque aspetto spaziale è necessario disporre di un'entità che rappresenti la posizione. A questo scopo, è stata definita la classe `Position` che incapsula tutte le informazioni riguardanti la posizione che sia essa virtuale o fisica. All'interno di questa classe quindi si possono trovare tutti i metodi di impostazione e recupero delle varie tipologie di posizione discusse nel capitolo precedente.

Le varie tipologie di posizione sono state definite tramite una gerarchia di classi specifica, al cui vertice si trova l'interfaccia `IPlace`, rappresentante la posizione generica (virtuale o fisica). L'interfaccia assume la seguente struttura:

Listing 4.1: IPlace.java

```
public interface IPlace {

    /**
     * @return whether this place term represent
     * a physical place
     */
     boolean isPhysical();

    /**
     * @return whether this place term represent
     * a virtual place
     */
     boolean isVirtual();

    /**
     * @return the term representation of this place
     */
     Term toTerm();

}
```

Questa interfaccia viene implementata dalle classi astratte `PhysicalPlace` e `VirtualPlace` che rappresentano, rispettivamente, la posizione fisica e quella virtuale, fornendo un costruttore che, a partire da una stringa, costruisce un termine Prolog rappresentante una specifica posizione, definita da ulteriori entità rappresentate dalle classi concrete seguenti:

- `PhPlace`: specializzazione dell'entità `PhysicalPlace` che rappresenta una posizione fisica assoluta.
- `MapPlace`: specializzazione dell'entità `PhysicalPlace` che rappresenta una posizione geografica.
- `OrgPlace`: specializzazione dell'entità `VirtualPlace` che rappresenta una posizione organizzativa.
- `IpPlace`: specializzazione dell'entità `VirtualPlace` che rappresenta una posizione virtuale assoluta.
- `DnsPlace`: specializzazione dell'entità `VirtualPlace` che rappresenta una posizione virtuale relativa.

In Figura 4.1 viene mostrata la gerarchia appena descritta.

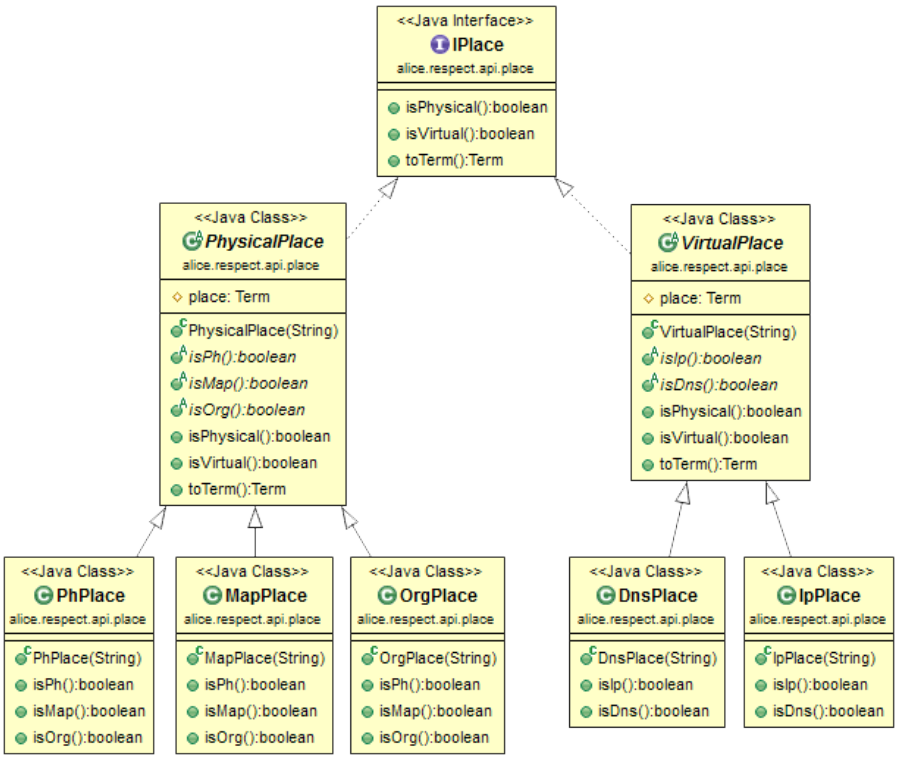


Figura 4.1: Posizione - Gerarchia

4.3.2 Eventi e spazio

Un generico evento che si verifica all'interno del sistema viene rappresentato dalla classe astratta `AbstractEvent` e, in particolare, dalle classi concrete (`InputEvent`, `OutputEvent` e `InternalEvent`) che la estendono.

Per iniettare il concetto di posizione all'interno degli eventi, quindi, si è agito direttamente sulla classe `AbstractEvent`, nella quale è stato esteso il costruttore aggiungendo un nuovo parametro rappresentate la posizione (Sezione 4.3.1) in cui si è verificato l'evento. Successivamente anche le classi concrete rappresentati le tipologie particolari di evento sono state adattate in accordo con questa modifica.

4.3.3 RespectVM e spazio

Per iniettare il concetto di posizione all'interno della macchina virtuale è stato sufficiente agire sull'entità astratta `AbstractTupleCentreVMContext`, aggiungendo un nuovo campo rappresentante la posizione (Sezione 4.3.1) e definendo i metodi per l'impostazione e il recupero di essa. La posizione della RespectVM viene poi inizializzata contestualmente all'avvio di nuova istanza della macchina virtuale e conseguentemente alla creazione di un nuovo contesto, rappresentato dalla classe `RespectVMContext`.

4.3.4 Agenti e spazio

Nonostante gli agenti TuCSoN facciano parte dello sistema nel quale è in esecuzione la RespectVM, si deve tenere presente che questi potrebbero essere in esecuzione su un nodo differente della rete. Per questo motivo devono anch'essi essere consapevoli del luogo in cui si trovano, quindi risulta necessario iniettare loro il concetto di posizione.

Analogamente a quanto effettuato per la RespectVM, si è deciso di modificare l'entità `ACCProxyAgentSide`, aggiungendo un nuovo campo rappresentante la posizione (Sezione 4.3.1) e definendo i metodi per l'impostazione e il recupero di essa. La posizione di un agente (a cui è associato un solo `ACCProxyAgentSide`) viene inizializzata durante l'avvio dell'agente stesso.

4.4 Considerazioni

Analizzando i predicati che si devono implementare per la coordinazione space-aware descritti nel capitolo precedente, per quanto concerne in particolare i predicati di osservazione `event_place(@S,?P)` e `start_place(@S,?P)`, si può notare che questi predicati impongono di essere a conoscenza della posizione dell'agente che ha richiesto una determinata operazione e quindi del luogo in cui è stato generato l'evento.

Nonostante la volontà di avere un dialogo completamente *event-driven*, l'attuale implementazione di TuCSoN è basata su una comunicazione a scambio di messaggi, all'interno dei quali vengono incapsulate solo le informazioni necessarie per discriminare, lato nodo, l'operazione richiesta e l'agente che l'ha effettuata. Infatti, la creazione dell'operazione viene effettuata concretamente solo una volta che l'entità `ACCProxyNodeSide` ha ricevuto il messaggio da parte dell'`ACCProxyAgentSide`, mentre la creazione dell'evento viene effettuata solo dopo che l'operazione giunge fino all'entità `AbstractTupleCentreVMContext` ed, in particolare, all'interno del metodo `doOperation`.

In modo tale che la macchina virtuale sia in grado di accedere alle informazioni sulla posizione nella quale è stato generato l'evento, è risultato necessario modificare l'interazione appena descritta. A tale scopo sono state valutate due diverse ipotesi di implementazione. Nella prima è stata valutata la possibilità di inserire la posizione come parametro aggiuntivo all'interno dell'entità `TucsonMsgRequest` (rappresentante il messaggio di richiesta) per poi far "fluire" tale informazione sino all'entità `AbstractTupleCentreVMContext`, nella quale viene creato l'evento.

Nella seconda ipotesi è stata valutata la possibilità di rendere l'intera interazione *event-driven* soddisfacendo sia i requisiti base di TuCSoN che quelli relativi alla coordinazione space-aware. Quest'ultima risulta essere più complicata ed invasiva, portando ad una profonda modifica del flusso esistente ma consentendo di rendere la piattaforma TuCSoN completamente event-driven. La prima ipotesi invece risulta essere più semplice e meno invasiva ma non consentirebbe l'esistenza globale della concezione di evento.

A valle di queste considerazioni, l'approccio che si è scelto di utilizzare è il secondo, infatti, nonostante risulti essere più complicato, garantirebbe una completezza che l'approccio più semplice non può fornire.

4.5 Interazione Event-driven

Nonostante sia stata effettuata l'iniezione del concetto di spazio negli eventi, non essendo possibile accedere alla posizione dell'agente, non si è in grado di inizializzare l'evento con tale informazione. Infatti, come già accennato nella sezione precedente, l'evento viene creato quando l'invocazione di una operazione di coordinazione raggiunge il centro di tuple e non quando viene richiesta all'entità `ACC` lato agente.

Con riferimento all'implementazione corrente, in fase di *completion* l'evento corrispondente viene generato dal centro di tuple. Questo è corretto in quanto è questa la prima entità TuCSoN che ne viene a conoscenza, ma non deve accadere in fase di *invocation* poiché sarebbe concettualmente opportuno che fosse l'`ACC` lato agente ad occuparsi della generazione dell'evento, essendo lui la prima entità ad esserne a conoscenza.

L'idea quindi consiste nello spostare la generazione di un evento dal momento della ricezione di una richiesta al centro di tuple, al momento di generazione della stessa da parte dell'`ACCProxyAgentSide` ed incapsulare tale evento, inizializzato anche con la posizione dell'agente, all'interno del messaggio di richiesta.

Poiché TuCSoN è stato progettato per essere distribuito sui nodi della rete, la comunicazione tra `ACCProxyAgentSide` e `ACCProxyNodeSide` avviene tramite il protocollo TCP. Questo aspetto, relativamente a ciò che si vuole ottenere, conduce inevitabilmente ad un problema tipico dei sistemi distribuiti: *la serializzazione*.

Per poter inviare un evento tramite il protocollo TCP, è necessario che questo sia serializzabile così come lo devono essere tutti i suoi campi, le interfacce e, ricorsivamente, tutte le entità software collegate ad esse. Questo porterebbe, anche nel caso in cui tutto fosse serializzabile, ad una assoluta inefficienza.

Ovviare a questo limite impone un approccio che consiste nel simulare il comportamento dell'implementazione corrente definendo un'entità che rappresenti l'evento serializzabile, che incapsuli solo le informazioni che consentano all'`ACCProxyNodeSide` di ricostruire l'operazione e l'evento così come definiti in TuCSoN. Queste considerazioni valgono, in maniera del tutto analoga, anche per i messaggi di risposta inviati dall'`ACCProxyNodeSide` verso l'`ACCProxyAgentSide`.

Nuove entità

Allo scopo di realizzare quanto descritto, sono state definite due nuove entità rappresentate dalle classi `InputEventMsg` e `OutputEventMsg`. Queste ultime rappresentano, rispettivamente, un evento generato per una richiesta effettuata da un agente e per una risposta da parte della macchina virtuale. Entrambe queste entità rappresentano una forma serializzabile di evento ed incapsulano le informazioni, incluse quella sullo spazio, necessarie per la ricostruzione dell'evento vero e proprio.

Di seguito viene riportata la loro struttura:

Listing 4.2: InputEventMsg.java

```
package alice.tucson.service;
```

```
public class InputEventMsg implements Serializable {

   private String source;
   private long opId;
   private int opType;
   private LogicTuple tuple;
   private String target;
   private String reactingTC;
   private final long time;
   private final Position place;

   public InputEventMsg(String source, long opId, int opType,
      LogicTuple tuple, String target, long time,
      Position place) {

      this.source = source;
      this.opId = opId;
      this.opType = opType;
      this.tuple = tuple;
      this.target = target;
      this.reactingTC = target;
      this.time = time;
      this.place = place;

   }

   [...] //getters and setters

}
```

Listing 4.3: OutputEventMsg.java

```
package alice.tucson.service;

public class OutputEventMsg implements Serializable {

   private long opId;
   private int opType;
   private boolean allowed;
   private boolean success;
   private LogicTuple reqTuple;
   private Object resTuple;
   private boolean resultSuccess;

   public OutputEventMsg(long i, int t, boolean a, boolean s,
      boolean ok, LogicTuple req, Object res) {

      this.opId = i;
      this.opType = t;
      this.success = s;
      this.allowed = a;
      this.reqTuple = req;
      this.resTuple = res;
      this.resultSuccess = ok;
```

```
    }

    [...] //getters and setters

}
```

Infine è risultato necessario agire anche sulle classi `TucsonMsgRequest` e `TucsonMsgReply` che rappresentano, rispettivamente, i messaggi di richiesta e di risposta scambiati tra i due `ACC`. Tali messaggi sono ora modellati per incapsulare solo l'entità serializzabile rappresentate l'evento, all'interno del quale vengono incapsulate tutte le altre informazioni.

Nuova interazione

Nel complesso, il flusso delle richieste e delle risposte relativamente all'interazione tra i due `ACC` non risulta variato.

Un agente che richiede una determinata operazione al centro di tuple, recupera il contesto, ovvero l'`ACCProxyAgentSide` a lui associato, dopodiché invoca su di esso la primitiva desiderata che, a sua volta, genera un messaggio di richiesta.

L'`ACCProxyAgentSide` effettua l'elaborazione dell'operazione richiesta richiamando il metodo `doOperation` nel quale avviene, oltre alla creazione della sessione di comunicazione, la generazione di un nuovo `InputEventMsg` il quale, a sua volta viene utilizzato per la creazione del messaggio di richiesta relativo all'operazione invocata dall'agente.

Listing 4.4: ACCProxyAgentSide.java

```
[...]

final InputEventMsg ev =
    new InputEventMsg(this.aid.toString(), op.getId(),
    op.getType(), op.getLogicTupleArgument(), tcid.toString(),
    System.currentTimeMillis(), this.getPosition());

final TucsonMsgRequest msg = new TucsonMsgRequest(ev);

[...]
```

Tale messaggio viene inviato tramite protocollo TCP all'`ACCProxyNodeSide` il quale si occupa di recuperare l'`InputEventMsg` da esso e di creare l'operazione corrispondente sfruttando la classe `RespectOperation`, che incapsula tutte le possibili operazioni che un agente può richiedere al centro di tuple.

Listing 4.5: ACCProxyNodeSide.java

```
[...]

final InputEventMsg evMsg = msg.getInputEventMsg();

//Operation Make
RespectOperation opRequested =
    this.makeOperation(null, evMsg.getOpType(),
                               evMsg.getTuple());

[...]
```

A questo scopo è stato definito il metodo `makeOperation` utilizzato per accedere alla classe `RespectOperation` e richiamare su di essa il metodo di costruzione relativo all'operazione richiesta dall'agente, specificando solo il motore Prolog da utilizzare, il tipo di operazione richiesta e la tupla (o il template) specificato.

Listing 4.6: ACCProxyNodeSide.java

```
private RespectOperation makeOperation(Prolog p, int opType,
   LogicTuple tuple) {

   RespectOperation op = null;
   try {
      if ((opType == TucsonOperation.getCode())
         || (opType == TucsonOperation.getSCode())
         || (opType == TucsonOperation.setCode())
         || (opType == TucsonOperation.setSCode()) ) {

         //blocking operation, no need for
         //completion listener
         op = RespectOperation
               .make(null, opType, tuple, null);

      } else {
         //non blocking operation, need for
         //completion listener
         op = RespectOperation
               .make(null, opType, tuple, this);
      }
   } catch (InvalidLogicTupleException e) {
      e.printStackTrace();
   }
   return op;
}
```

Giunti a questo punto, l'`ACCProxyNodeSide` si occupa di creare un nuovo `InputEvent` recuperando le informazioni necessarie dal messaggio ricevuto ed in particolare dall'entità `InputEventMsg`.

Listing 4.7: ACCProxyNodeSide.java

```
[...]

//InputEvent Creation
InputEvent ev = null;
if (this.tcId != null) {
    ev = new InputEvent(this.tcId, opRequested, tid,
                evMsg.getTime(), evMsg.getPlace());
} else {
    ev = new InputEvent(this.agentId, opRequested, tid,
                evMsg.getTime(), evMsg.getPlace());
}

[...]
```

In seguito a quanto descritto, il flusso delle operazioni resta il medesimo mostrato nella Tabella 4.1 con alcune differenze riguardanti la costruzione dell'operazione, che non viene più effettuata dall'entità `RespectTC` e la costruzione dell'evento di ingresso, che non viene più effettuata nel metodo `doOperation` di `RespectVMContext`. Inoltre, ricostruendo l'evento immediatamente alla ricezione della richiesta, risulta sufficiente invocare i metodi di elaborazione delle operazioni, della classe `TupleCentreContainer`, specificando come parametro il solo `InputEvent` appena generato.

Le modifiche discusse in questa sezione ed applicate all'implementazione, permettono di definire l'interazione all'interno dell'infrastruttura TuCSoN come *event-driven*, in cui, per limitazioni tecnologiche e di efficienza, l'unico flusso non strettamente ad eventi appare nella comunicazione tramite protocollo TCP, tra `ACCProxyAgentSide` e `ACCProxyNodeSide`.

4.6 Predicati spaziali

Ora che sono stati definiti gli elementi di base per la gestione dello spazio (Sezione 4.3.1) e che risulta possibile, grazie all'interazione event-driven, accedere alla posizione in cui è stato generato l'evento, si può procedere discutendo l'implementazione dell'estensione per la *space-awareness* e quindi di tutti gli elementi che permettono di dialogare a proposito di spazio definiti dal modello *st*ReSPecT, trattato nel capitolo precedente.

4.6.1 Introduzione

Prima di procedere con la spiegazione dei predicati spaziali implementati, si ricordi che questi necessitano in generale di due informazioni basilari (Sottosezione 3.4.4):

- `S`: parametro associato alla tipologia di spazio relativo alla posizione specificata da `P`. Questo parametro può assumere le forme `ph`, `map`, `org`, `ip` e `dns`.
- `P`: parametro associato alla posizione specificata in accordo con `S`.

Ciascun predicato richiederà che tali parametri siano di tipo *fully-specified* oppure di tipo *input/output*. Nel primo caso, gli argomenti devono essere *ground* ovvero non possono contenere variabili *fresh* nel momento in cui la primitiva viene invocata. Nel secondo caso invece gli argomenti possono anche essere variabili *fresh* o contenerne alcune, in tal caso verrà effettuata l'unificazione dal motore Prolog sottostante l'infrastruttura di coordinazione.

La classe di riferimento per l'implementazione dei predicati spaziali è la `Respect2PLibrary` ovvero la libreria sulla quale si appoggia la `RespectVM` per gestire i predicati utilizzabili nelle guardie e nel body delle reazioni. Per questo motivo predicati di osservazione e guardie con i quali si intende estendere l'infrastruttura devono essere definiti in questa libreria. Per quanto riguarda gli eventi spaziali invece, essi non possono essere richiamati all'interno del body di una reazione ma rappresentano eventi generati internamente alla macchia virtuale e per questo la loro implementazione necessita di una trattazione differente.

4.6.2 Predicati di osservazione

Per l'implementazione dei predicati di osservazione si è agito direttamente sulla classe `Respect2PLibrary`, definendo tre nuovi metodi: `current_place_2`, `event_place_2` e `start_place_2`.

current_place

Il predicato di osservazione `current_place(@S,?P)` ha successo se la posizione P, specificata in accordo con S, unifica con la posizione del nodo nel quale il centro di tuple è in esecuzione.

Si può notare che l'argomento P viene definito come un parametro di *input/output* quindi, oltre alla possibilità di sapere se l'agente si trova nella stessa posizione del centro di tuple, tale predicato permette di recuperare tale posizione semplicemente specificando una variabile come secondo argomento.

Nel caso in cui venga specificata una variabile *fresh* la macchina virtuale si occupa semplicemente dell'unificazione di essa con la posizione del nodo. In caso contrario, e se il primo argomento indica una posizione fisica di tipo `ph`, viene invocato il predicato di guardia `near`: questo perché a causa della alta variabilità delle coordinate GPS, espresse in latitudine e longitudine, risulta opportuno rilassare il vincolo controllando se la posizione specificata si trova entro una certa distanza da quella del nodo. Questo parametro è rappresentato da una variabile definita nella classe `AbstractTupleCentreVMContext` ed inizializzata all'avvio della `RespectVM` ad un valore pari a 10, ovvero dieci metri.

Se il primo argomento indica una posizione di tipo differente da `ph` si è scelto di effettuare l'unificazione diretta, analogamente al caso di variabile *fresh*.

Listing 4.8: Respect2PLibrary.java

```
[...]

public boolean current_place_2(final Term space,
   final Term position) {

   final Position vmPosition = this.vm.getPosition();
   final Term vmPosTerm =
               vmPosition.getPlace(space).toTerm();
   if(!(position instanceof Var)
      && this.unify(space, Term.createTerm(Position.PH))){
         return near_3(space, position,
            this.vm.getDistanceTollerance());
   } else {
      return this.unify(position, vmPosTerm);
   }
}

[...]
```

Come esempio di test, è stata considerata la seguente specifica ReSpecT:

Listing 4.9: currentPlace_spec.rsp

```
reaction(
    out(testCurrentPlace(S,P)),
    (completion, operation),
    (
        current_place(S,P),
        out(currentPlace(P))
    )
).
```

Nel momento in cui l'agente inserisce una tupla *testCurrentPlace(S,P)* all'interno del centro di tuple, viene innescata la reazione mostrata sopra e nel cui body si effettua l'invocazione della primitiva `current_place` e, se questa ha successo, viene inserita una nuova tupla *currentPlace(P)*, nella quale l'argomento P risulta unificato con la posizione recuperata dal predicato di osservazione.

Nel caso in cui l'agente specifichi, come secondo parametro, una variabile, la reazione viene sicuramente valutata con successo e P viene unificato con la posizione del centro di tuple. Se invece l'agente specifica una determinata posizione, ad esempio la sua, tale reazione viene valutata con successo solo se la posizione specificata si trova nelle vicinanze (al massimo a dieci metri) del nodo sul quale il centro di tuple è in esecuzione.

Per esempio, supponendo che il nodo si trovi alla posizione fisica assoluta *coords(44.12,12.44)*, se l'agente inserisce una tupla *testCurrentPlace(ph,P)*, la reazione è innescata e viene invocato il predicato `current_place(ph,P)`. Poiché come secondo argomento è presente una variabile, l'esecuzione di questo predicato porta all'unificazione di P con la posizione del nodo TuCSoN, perciò viene inserita una nuova tupla *currentPlace(coords(44.12,12.44))* all'interno del centro di tuple.

event_place

Il predicato di osservazione `event_place(@S,?P)` ha successo se la posizione P, specificata in accordo con S, unifica con la posizione nella quale è stato originato l'evento.

Questo predicato si riferisce alla *causa diretta* dell'evento, dunque se l'evento supera con successo la guardia *operation*, ciò che serve è la posizione dell'agente, in caso contrario (e.g. è un evento link_in), serve la posizione del nodo sul quale il centro di tuple sorgente del linking è in esecuzione.

Anche in questo caso si può notare che l'argomento P viene definito come un parametro di *input/output* quindi, oltre alla possibilità di sapere se l'evento corrente è stato generato nella posizione specificata in ingresso, tale predicato permette di recuperare tale posizione semplicemente specificando una variabile *fresh* come secondo argomento.

Quando si effettua l'invocazione di questo predicato, viene immediatamente recuperato l'evento corrente. Se tale evento è di tipo *link_in* viene recuperata la posizione del nodo sul quale il centro di tuple sorgente del linking è in esecuzione, altrimenti viene recuperata la posizione nella quale esso è stato generato, dopodiché si procede all'unificazione di questa con quella specificata in ingresso.

Listing 4.10: Respect2PLibrary.java

```
[...]

public boolean event_place_2(final Term space,
   final Term position) {
   final InputEvent ev = this.vm.getCurrentEvent();
   Term ePlaceTerm = null;

   if(ev.isLinking()) {
      ePlaceTerm =
            this.vm.getPosition().getPlace(space).toTerm();
   } else {
      ePlaceTerm =
            ev.getPosition().getPlace(space).toTerm();
   }

   return this.unify(position, ePlaceTerm);
}

[...]
```

Come esempio di test, è stata considerata la seguente specifica ReSpecT:

Listing 4.11: eventPlace_spec.rsp

```
reaction(
   out(testEventPlace(S,P)),
   (completion, operation),
   (
      event_place(S,P),
      out(eventPlace(P))
   )
).
```

Nel momento in cui l'agente inserisce una tupla *testEventPlace(S,P)* all'interno del centro di tuple, viene innescata la reazione mostrata sopra e nel cui body si effettua l'invocazione della primitiva `event_place` e, se questa ha successo, viene inserita una nuova tupla *eventPlace(P)*, nella quale l'argomento P risulta unificato con la posizione recuperata dal predicato di osservazione.

Nel caso in cui l'agente specifichi, come secondo parametro, una variabile, la reazione viene sicuramente valutata con successo e P viene unificato con la posizione nella quale è stato generato l'evento. Se invece l'agente specifica una determinata posizione, tale reazione viene valutata con successo solo se la posizione specificata unifica con quella nella quale l'evento è stato generato.

Per esempio, supponendo che l'agente si trovi alla posizione fisica assoluta *coords(44.10,12.41)*, se esso inserisce una tupla *testEventPlace(ph,P)*, la reazione è innescata e viene invocato il predicato `event_place(ph,P)`. Poiché come secondo argomento è presente una variabile, l'esecuzione di questo predicato porta all'unificazione di P con la posizione nella quale è stato generato l'evento ovvero quella in cui si trova l'agente. Viene dunque inserita una nuova tupla *currentPlace(coords(44.10,12.41))* all'interno del centro di tuple.

start_place

Il predicato di osservazione `start_place(@S,?P)` ha successo se la posizione P, specificata in accordo con S, unifica con la posizione nella quale è stata originata la catena di eventi che ha portato all'evento corrente.

Questo predicato si riferisce alla *causa primaria* dell'evento, dunque, in generale, serve la posizione dell'agente che ha richiesto l'operazione.

Anche in questo caso si può notare che l'argomento P viene definito come un parametro di *input/output* quindi, oltre alla possibilità di sapere se la catena di eventi, che ha portato all'evento corrente, è stata generata nella posizione specificata in ingresso, tale predicato permette di recuperare tale posizione semplicemente specificando una variabile *fresh* come secondo argomento.

Ancora una volta, quando si effettua l'invocazione di questo predicato, viene immediatamente recuperato l'evento corrente e, dopo averene discriminato il tipo e recuperato la posizione corrispondente, si procede all'unificazione di questa con quella specificata in ingresso.

Listing 4.12: Respect2PLibrary.java

```
[...]
public boolean start_place_2(final Term space,
    final Term place) {
    final AbstractEvent e = this.vm.getCurrentEvent();
    Term startEvPosTerm = null;

    if (e.isInternal()) {
        final InternalEvent ie = (InternalEvent) e;
        startEvPosTerm =
            ie.getInputEvent().getPosition().getPlace(space)
              .toTerm();
    }

    if (e.isOutput()) {
        final OutputEvent oe = (OutputEvent) e;
        startEvPosTerm =
            oe.getInputEvent().getPosition().getPlace(space)
              .toTerm();
    }

    return this.unify(place, startEvPosTerm);
}
[...]
```

Come esempio di test, è stata considerata la seguente specifica ReSpecT:

Listing 4.13: startPlace_spec.rsp

```
reaction(
    out(testStartPlace(S,P)),
    (completion, operation),
    (
        out(startPlaceTrigger(S,P))
    )
```

```
).

reaction(
    out(startPlaceTrigger(S,P)),
    (internal),
    (
        start_place(S,P),
        out(startPlace(P))
    )
).
```

Nel momento in cui l'agente inserisce una tupla *testStartPlace(S,P)* all'interno del centro di tuple, viene innescata la prima reazione e nel cui body viene effettuato l'inserimento di una seconda tupla *startPlaceTrigger(S,P)*. Questa operazione porta all'innesco della seconda reazione, dando così origine ad una catena di eventi. Nel body della seconda reazione viene, analogamente a quanto visto per gli altri predicati di osservazione, effettuata l'invocazione della primitiva `start_place` e, se questa ha successo, viene inserita una nuova tupla *startPlace(P)*, nella quale l'argomento P risulta unificato con la posizione recuperata dal predicato di osservazione.

Nel caso in cui l'agente specifichi, come secondo parametro, una variabile, la reazione viene sicuramente valutata con successo e P viene unificato con la posizione nella quale è stata originata la catena di eventi che ha portato all'evento corrente. Se invece l'agente specifica una determinata posizione, tale reazione viene valutata con successo solo se la posizione specificata unifica con quella nella quale la catena di eventi è stata originata.

Per esempio, supponendo che l'agente si trovi alla posizione fisica assoluta *coords(44.10,12.41)*, se esso inserisce una tupla *testStartPlace(ph,P)*, la reazione è innescata e viene inserita una tupla *startPlaceTrigger(ph,P)* che porta all'innesco della seconda reazione, nella quale viene invocato il predicato `start_place(ph,P)`. Poiché come secondo argomento è presente una variabile, l'esecuzione di questo predicato porta all'unificazione di P con la posizione nella quale è stata originata la catena di eventi che ha portato all'evento corrente, ovvero quella in cui si trova l'agente. Viene dunque inserita una nuova tupla *startPlace(coords(44.10,12.41))* all'interno del centro di tuple.

Esempio applicativo: *SpatialLog*

Allo scopo di mostrare una semplice applicazione dei predicati di osservazione appena discussi, è stata definita la seguente specifica ReSpecT:

Listing 4.14: spatialLog_spec.rsp

```
reaction(
    in(q(X)),
    (completion, operation),
    (
        current_place(ph,DevPos),
        event_place(ph,AgentPos),
        out(in_log(AgentPos,DevPos,q(X)))
    )
).
```

Eseguendo questo esempio, viene inserita una tupla `in_log/3` contenente informazioni spaziali ogni volta che un agente TuCSoN recupera una tupla della forma `q(_)` dal centro di tuple: questa specifica implementa dunque una sorta di *log spaziale*, registrando le posizioni assolute sia dell'agente che richiede la tupla che del dispositivo sul quale il centro di tuple è in esecuzione.

Per esempio, supponendo che l'agente si trovi nella posizione *coords(44.10,12-.41)* e il nodo nella posizione *coords(44.23,12.45)*, se l'agente recupera una tupla *q(spatialLog)* presente nello spazio di tuple, la reazione è innescata e vengono invocati i predicati `current_place(ph,DevPos)` e `start_place(ph,AgentPos)`. Tramite essi vengono recuperate, rispettivamente, le posizioni fisiche assolute del dispositivo che ospita il nodo e del luogo nel quale è stato generato l'evento e quindi la posizione dell'agente. Poiché in entrambe le invocazioni il secondo argomento risulta essere una variabile, l'esecuzione di questi predicati porta all'unificazione di *DevPos* con *coords(44.23,12.45)* e di `AgentPos` con *coords(44.10,12.41)*. Viene dunque inserita una nuova tupla *in_log(coords(44.10, 12.41), coords(44.23,12.45), q(spatialLog))* all'interno del centro di tuple.

4.6.3 Guardie

Per l'implementazione dei predicati di guardia si è agito, ancora una volta, sulla classe `Respect2PLibrary`, definendo due nuovi metodi: `near_3` e `at_2`.

near

Il predicato di guardia `near(@S,@P,@R)` ha successo quando il centro di tuple è in esecuzione in una posizione contenuta nella regione spaziale avente centro P e raggio R, specificata in accordo con S.

In questo caso gli argomenti P ed R (analogamente ad S) vengono definiti come parametri *fully-specified* quindi, a differenza dei predicati di osservazione, non è possibile specificare variabili *fresh* che dovranno essere unificate, dunque l'agente è obbligato a specificare sia la posizione rappresentante il centro che il raggio della regione spaziale.

Le diverse tipologie di posizione devono essere trattate separatamente, in quanto ciascuna di esse viene definita in una "forma" specifica e quindi non è possibile generalizzarne la trattazione. Infatti, mentre la posizione di tipo `ph` è rappresentata da un termine nella forma `coords(Lat,Lng)`, tutte le altre posizioni sono definite come stringhe aventi una loro formattazione specifica.

Quando è richiesta la valutazione di questa guardia, viene immediatamente recuperata la posizione della `RespectVM` relativa al tipologia specificata in ingresso dal termine *space*, dopodiché si procede con la discriminazione della tipologia di valutazione richiesta, trattando le informazioni nella maniera più opportuna.

Listing 4.15: Respect2PLibrary.java

```
[...]
public boolean near_3(final Term space, final Term center,
    final Term radius) {
    final Position vmPosition = this.vm.getPosition();
```

```
    final Term vmPosTerm = vmPosition.getPlace(space).toTerm();

    if(this.unify(space,
                  Term.createTerm(Position.PH))){
        [...]
    } else if(this.unify(space,
                         Term.createTerm(Position.MAP))){
        [...]
    } else if(this.unify(space,
                         Term.createTerm(Position.ORG))){
        [...]
    } else if(this.unify(space,
                         Term.createTerm(Position.IP))){
        [...]
    } else if(this.unify(space,
                         Term.createTerm(Position.DNS))){
        [...]
    }
    return false;
}
[...]
```

Se il termine *space* in ingresso è riferito ad un posizione di tipo `ph`, dopo aver controllato che il termine *center* sia stato specificato correttamente, vengono recuperate le coordinate (latitudine e longitudine) della posizione del nodo e del centro della regione spaziale di interesse. Fatto ciò è possibile controllare se la posizione, nel nodo sul quale in centro di tuple è in esecuzione, risulta compresa in un cerchio di centro *center* e di raggio *radius*. In caso affermativo, la valutazione della guardia ha successo.

In questo caso il termine *radius* assume il significato di distanza, espressa in metri, tra le due posizioni in esame. Nello specifico, se la distanza è minore o uguale al raggio specificato allora la guardia ha successo, altrimenti fallisce.

Listing 4.16: Respect2PLibrary.java

```
[...]

    if(this.unify(space, Term.createTerm(Position.PH))){
        final Struct vmPosStruct = (Struct) vmPosTerm;
        final Struct centerStruct = (Struct) center;
        if (! "coords".equals(centerStruct.getName())) {
            return false;
        }
        final float vmX =
            ( (alice.tuprolog.Number) vmPosStruct.getArg(0) )
                .floatValue();
        final float vmY =
            ( (alice.tuprolog.Number) vmPosStruct.getArg(1) )
                .floatValue();
        final float cX =
            ( (alice.tuprolog.Number) centerStruct.getArg(0) )
                .floatValue();
        final float cY =
            ( (alice.tuprolog.Number) centerStruct.getArg(1) )
```

```
            .floatValue();
      final float radiusN =
         GeoUtils.toDegrees(( (alice.tuprolog.Number) radius)
            .floatValue());

      return Math.pow((vmX - cX),2) + Math.pow((vmY - cY),2)
               <= Math.pow(radiusN, 2);
   }

[...]
```

Se il termine *space* in ingresso è riferito ad un posizione di tipo `map` viene sfruttato un servizio di geolocalizzazione creato appositamente per la piattaforma di esecuzione corrente, questo allo scopo di effettuare il *Geocoding* della posizione (fisica geografica) specificata in ingresso. Tale meccanismo consiste nell'effettuare una richiesta al servizio di geolocalizzazione per "tradurre" un indirizzo geografico (e.g. Via Sacchi, 3, Cesena, Italia) in coordinate espresse sotto forma di latitudine e longitudine. Fatto ciò, viene costruito un termine rappresentante la stessa posizione data in ingresso ma espressa in termini fisici assoluti (coordinate) ed è possibile iterare sul predicato di guardia `near(ph,Center,Radius)` (specificandola come centro) in modo tale da effettuare le medesime elaborazioni di cui al caso precedente.

Gli aspetti specifici riguardanti il servizio di geolocalizzazione, che in parte appaiono nello stralcio di codice riportato di seguito, verranno descritti in maniera dettagliata nel prossimo capitolo.

Listing 4.17: Respect2PLibrary.java

```
[...]

   if(this.unify(space, Term.createTerm(Position.MAP))){
      GeolocationServiceManager geolocationManager =
            GeolocationServiceManager.getGeolocationManager();
      if(geolocationManager.getServices().size() > 0) {
         int platform  = PlatformUtils.getPlatform();
         GeolocationService geoService =
            GeolocationServiceManager.getGeolocationManager()
                  .getAppositeService(platform);
         if(geoService != null) {
            Term centerCoords = geoService
             .geocode(Tools.removeApices(center.toString()));
            return
               this.near_3(Term.createTerm("ph"),
                           centerCoords,
                           radius);
         }
      }
   }

[...]
```

Se il termine *space* in ingresso è riferito ad un posizione di tipo `org` vengono effettuate le stesse considerazioni di cui al caso precedente ma solo successivamente ad una minima preparazione delle informazioni.

Infatti, nel caso di posizione fisica organizzativa, è stato deciso per semplicità di creare una sua rappresentazione partendo da quella geografica. In particolare, questo tipo di posizione, assume la forma *"Workspace at (map address)"*, dunque per una corretta elaborazione è sufficiente recuperare la parte *map address* e poi iterare sulla guardia `near(map,Center,Radius)` specificandola come centro.

Listing 4.18: Respect2PLibrary.java

```
[...]

   if(this.unify(space, Term.createTerm(Position.ORG))){
      String orgCenterS =
         Tools.removeApices(center.toString());
      String[] orgCenterParts = orgCenterS.split("at ");
      return
         this.near_3(
            Term.createTerm("map"),
            Term.createTerm("'"+orgCenterParts[1]+"'"),
            radius);
   }

[...]
```

Se il termine *space* in ingresso è riferito ad un posizione di tipo `ip` devono essere effettuate elaborazioni riguardanti gli indirizzi IP espressi nella forma *"IP/netmask"*.

In questo caso il termine *radius* assume il significato di distanza tra sotto-reti IP. Più nello specifico, viene effettuato un controllo sulle *netmask* in modo tale che con un raggio pari a zero, la guardia abbia successo se l'indirizzo di rete specificato in ingresso appartiene alla stessa sotto-rete dell'indirizzo del nodo sul quale è in esecuzione il centro di tuple, oppure, con un raggio pari a uno, la guardia abbia successo se l'indirizzo di rete in ingresso appartiene alla stessa sotto-rete o alle sotto-reti immediatamente adiacenti a quella del nodo, e così via; ad esempio se il raggio è pari a uno e la maschera di rete dell'indirizzo IP del nodo è /24, l'indirizzo specificato in ingresso può appartenere alle reti /23, /24 o /25.

Innanzitutto, viene effettuato il controllo di distanza tra l'indirizzo IP del nodo e quello specificato in ingresso, dopodiché, se tali indirizzi risultano alla giusta distanza, in accordo con il termine *radius*, si procede controllando se appartengono o meno alla stessa rete, valutando la loro compatibilità. In caso affermativo, la guardia viene valutata con successo, altrimenti fallisce.

Listing 4.19: Respect2PLibrary.java

```
[...]

   if(this.unify(space, Term.createTerm(Position.IP))){
      final float radiusN = ( (alice.tuprolog.Number) radius)
         .floatValue();

      String vmIpS = Tools.removeApices(vmPosTerm.toString());
      String vmIp = NetworkUtils.getIp(vmIpS);
      int vmMask = NetworkUtils.getNetmask(vmIpS);
      String vmDecMask =
```

```
            NetworkUtils.getDecimalNetmask(vmIpS);

      String centerIpS =
             Tools.removeApices(center.toString());
      String centerIp = NetworkUtils.getIp(centerIpS);
      int mask = NetworkUtils.getNetmask(centerIpS);
      String decMask =
             NetworkUtils.getDecimalNetmask(centerIpS);

      if (vmMask <= (mask + radiusN) &&
          vmMask >= (mask - radiusN)) {
         return
           NetworkUtils.sameNetwork(vmIp, centerIp, decMask);
      }
      return false;
   }

[...]
```

Per poter effettuare questi controlli, è necessario disporre di una classe di utilità che incapsuli tutte le funzioni utili allo scopo di elaborare e manipolare gli indirizzi di rete. A tal fine è stata definita la classe `NetworkUtils` tramite la quale è possibile avvalersi di funzionalità progettate appositamente per recuperare la parte dell'indirizzo relativa ad IP o *netmask*, convertire la maschera di rete dalla rappresentazione "/XX" a quella decimale (e.g. /24 corrisponde a 255.255.255.0) e controllare se due indirizzi IP appartengono alla stessa rete:

Listing 4.20: NetworkUtils.java

```
[...]

public static boolean sameNetwork(String ip1, String ip2,
   String mask) {
   try {
      byte[] a1 = InetAddress.getByName(ip1).getAddress();
      byte[] a2 = InetAddress.getByName(ip2).getAddress();
      byte[] m = InetAddress.getByName(mask).getAddress();

      for (int i = 0; i < a1.length; i++)
         if ((a1[i] & m[i]) != (a2[i] & m[i]))
            return false;
   } catch (UnknownHostException e) {
      e.printStackTrace();
   }
   return true;
}

[...]
```

Infine, se il termine *space* in ingresso è riferito ad un posizione di tipo `dns` devono essere effettuate elaborazioni riguardanti i nomi di dominio al quale il nodo e l'agente appartengono.

In questo caso il termine *radius* assume il significato di distanza tra i domini. Più nello specifico, viene effettuato un controllo sul nome di dominio in

modo tale che con un raggio pari a zero, la guardia abbia successo se il dominio specificato in ingresso coincide esattamente con quello del nodo sul quale è in esecuzione il centro di tuple, oppure, con un raggio pari a uno, la guardia abbia successo se il dominio in ingresso è contenuto, al massimo a distanza uno, nel dominio a cui il nodo appartiene, o viceversa; ad esempio, se il raggio è pari a uno e il dominio a cui il nodo appartiene è *ingce.unibo.it*, allora il dominio specificato in ingresso può appartenere ad esempio ai domini *ingbo.unibo.it* o *unibo.it*.

Innanzitutto si effettua il controllo sul raggio specificato in ingresso, se questo è pari a zero allora la guardia ha successo solo se il dominio in ingresso coincide esattamente a quello che contiene il nodo. Se invece il raggio è maggiore di zero viene eseguito un controllo incrementale partendo dall'ultima parte del dominio specificato in ingresso, verificando ad ogni ciclo se questo è contenuto o meno nel dominio a cui il nodo appartiene. Se lo è, viene controllata la distanza dal dominio d'ingresso e se questa risulta minore o uguale a quella specificata allora la guardia ha successo, altrimenti viene eseguito un altro ciclo.

Nel caso peggiore, il processo continua fino a che non è stato eseguito il controllo con l'intero nome di dominio specificato in ingresso. Se anche in questo caso non si trova corrispondenza con il dominio del nodo, rispettando anche la condizione di distanza massima, allora la guardia fallisce.

Listing 4.21: Respect2PLibrary.java

```
[...]

  if(this.unify(space, Term.createTerm(Position.DNS))){
     final float radiusN = ( (alice.tuprolog.Number) radius)
        .floatValue();

     String vmDnsS =
            Tools.removeApices(vmPosTerm.toString());
     String centerDnsS =
            Tools.removeApices(center.toString());

     if(radiusN == 0) {
        return vmDnsS.equals(centerDnsS);
     } else {
        String[] vmDnsParts = vmDnsS.split("\\.");
        String[] centerDnsParts = centerDnsS.split("\\.");
        String toCheck = "";
        for (int i = centerDnsParts.length-1; i >= 0 ; i--) {
           for (int j = i; j < centerDnsParts.length; j++) {
              toCheck += centerDnsParts[j] +
                         (j < centerDnsParts.length-1 ?
                            "." : "");
           }
           if(vmDnsS.contains(toCheck)) {
              for (int k = (int) radiusN; k > 0; k--) {
                 if(vmDnsParts[k].equals(centerDnsParts[i])){
                    return true;
                 }
              }
           }
```

```
            toCheck = "";
        }
    }
}

[...]
```

Come esempio di test del predicato di guardia `near`, è stata considerata la seguente specifica ReSpecT:

Listing 4.22: guardNear_spec.rsp

```
reaction(
   out(testNear(S,P,R)),
   (completion, operation, near(S,P,R)),
   (
      out(nearRes(P))
   )
).
```

Nel momento in cui l'agente inserisce una tupla *testNear(S,P,R)* all'interno del centro di tuple, viene innescata la reazione mostrata sopra tra le cui guardie viene valutata anche la guardia `near`. Se questa ha successo, secondo i criteri appena descritti, viene inserita una nuova tupla *nearRes(P)*, nella quale P risulta unificato con il centro della regione spaziale specificato dall'agente.

Per la corretta valutazione del predicato di guardia `near`, come già detto, è necessario che i parametri in ingresso siano *fully-specified*, dunque non è possibile specificare parametri di *input/output* che dovranno essere unificati, infatti, se questo si verifica la guardia fallisce.

Per esempio, supponendo che l'agente si trovi alla posizione fisica assoluta *coords(44.102581, 12.412398)*, se esso inserisce una tupla *testNear(ph, coords(44.102581, 12.412398), 10))*, la reazione è innescata e viene valutata la guardia `near(ph, coords(44.102581, 12.412398), 10))`. Nello specifico, viene valutato se la posizione fisica del nodo si trova in un cerchio di raggio dieci (metri) dalla posizione specificata. Sapendo che dieci metri in gradi longitudinali/latitudinali equivalgono approssimativamente a 8.99281×10^{-5}, e supponendo che il nodo si trovi nella posizione *coords(44.102531, 12.412348)*, la guardia verrà valutata con successo poiché la posizione del nodo risulta interna al cerchio avente come centro la posizione dell'agente e raggio pari a dieci metri. Viene dunque inserita una nuova tupla *nearRes(coords(44.102581, 12.412398))* che riporta la posizione specificata dall'agente come centro della regione spaziale.

at

Il predicato di guardia `at(@S,@P)` ha successo quando il centro di tuple è in esecuzione alla posizione P, specificata in accordo con S.

In questo caso, analogamente al predicato precedente, l'argomento P viene definito come parametro *fully-specified* quindi non è possibile specificare variabili *fresh* che dovranno essere unificate, dunque l'agente è obbligato a specificare la posizione nella quale intende controllare se è presente il nodo sul quale è in esecuzione il centro di tuple.

Analogamente a quanto visto per il predicato di osservazione `current_place`, a causa della alta variabilità delle coordinate GPS, espresse in latitudine e longitudine, risulta opportuno rilassare il vincolo controllando se la posizione specificata si trova entro una certa distanza dal nodo.

Per questo motivo, nell'implementazione di questa guardia si è potuto sfruttare il predicato `near` specificando, come raggio, il parametro rappresentate la distanza massima (già utilizzato nell'implementazione di `current_place`).

Listing 4.23: Respect2PLibrary.java

```
[...]

public boolean at_2(final Term space, final Term position) {
    return
        near_3(space, position,
                this.vm.getDistanceTollerance());
}

[...]
```

Come esempio di test del predicato di guardia `at`, è stata considerata la seguente specifica ReSpecT:

Listing 4.24: guardAt_spec.rsp

```
reaction(
    out(testAt(S,P)),
    (completion, operation, at(S,P)),
    (
        out(atRes(P))
    )
).
```

Nel momento in cui l'agente inserisce una tupla *testAt(S,P)* all'interno del centro di tuple, viene innescata la reazione mostrata sopra tre le cui guardie viene valutata anche la guardia `at`. Se questa ha successo, viene inserita una nuova tupla *atRes(P)*, nella quale P risulta unificato con il centro della regione spaziale specificato dall'agente.

Analogamente a `near`, per la corretta valutazione del predicato di guardia `at`, è necessario che i parametri in ingresso siano *fully-specified*, dunque non è possibile specificare parametri di *input/output* che dovranno essere unificati, infatti, se questo si verifica la guardia fallisce.

Per esempio, supponendo che l'agente si trovi alla posizione fisica assoluta *coords(44.102581, 12.412398)*, se esso inserisce una tupla *testAt(ph, coords(44.102581, 12.412398), 10))*, la reazione è innescata e viene valutata la guardia `at(ph, coords(44.102581, 12.412398), 10))`. Supponendo che il nodo si trovi nella posizione *coords(44.102531, 12.412348)*, la valutazione terminerà con successo poiché internamente viene richiamata la guardia `near` e la posizione del nodo risulta interna ad un cerchio di centro *coords(44.102581, 12.412398)* e raggio dieci (metri). Viene dunque inserita una nuova tupla *atRes(coords(44.102581, 12.412398))* che riporta la posizione specificata dall'agente come centro della regione spaziale.

4.6.4 Eventi

Per gestire gli eventi spaziali `from(?S,?P)` e `to(?S,?P)`, è stato necessario estendere il comportamento della `RespectVM`, modificando la macchina a stati. Inoltre, per la corretta elaborazione di questi eventi, è stata arricchita anche la classe `RespectVMContext` con opportune funzioni per il recupero delle reazioni spaziali (presenti nella specifica di comportamento) e la loro valutazione.

La gestione di questi eventi è unica per ciascuno di essi senza particolari differenze, se non per aspetti che riguardano la differente "signature" per la quale presentano ovviamente diversi funtori.

Innanzitutto, si ricordi che:

- `from(?S,?P)`: corrisponde ad un evento spaziale generato quando il dispositivo che ospita il centro di tuple comincia a muoversi dalla posizione *P*, specificata in accordo con *S*.
- `to(?S,?P)`: corrisponde ad un evento spaziale generato quando il dispositivo che ospita il centro di tuple termina il movimento e raggiunge la posizione *P*, specificata in accordo con *S*.

Per prima cosa si deve considerare che questi due eventi devono essere generati solo ed esclusivamente se, nella specifica di comportamento del centro di tuple, sono presenti reazioni che li riguardano, in modo tale da evitare che vengano generati anche quando l'agente o il sistema non presentino un comportamento che specifichi di reagire a tali eventi.

A tale scopo si è agito sulla classe `RespectVMContext` e, nello specifico, sui metodi `addReactionSpecHelper` e `setReactionSpecHelper`. Quest'ultimo, in particolare, viene eseguito quando un agente TuCSoN invoca la primitiva `setS` per iniettare uno specifico comportamento all'interno del centro di tuple.

In questo caso viene effettuato un controllo, sulla specifica che si vuole iniettare nel sistema, alla ricerca di reazioni che abbiano come primo argomento l'evento `from` o `to`. A questo proposito si sfruttano due appositi metodi definiti all'interno della classe `RespectVMContext` (*findFromReactions* e *findToReactions*) nei quali viene costruito un termine che rappresenta il template della reazione da ricercare nella specifica, dopodiché tramite il motore Prolog si effettua l'unificazione e, se questa ha successo, la reazione che unifica con il template specificato viene recuperata ed inserita nella lista delle reazioni spaziali trovate. Al termine del processo questi metodi restituiscono la lista delle reazioni riferite all'evento `from` e `to`, rispettivamente.

Listing 4.25: RespectVMContext.java

```
[...]

public Iterator<Term> findFromReactions() {
   final List<Term> foundReactions = new ArrayList<Term>();
   try {
      final Struct from =
               new Struct("from",
                          new alice.tuprolog.Var("S"),
                          new alice.tuprolog.Var("P"));
```

```
        final Struct fev =
                  new Struct("reaction",
                             from,
                             new alice.tuprolog.Var("G"),
                             new alice.tuprolog.Var("R"));

        SolveInfo info = this.trigCore.solve(fev);
        while (info.isSuccess()) {
            foundReactions.add(from);
            if (this.trigCore.hasOpenAlternatives()) {
                info = this.trigCore.solveNext();
            } else {
                break;
            }
            this.trigCore.solveEnd();
        }
    } catch (final NoMoreSolutionException e) {
        this.notifyException(
              "INTERNAL ERROR: fetchFromReactions");
        this.trigCore.solveEnd();
    }
    return foundReactions.iterator();
}

public Iterator<Term> findToReactions() {

    [...similar to findFromReactions...]

}

[...]
```

Una volta recuperate le liste delle reazioni spaziali, se almeno una di esse risulta non vuota, il servizio di geolocalizzazione dovrà essere informato a proposito del fatto che il sistema deve ricevere gli aggiornamenti sul movimento del dispositivo: è dunque necessario che il servizio stesso si occupi della generazione degli eventi spaziali per poi notificarli alla `RespectVM`.

La notifica degli eventi spaziali avviene semplicemente aggiungendo l'evento appena generato al multi-set *SitE* rappresentate l'insieme degli eventi in ingresso relativi ad eventi temporali, spaziali e ambientali.

Listing 4.26: RespectVMContext.java

```
[...]

public void notifyInputEnvEvent(final InputEvent in) {
    this.addEnvInputEvent(in);
    this.vm.notifyNewInputEvent();
}

[...]
```

Una volta aggiunto l'evento al multi-set *SitE*, la `RespectVM`, che nel caso più semplice si trova nello stato di *Idle*, si occupa di controllare la presenza

di eventi ambientali tramite il metodo `pendingEnvEvents()` definito all'interno della classe `AbstractTupleCentreVMContext`.

Listing 4.27: AbstractTupleCentreVMContext.java

```
[...]

public boolean pendingEnvEvents() {
   synchronized (this.inputEnvEvents) {
      return this.inputEnvEvents.size() > 0;
   }
}

[...]
```

Listing 4.28: IdleState.java

```
[...]

@Override
public AbstractTupleCentreVMState getNextState() {
   if (this.vm.pendingEvents()) {
      return this.listeningState;
   } else if (this.vm.pendingEnvEvents()) {
      return this.fetchEnvState;
   }
   return this;
}

[...]
```

Nel caso in cui vi siano eventi spaziali, la macchina virtuale transita allo stato *FetchEnvState*, all'interno del quale viene recuperato il primo evento presente nel multi-set *SitE* e, tramite il metodo `fetchTriggeredReactions` (definito in `RespectVMContext`), si recuperano le reazioni (anche spaziali) innescate dall'evento e le si aggiunge al multi-set *Re* rappresentate le reazioni innescate in attesa di essere valutate.

Listing 4.29: FetchEnvState.java

```
[...]

@Override
public void execute() {
   this.vm.fetchPendingEnvEvent();
   final InputEvent ev = this.vm.getCurrentEvent();
   this.vm.fetchTriggeredReactions(ev);
}

[...]
```

Infine, se sono presenti reazioni innescate, la `RespectVM` transita allo stato *ReactingState*, all'interno del quale vengono recuperate e valutate, una per volta, le reazioni presenti nel multi-set *Re*, comprese eventuali reazioni spaziali innescate dagli eventi `from` e `to`.

Listing 4.30: ReactingState.java

```
[...]

@Override
public void execute() {
    TriggeredReaction tr = this.vm.removeTriggeredReaction();
    if (tr != null) {
        this.vm.evalReaction(tr);
    } else if (this.vm.timeTriggeredReaction()) {
        tr = this.vm.removeTimeTriggeredReaction();
        if (tr != null) {
            this.vm.evalReaction(tr);
            this.vm.updateSpecAfterTimedReaction(tr);
        }
    }
}

[...]
```

Un esempio applicativo riguardante la generazione e la gestione degli eventi spaziali viene riportato al termine del prossimo capitolo in quanto prima risulta necessario descrivere l'implementazione del servizio di geolocalizzazione e come essa si interfaccia con la `RespectVM`. Quindi nel prossimo capitolo verranno mostrati gli aspetti implementativi specifici dell'architettura software utilizzata per agganciare la macchina virtuale ad una piattaforma di geolocalizzazione generica che dovrà poi essere implementata in maniera specifica dallo sviluppatore.

Capitolo 5

Geolocalizzazione

Ora che sono stati definiti un modello architetturale che identifica i componenti, e ne definisce le possibili interazioni a livello di coordinazione interna, ed un linguaggio di coordinazione progettato per poter gestire un sistema mobile e quindi per far si che i vari componenti possano dialogare a proposito di spazio, è necessario definire un'architettura che espleti connessioni e interazioni tra il sistema TuCSoN e la piattaforma di geolocalizzazione. A questo scopo, per poter garantire l'interfacciamento con qualsiasi piattaforma di geolocalizzazione, il middleware di coordinazione è stato esteso con un nuovo livello logico chiamato *Geolocation* che incapsulerà le entità generiche per creazione e controllo del servizio di geolocalizzazione, di qualunque tipo esso sia, e ricezione e notifica degli eventi relativi alla posizione.

5.1 Livello Geolocation

Il livello *Geolocation* (Figura 5.1) è stato progettato per poter astrarre dalle tecnologie specifiche tramite la creazione di entità generiche che fungeranno da ponte tra il servizio di geolocalizzazione e il centro di tuple. Questo livello logico è composto essenzialmente da tre entità:

- `GeolocationService`: entità astratta che rappresenta un servizio di geolocalizzazione generico.
- `GeolocationServiceListener`: entità ascoltatrici che restano in attesa della ricezione di notifiche riguardanti variazione di posizione e generazione degli eventi di movimento provenienti dal servizio associato.
- `GeolocationServiceManager`: entità responsabile della creazione, registrazione e rimozione dei servizi di geolocalizzazione.

Per garantire la corretta interazione tra il servizio di geolocalizzazione e il middleware di coordinazione, viene utilizzato il pattern architetturale *EventListener* che prevede la presenza di almeno un ascoltatore in attesa delle notifiche provenienti da altre entità, in questo caso dal servizio di geolocalizzazione.

In questa sezione vengono analizzati i tre componenti appena enunciati, mostrandone le caratteristiche principali e le modalità di interazione.

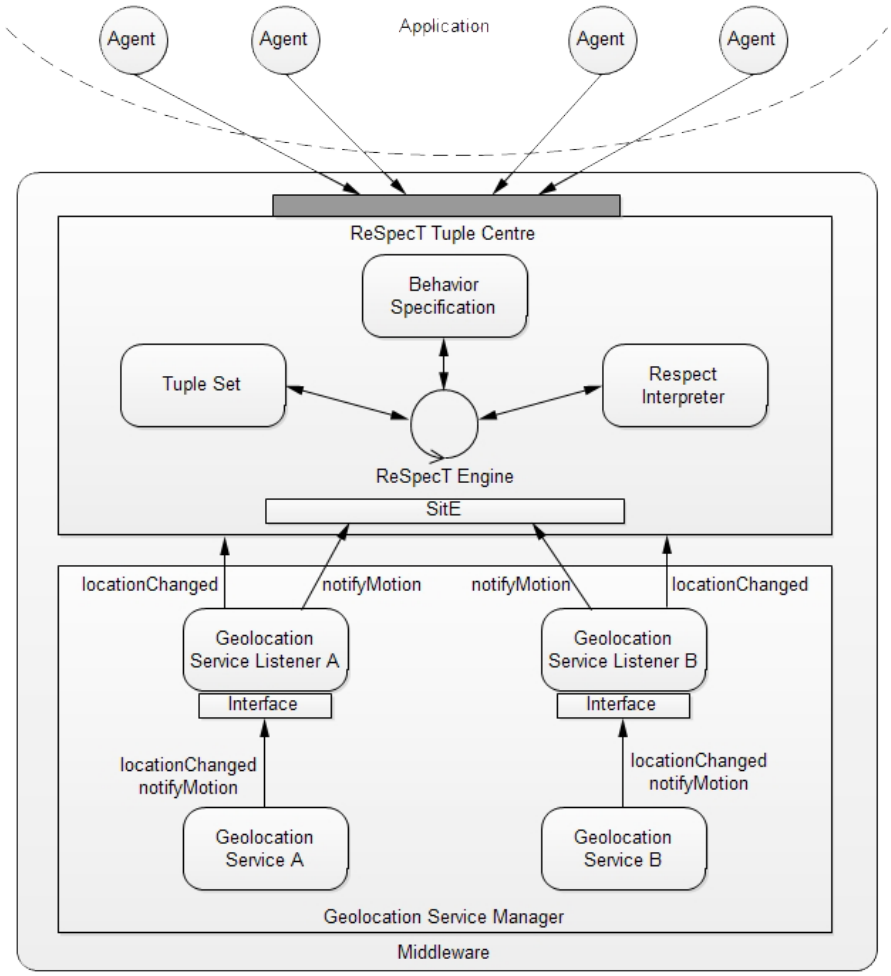


Figura 5.1: Livello *Geolocation*

5.1.1 GeolocationService

L'entità astratta *GeolocationService* rappresenta il generico servizio di geolocalizzazione, al quale vengono associati un identificatore (`GeoServiceId`), una piattaforma di esecuzione rilevata tramite la classe di utilità `PlatformUtils` e un centro di tuple a cui riferirsi per gli aggiornamenti su posizione e movimento. Questo componente, inoltre, incapsula anche una lista di ascoltatori alla quale vengono aggiunge le entità `GeolocationServiceListener` che devono ricevere notifiche relative alla posizione.

La struttura generica di un `GeolocationService` è definita da un'apposita interfaccia denominata `IGeolocationService`, la quale dichiara tutti i metodi che un qualsiasi servizio di geolocalizzazione deve implementare per garantire la corretta interazione con gli ascoltatori e quindi con il centro di tuple. Tale interfaccia presenta la seguente struttura:

`void notifyLocationChanged(double lat, double lng)` – utilizzato per notificare agli ascoltatori collegati che la posizione fisica assoluta del dispostivo che ospita il centro di tuple è cambiata.

`void notifyLocationChanged(IPlace place)` – analogo al precedente ma, tramite l'argomento di tipo `IPlace`, è possibile notificare il cambiamento di una posizione di qualsiasi tipo.

`void notifyStartMovement(double lat, double lng)` – utilizzato per notificare agli ascoltatori che è cominciato il movimento da una posizione fisica assoluta.

`void notifyStartMovement(String space, IPlace place)` – analogo al precedente ma, tramite l'argomento di tipo `IPlace`, è possibile notificare che è cominciato il movimento da una posizione di qualsiasi tipo.

`void notifyStopMovement(double lat, double lng)` – utilizzato per notificare agli ascoltatori che è terminato il movimento in una posizione fisica assoluta.

`void notifyStopMovement(String space, IPlace place)` – analogo al precedente ma, tramite l'argomento di tipo `IPlace`, è possibile notificare la terminazione del movimento in una posizione di qualsiasi tipo.

`int getPlatform()` – utilizzato per recuperare la piattaforma di esecuzione sulla quale il servizio di geolocalizzazione è in esecuzione.

`GeoServiceId getServiceId()` – utilizzato per recuperare l'identificatore del servizio.

`TucsonTupleCentreId getTcId()` – utilizzato per recuperare l'identificatore del centro di tuple sul quale è attivo il servizio.

`void addListener(IGeolocationServiceListener l)` – utilizzato per collegare un nuovo ascoltatore al servizio geolocalizzazione.

`void stop()` – utilizzato per interrompere il servizio e quindi gli aggiornamenti su posizione e movimento.

`void start()` – utilizzato per avviare il servizio e quindi ricevere gli aggiornamenti su posizione e movimento.

`boolean isRunning()` – utilizzato per controllare se il servizio è in esecuzione o meno.

`void generateSpatialEvents(boolean generate)` – utilizzato per impostare la variabile di controllo relativa alla generazione degli eventi spaziali.

`Term geocode(String address)` – utilizzato per richiedere al servizio il *Geocoding* di un indirizzo geografico dato in ingresso.

Relativamente al metodo `generateSpatialEvents(boolean generate)`, esso viene richiamato durante la fase di controllo della presenza di reazioni spaziali, mostrata nella Sottosezione 4.6.4, allo scopo di impostare il valore di una variabile di controllo, denominata *genSpatialEvents*, definita all'interno della classe

`GeolocationService`. In particolare, se risultano presenti reazioni spaziali nella specifica di comportamento, tale variabile assume valore `true`, altrimenti `false`. In questo modo è possibile, per il servizio di geolocalizzazione, sapere se devono o meno essere generati gli eventi spaziali `from` e `to`.

Come già accennato, `GeolocationService` rappresenta un'entità astratta, che deve essere concretizzata da un'implementazione specifica del servizio di geolocalizzazione, la quale dovrà fornire il supporto necessario per rendere possibile l'interazione tra la tecnologia propria del dispositivo utilizzato e il middleware di coordinazione. Tale implementazione corrisponde alle API specifiche per la tecnologia del dispositivo e sarà incaricata di interfacciarsi con esse, recuperare le informazioni di posizione richieste e notificarle agli ascoltatori collegati sfruttando i metodi già definiti nell'implementazione astratta `GeolocationService`.

Relativamente a quanto detto, i metodi definiti in quest'ultima, relativi agli aggiornamenti di posizione e movimento, si occupano semplicemente di scorrere la lista degli ascoltatori collegati ed invocare il metodo opportuno, dipendentemente da ciò che si deve notificare.

Listing 5.1: GeolocationService.java

```
[...]

@Override
public void notifyLocationChanged(double lat, double lng) {
   for (IGeolocationServiceListener l : listeners) {
      l.locationChanged(
            new PhPlace("coords(" + lat + "," + lng + ")"));
   }
}

@Override
public void notifyLocationChanged(IPlace place) {
   for (IGeolocationServiceListener l : listeners) {
      l.locationChanged(place);
   }
}

@Override
public void notifyStartMovement(double lat, double lng) {
   IPlace place =
         new PhPlace("coords(" + lat + "," + lng + ")");
   for (IGeolocationServiceListener l : listeners) {
      l.moving(RespectOperation.OPTYPE_FROM, Position.PH,
               place);
   }
}

@Override
public void notifyStopMovement(double lat, double lng) {
   IPlace place =
         new PhPlace("coords(" + lat + "," + lng + ")");
   for (IGeolocationServiceListener l : listeners) {
```

```
            l.moving(RespectOperation.OPTYPE_TO, Position.PH,
                    place);
        }
    }

    @Override
    public void notifyStartMovement(String space, IPlace place) {
        for (IGeolocationServiceListener l : listeners) {
            l.moving(RespectOperation.OPTYPE_FROM, space, place);
        }
    }

    @Override
    public void notifyStopMovement(String space, IPlace place) {
        for (IGeolocationServiceListener l : listeners) {
            l.moving(RespectOperation.OPTYPE_TO, space, place);
        }
    }

    [...]
```

Oltre a questi, nella classe `GeolocationService` è presente l'implementazione di tutti i metodi dichiarati nell'interfaccia `IGeolocationService` e, ad eccezione di `start()` e `stop()`, non è necessario ridefinirli all'interno dell'implementazione specifica del servizio. Per quanto riguarda i due metodi esclusi, essi devono consentire l'avvio e la terminazione del servizio specifico, azioni per le quali è necessario interfacciarsi direttamente con le API della tecnologia utilizzata e quindi deve essere effettuata una loro specializzazione ma sempre eseguendo anche la l'implementazione generica.

5.1.2 GeolocationServiceListener

Il *GeolocationServiceListener* rappresenta l'entità ascoltatore in attesa della ricezione di notifiche riguardanti aggiornamenti di posizione e generazione degli eventi di movimento provenienti dal servizio di geolocalizzazione al quale è collegata. A questo ascoltatore viene associato un servizio specifico e l'identificatore del centro di tuple sul quale è attivo tale servizio: risulta chiaro quindi che l'associazione tra questi due componenti risulti di tipo 1:1, perciò, per ogni servizio verrà creato e collegato un nuovo ascoltatore che sarà in ascolto degli eventi provenienti solo da esso. In particolare, una volta ricevuti gli aggiornamenti l'ascoltatore è incaricato di aggiornare la posizione della macchina virtuale e, se richiesto, di occuparsi della generazione e notifica degli eventi di movimento.

La struttura generica di questa entità è stata definita tramite un'interfaccia, denominata `IGeolocationServiceListener`, che incapsula tutti i metodi principali necessari per una corretta interazione tra servizio e ascoltatore del servizio, nonché tra quest'ultimo e il centro di tuple:

`void locationChanged(IPlace place)` – utilizzato dal servizio associato per notificare il cambiamento di posizione. Il parametro *place* rappresenta la nuova posizione.

`void moving(int type, String space, IPlace place)` – utilizzato dal servizio associato per notificare che il dispositivo ha cominciato un movimento

oppure che lo ha terminato. Il parametro *type* si riferisce al tipo di evento, ovvero se il movimento è cominciato (evento `from`) oppure terminato (evento `to`), *space* rappresenta la tipologia di posizione (`ph`, `map`, `org`, `ip` o `dns`) e infine *place* rappresenta la posizione di partenza/arrivo.

`GeolocationService getService()` – utilizzato per recuperare il servizio associato all'ascoltatore.

`GeoServiceId getServiceId()` – utilizzato per recuperare l'identificatore del servizio associato all'ascoltatore.

`TucsonTupleCentreId getTcId()` – utilizzato per recuperare l'identificatore del centro di tuple sul quale è attivo il servizio associato all'ascoltatore.

L'implementazione concreta di questa struttura è rappresentata dalla classe `GeolocationServiceListener`, la quale, ricevendo notifiche da parte del servizio al quale è collegata, deve reagire opportunamente aggiornando la posizione della `RespectVM` oppure generando e notificando ad essa gli eventi spaziali eventualmente richiesti dalla specifica di comportamento iniettata nel centro di tuple.

A questo scopo è stato necessario definire un nuovo contesto, rappresentato dalla classe `SpatialContext`, tramite il quale recuperare la specifica istanza della `RespectVM` sulla quale notificare gli aggiornamenti riguardati la posizione. Il contesto spaziale implementa l'interfaccia `ISpatialContext` che ne definisce la seguente struttura:

`void notifyInputEnvEvent(InputEvent ev)` – utilizzato per notificare alla macchina virtuale la presenza di un nuovo evento spaziale. Tramite questo metodo l'evento spaziale appena generato viene inserito nel multi-set *SitE* degli eventi situati e notificato alla `RespectVM`.

`long getCurrentTime()` – utilizzato per recuperare il tempo locale della macchina virtuale.

`Position getPosition()` – utilizzato per recuperare la posizione corrente della macchina virtuale.

`void setPosition(IPlace place)` – utilizzato per impostare una specifica tipologia di posizione della macchina virtuale. Tramite questo metodo si accede direttamente alla posizione della macchina virtuale, modificandola.

Definito questo nuovo contesto, l'entità ascoltatore è in grado di interagire con la `RespectVM` tramite il metodo `getSpatialContext(TupleCentreId id)` definito all'interno della classe `RespectTCContainer` che, a sua volta, recupera il contesto spaziale invocando il metodo `getSpatialContext()` definito nella classe `RespectTC`. Quest'ultimo si occupa di creare un nuovo contesto inizializzandolo con l'istanza della `RespectVM` relativa al centro di tuple specificato dall'ascoltatore e quindi quello sul quale è attivo il servizio di geolocalizzazione.

Quando l'ascoltatore riceve una notifica di aggiornamento della posizione dall'entità `GeolocationService`, si occupa semplicemente di recuperare il contesto spaziale ed invocare su di esso il metodo `setPosition(IPlace place)` specificando in ingresso la nuova posizione.

Listing 5.2: GeolocationServiceListener.java

```
[...]

@Override
public void locationChanged(IPlace place) {
   ISpatialContext context =
      RespectTCContainer.getRespectTCContainer()
         .getSpatialContext(tcId.getInternalTupleCentreId());
   context.setPosition(place);
}

[...]
```

Analogamente, quando l'ascoltatore riceve una notifica relativa all'inizio o alla fine di un movimento, si occupa di recuperare il contesto spaziale, dopodiché a seconda della tipologia di evento specificata in ingresso (*type*) viene costruita la tupla logica corrispondente, l'operazione relativa ed infine viene creato un nuovo `InputEvent`, che viene notificato alla macchina virtuale tramite il metodo `notifyInputEnvEvent(InputEvent ev)`. A questo punto la `RespectVM` elaborerà l'evento valutando anche le reazioni innescate da esso.

Listing 5.3: GeolocationServiceListener.java

```
[...]

@Override
public void moving(int type, String space, IPlace place) {
   try {
      ISpatialContext context =
         RespectTCContainer.getRespectTCContainer()
         .getSpatialContext(tcId.getInternalTupleCentreId());

      LogicTuple tuple = null;
      RespectOperation op = null;

      if( type == RespectOperation.OPTYPE_FROM ){
         tuple =
            LogicTuple.parse(
               "from(" + space + "," + place.toTerm() + ")"
            );
         op = RespectOperation.makeFrom( null, tuple, null);

      }else if( type == RespectOperation.OPTYPE_TO ){
         tuple =
            LogicTuple.parse(
               "to(" + space + "," + place.toTerm() + ")"
            );
         op = RespectOperation.makeTo( null, tuple, null);

      }

      InputEvent ev =
               new InputEvent(
                        this.service.getServiceId(),
```

```
                        op,
                        this.tcId,
                        context.getCurrentTime(),
                        context.getPosition()
                );

        context.notifyInputEnvEvent(ev);

    } catch (InvalidLogicTupleException e) {
        e.printStackTrace();
    }
}

[...]
```

Con riferimento alle entità *GeolocationService* e *GeolocationServiceListener*, in Figura 5.2 viene mostrata la gerarchia delle classi relative e le loro relazioni.

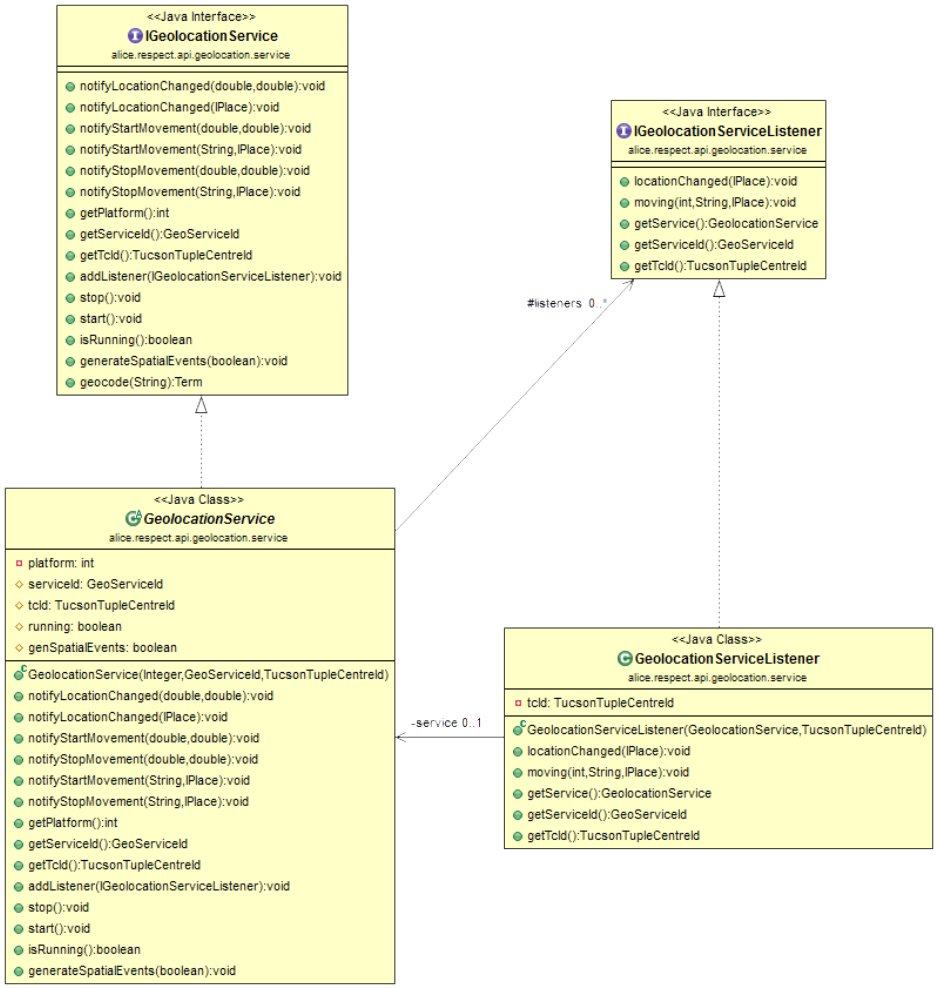


Figura 5.2: GeolocationService e GeolocationServiceListener - Gerarchia

5.1.3 GeolocationServiceManager

Il *GeolocationServiceManager* rappresenta l'entità responsabile della creazione, registrazione e rimozione dei servizi di geolocalizzazione. Tramite esso è possibile risalire ad informazioni utili per poter interfacciarsi con un determinato servizio.

Questa entità è stata definita adottando il pattern *Singleton* ed incapsula una mappa chiave-valore, rappresentante la lista dei servizi di geolocalizzazione attivi nell'infrastruttura di coordinazione, inoltre fornisce i metodi necessari per:

creare servizi – è possibile creare un nuovo servizio sfruttando il metodo `createNodeService` specificando la piattaforma di esecuzione, l'identificatore del servizio, la sua classe e il centro di tuple sul quale è attivo. Dopo avere creato una nuova istanza di `GeolocationService`, esso viene aggiunto alla lista dei servizi attivi e gli viene agganciato un nuovo `GeolocationServiceListener`.

aggiungere servizi – nel caso in cui si disponga di servizi già istanziati, è possibile aggiungerli al sistema tramite il metodo `addService`, il quale, preso in ingresso un `GeolocationService` si occupa di aggiungerlo alla lista dei servizi attivi.

rimuovere servizi – è possibile rimuovere servizi in modalità selettiva, tramite il metodo `destroyService(id)`, oppure globale, tramite il metodo `destroyAllServices()`. Nel primo caso il servizio viene interrotto e rimosso dalla lista dei servizi attivi, nel secondo caso, utile in fase di shutdown dell'intero sistema, tutti i servizi vengono interrotti e la lista dei servizi attivi viene svuotata.

recuperare servizi – è possibile recuperare servizi in due modi. Il primo consiste nello specificare il nome del servizio come parametro di ingresso al metodo `getServiceByName(name)`, il secondo invece nello specificare l'identificatore della piattaforma di esecuzione come parametro di ingresso al metodo `getAppositeService(platform)`. In entrambi i casi viene restituito, se esiste, il servizio associato al parametro specificato.

In particolare, per la creazione dinamica e generica di un servizio vengono sfruttate le librerie *Reflection* presenti in Java, tramite le quali, a partire da una generica classe è possibile istanziarla, interrogarla e conoscere i nomi dei suoi metodi, dei suoi attributi e di tutto ciò che essa contiene, direttamente durante la sua esecuzione. Inoltre è possibile interrogarne i metodi, o invocarli, come se ci si trovasse in un normale flusso di esecuzione.

Come spiegato all'inizio della Sottosezione 5.1.1, ad un servizio viene associata anche la piattaforma di esecuzione con la quale esso è compatibile. Questa associazione permette, all'interno dell'infrastruttura, di recuperare il servizio corretto a seconda della piattaforma sulla quale è in esecuzione il nodo. Questo viene effettuato tramite il metodo `getAppositeService(int platform)`, il quale restituisce un servizio a cui è associata la piattaforma di esecuzione specificata dal parametro *platform*.

Il rilevamento della piattaforma di esecuzione viene effettuato tramite il metodo `getPlatform()` definito all'interno della classe di utilità `PlatformUtils` e che restituisce l'identificatore associato alla piattaforma stessa (definito nell'interfaccia `Platforms`).

Listing 5.4: PlatformUtils.java

```
[...]

public static int getPlatform() {
   Properties p = System.getProperties();
   String type = (String) p.getProperty("os.name");
   type = type.toLowerCase();

   String specType =
          (String) p.getProperty("java.runtime.name");
   specType = specType.toLowerCase();

   if (type.contains("windows")) {
      return Platforms.WINDOWS;
   } else if (type.contains("linux")) {
      if(specType.contains("android")){
         return Platforms.ANDROID;
      } else {
         return Platforms.LINUX;
      }
   } else {
      return Platforms.MACOS;
   }
}

[...]
```

Il compito della creazione del servizio di geolocalizzazione viene assegnato al nodo TuCSoN, il quale delegherà l'azione al *GeolocationServiceManager*. A tale scopo viene esteso il comportamento del sistema in fase di configurazione, definendo nuove entità aventi il compito di configurare i servizi di geolocalizzazione, attendendo richieste da parte degli agenti e delegandole al *GeolocationServiceManager*.

5.2 Configurazione del servizio

Per la configurazione iniziale del sistema, si sfrutta un agente TuCSoN, denominato *GeolocationConfigAgent*, appositamente definito per gestire le richieste di creazione e rimozione dei servizi di geolocalizzazione.

Questo agente viene avviato contestualmente all'avvio del nodo TuCSoN stesso e resta in attesa di richieste su un centro di tuple per la configurazione, denominato *geolocationConfigTC*, nel quale viene iniettata una particolare specifica di comportamento definita del file *geolocation_spec.rsp* e riportata di seguito:

Listing 5.5: geolocation_spec.rsp

```
reaction(
   out( createGeolocationService(Sid,Sclass,Stcid) ),
   response,
   (
      out( cmd(createGeolocationService) )
   )
).
```

```
reaction(
   out( destroyGeolocationService(Sid) ),
   response,
   (
      out( cmd(destroyGeolocationService) )
   )
).
```

Invocando primitive `out` su questo centro di tuple, è possibile configurare il sistema e prepararlo per ricevere aggiornamenti su posizione e movimento, creando nuovi servizi di geolocalizzazione o rimuovendoli qualora non risultino più necessari. Come si può notare dalla specifica ReSpecT riportata sopra, il centro di tuple *geolocationConfigTC* reagisce all'inserimento delle seguenti tuple:

createGeolocationService(Sid, Sclass, Stcid) – utilizzata per la creazione di un nuovo servizio identificato da *Sid* (`GeoServiceId`), la cui implementazione specifica è definita nella classe *Sclass* (user-defined) e al quale viene associato un centro di tuple, sul quale tale servizio sarà attivo, identificato da *Stcid* (`TucsonTupleCentreId`).

destroyGeolocationService(Sid) – utilizzata per la rimozione del servizio di golocalizzazione identificato da *Sid* (`GeoServiceId`).

All'inserimento di una tupla tra quelle elencate, all'interno del centro di tuple *geolocationConfigTC*, quest'ultimo reagisce inserendo a sua volta una tupla, nella forma `cmd(X)`, rappresentante il comando corrispondente alla richiesta effettuata. L'agente *GeolocationConfigAgent* resta in attesa di una qualsiasi tupla della forma `cmd(X)` e, a seconda del valore unificato con la variabile *X*, si occupa di recuperare (tramite la primitiva `in`) la richiesta corrispondente ed elaborarla interagendo con l'entità *GeolocationServiceManager*.

In particolare, per quanto riguarda le richieste di creazione, successivamente ad una preliminare preparazione dei dati, l'agente interagisce con il *GeolocationServiceManager*, invocando il metodo `createNodeService`.

Listing 5.6: GeolocationConfigAgent.java

```
[...]

if (CREATE_GEOLOCATION_SERVICE.equals(name)) {

   [...retrieve the tuple
       ''createGeolocationService(Sid,Sclass,Stcid)''...]

   [...prepare data (platform, service id, tcid)...]

   GeolocationServiceManager.getGeolocationManager()
         .createNodeService(platform,
                            sId,
                            t.getArg(1).toString(),
                            tcId
         );
}
```

```
[...]
```

Analogamente, per quanto riguarda le richieste di eliminazione, successivamente ad una preliminare preparazione dei dati, l'agente interagisce con il *GeolocationServiceManager* invocando, in questo caso, il metodo *destroyService*.

Listing 5.7: GeolocationConfigAgent.java

```
[...]

if (DESTROY_GEOLOCATION_SERVICE.equals(name)) {

   [...retrieve the tuple
       ''destroyGeolocationService(Sid)''...]

   [...prepare data (service id)...]

   GeolocationServiceManager.getGeolocationManager()
      .destroyService(sId);
}

[...]
```

L'approccio di sfruttare il centro di tuple permette, in linea di principio, di aggiungere e rimuovere i servizi di geolocalizzazione a tempo di esecuzione e quindi garantisce versatilità e adattatività del sistema.

5.3 Geolocalizzazione degli agenti

Quanto mostrato fin ora permette di registrare servizi di geolocalizzazione ad nodo TuCSoN, dunque permette di aggiornare la posizione della sola `RespectVM`. Per questo motivo e poiché un agente potrebbe essere in esecuzione su un dispositivo nel quale la macchina virtuale non è in esecuzione, risulta necessario separare gli ambiti fornendo la possibilità di agganciare personali servizi di geolocalizzazione agli agenti TuCSoN, anche laddove non sia in esecuzione un nodo.

A tale scopo è stata modificata la classe `ACCProxyAgentSide`, affiancando nuove funzioni a quelle discusse nella Sottosezione 4.3.4. In particolare, sono stati aggiunti tre metodi:

`void setPosition(IPlace p)` – utilizzato per impostare una specifica tipologia di posizione dell'agente TuCSoN. Tramite questo metodo si accede direttamente alla posizione dell'agente, modificandola.

`attachGeolocationService(String class, TucsonTupleCentreId tcId)` – utilizzato per collegare all'agente un servizio di geolocalizzazione la cui implementazione è definita nella classe *class* e a cui è associato un centro di tuple, identificato da *tcId*.

`void createGeolocationService(TucsonTupleCentreId tcId, String class)` – utilizzato per creare un nuovo servizio di geolocalizzazione la cui implementazione è definita nella classe *class* e a cui è associato un centro di tuple, identificato da *tcId*.

Se un agente vuole conoscere la propria posizione, dovrà semplicemente invocare il metodo `attachGeolocatioService` sull'`ACCProxyAgentSide` a lui associato. Nello specifico, tramite tale metodo l'`ACC` controlla se esiste già un servizio associato all'agente, in tal caso si occupa solo di avviarlo (se non risulta già in esecuzione).

Listing 5.8: ACCProxyAgentSide.java

```
[...]

public void attachGeolocationService(String className,
   TucsonTupleCentreId tcId)
   throws TucsonInvalidTupleCentreIdException {

   GeolocationServiceManager geolocationManager =
         GeolocationServiceManager.getGeolocationManager();

   if(geolocationManager.getServices().size() > 0) {
      GeolocationService geoService =
               geolocationManager
                  .getServiceByName(
                     this.aid.getAgentName() + "_GeoService"
                  );

      if(geoService != null) {
         this.myGeolocationService = geoService;
         this.log("A geolocation service is already attached
                   to this agent, using this.");

         if(!geoService.isRunning()) {
            geoService.start();
         }
      } else {
         this.createGeolocationService(tcId, className);
      }
   } else {
      this.createGeolocationService(tcId, className);
   }
}

[...]
```

Nel caso in cui non esista alcun servizio collegato all'agente ne viene creato uno nuovo tramite il metodo `createGeolocationService`, sfruttando la stessa entità *GeolocationServiceManager* utilizzata per la geolocalizzazione del nodo TuCSoN. In particolare, dati in ingresso il nome della classe che definisce l'implementazione specifica del servizio e l'identificatore del centro di tuple da associare ad esso, l'`ACC` si interfaccia con il *GeolocationServiceManager*, delegando ad esso la creazione del servizio invocando il metodo `createAgentService`. Fatto ciò l'`ACC` si occupa solo di avviare il servizio appena creato.

Listing 5.9: ACCProxyAgentSide.java

```
[...]
```

```
private void createGeolocationService(
    TucsonTupleCentreId tcId, String className) {
    try {

        [...prepare data (platform, service id)...]

        this.myGeolocationService =
              GeolocationServiceManager.getGeolocationManager()
                    .createAgentService(platform,
                                        sId,
                                        className,
                                        tcId,
                                        this
                    );

        if(this.myGeolocationService != null) {
            this.myGeolocationService.start();
        } else {
            this.log("Error during service creation");
        }
    } catch (Exception e) {
        this.log(
            "Error during service creation: " + e.getMessage()
        );
    }
}

[...]
```

Come si può notare, anche l'entità *GeolocationServiceManager* è stata arricchita con il metodo `createAgentService(Integer platform, GeoServiceId sId, String className, TucsonTupleCentreId tcId, ACCProxyAgentSide acc)` che viene utilizzato per la creazione di un nuovo servizio identificato da *sId*, la cui implementazione specifica è definita nella classe *className* (user-defined) e al quale vengono associati un centro di tuple identificato da *tcId* e uno specifico `ACCProxyAgentSide` collegato all'agente.

Questo metodo, analogamente a `createNodeService`, si occupa della creazione di un nuovo servizio, istanziando la classe specificata in ingresso e aggiungendolo alla lista dei servizi attivi. A differenza del servizio creato per il nodo TuCSoN, in questo caso, viene collegato un ascoltatore specifico, denominato *AgentGeolocationServiceListener*, per permettere la corretta interazione con l'`ACCProxyAgentSide` associato all'agente che desidera conoscere la propria posizione.

Analogamente all'entità *GeolocationServiceListener*, all'ascoltatore *AgentGeolocationServiceListener* viene associato un servizio specifico e l'identificatore del centro di tuple sul quale esso è attivo, quindi, anche in questo caso l'associazione tra servizio e ascoltatore risulta di tipo 1:1. In aggiunta, gli viene anche associato uno specifico `ACCProxyAgentSide` che deve ricevere gli aggiornamenti sulla posizione.

L'ascoltatore *AgentGeolocationServiceListener* presenta la medesima struttura di `GeolocationServiceListener`, implementando anch'esso l'interfaccia

`IGeolocationServiceListener`, ma fornendo una differente implementazione dei metodi. In particolare, relativamente agli aggiornamenti di posizione, questa entità si occupa di invocare, sull'`ACCProxyAgentSide` associato, il metodo `setPosition(IPlace place)` specificando in ingresso la nuova posizione.

Listing 5.10: AgentGeolocationServiceListener.java

```
[...]

@Override
public void locationChanged(IPlace place) {
    this.acc.setPosition(place);
}

[...]
```

Successivamente alla definizione dell'entità *AgentGeolocationServiceListener*, la gerarchia delle classi e le loro relazioni è stata leggermente modificata, come mostrato in Figura 5.3.

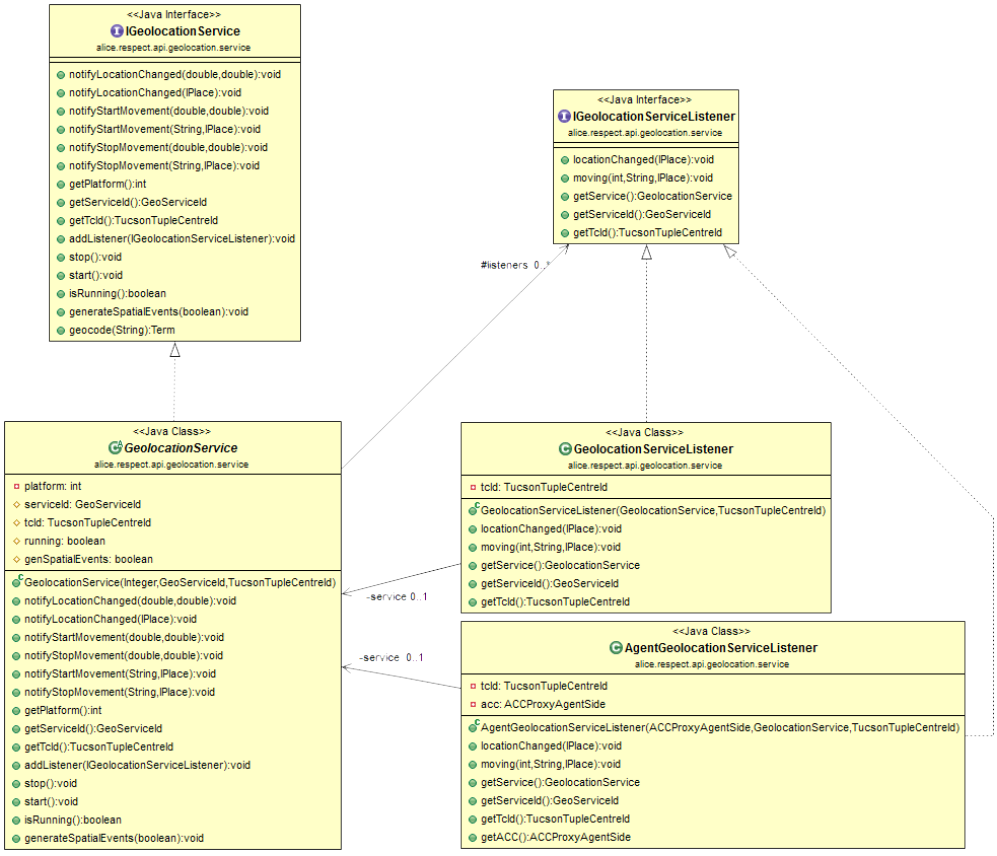


Figura 5.3: GeolocationService - Gerarchia

5.4 Posizione ed operazioni di linking

Quando vengono effettuate operazioni di *linking*, chi richiede l'operazione non è un agente TuCSoN ma un centro di tuple. In questo caso, per la comunicazione con l'`ACCProxyNodeSide` viene utilizzata una terza entità *proxy*, denominata `InterTupleCentreACCProxy`.

Per questo motivo, allo scopo di inserire correttamente la posizione all'interno del messaggio di richiesta, è stato necessario intervenire su quest'ultima definendo un metodo tramite il quale, sfruttando il contesto spaziale (Sottosezione 5.1.2), è possibile recuperare la posizione del centro di tuple che sta richiedendo l'operazione.

Listing 5.11: InterTupleCentreACCProxy.java

```
[...]

public Position getPosition() {
    ISpatialContext context =
       RespectTCContainer.getRespectTCContainer()
             .getSpatialContext(
                   this.aid.getInternalTupleCentreId()
             );
    return context.getPosition();
}

[...]
```

5.5 Esempio applicativo: *MotionLog*

Allo scopo di mostrare una possibile applicazione del servizio di geolocalizzazione, è stata definita la seguente specifica ReSpecT:

Listing 5.12: motionLog_spec.rsp

```
reaction( from(ph,StartP), true,
    ( current_time(StartT),
    out(start_log(StartP,StartT)) )).

reaction( to(ph,ArrP), true,
    ( current_time(ArrT),
    out(stop_log(ArrP,ArrT)) )).

reaction( out(stop_log(ArrP,ArrT)),
    internal,
    ( in(start_log(StartP,StartT)),
      in(stop_log(ArrP,ArrT)),
      out(m_log(StartP,ArrP,StartT,ArrT)) )).
```

Nell'insieme questa specifica di comportamento permette attuare una sorta di *physical motion log*, includendo tempi e posizioni di partenza e arrivo. Infatti, la prima reazione immagazzina informazioni relative all'inizio di un movimento fisico in una tupla `start_log/2`, la seconda alla terminazione del movimento in una tupla `stop_log/2`, infine l'ultima reazione rimuove entrambe queste tuple

e registra i relativi dati in una tupla `m_log/4`, rappresentante le informazioni essenziali relative alla traiettoria del dispositivo mobile che ospita il centro di tuple.

Non avendo ancora sviluppato una piattaforma di geolocalizzazione eseguibile su un dispositivo mobile, per eseguire questo esempio e verificare il corretto funzionamento del livello logico *Geolocation* discusso in questo capito ed implementato nell'infrastruttura TuCSoN, è stata sviluppata una semplice applicazione Java all'interno della quale sono state definite due entità:

1. Un agente TuCSoN che si occupa di effettuare la richiesta di creazione del servizio di geolocalizzazione ed iniettare in esso la specifica di comportamento d'esempio.
2. Un'entità, denominata *MyGeolocationService*, che estende il generico *GeolocationService* e che incapsula un processo (*Thread*) che simula un servizio di geolocalizzazione che si occupa di notificare all'ascoltatore collegato gli aggiornamenti riguardanti posizione e, se la specifica di comportamento lo richiede, di movimento.

In particolare, all'interno all'entità *MyGeolocationService* è stato definito un processo che ciclicamente, in maniera fittizia, si occupa di notificare:

- un aggiornamento della posizione.
- la partenza di un movimento.
- la terminazione di un movimento.

Listing 5.13: MyGeolocationService.java

```
[...]

@Override
public void run() {
   while(!stopFlag.isSet()) {
      float lat = 44.421833f;
      float lng = 12.911683f;
      service.notifyLocationChanged(lat, lng);

      if(this.genSpatialEvents) {
         [...wait 3 seconds...]
         lat = 44.421833f;
         lng = 21.911683f;
         service.notifyStartMovement(lat, lng);

         [...wait 3 seconds...]
         lat = 44.428833f;
         lng = 12.912944f;
         service.notifyStopMovement(lat, lng);
      }
      [...wait 30 seconds...]
   }
}

[...]
```

In questo modo è stato possibile verificare il corretto funzionamento del livello logico *Geolocation* e la corretta interazione tra le entità discusse in questo capitolo.

Per esempio, considerando le coordinate mostrate nello stralcio di codice riportato sopra e la specifica ReSpecT d'esempio, all'invocazione del metodo *notifyStartMovement(44.421833,12.911683)* viene generato un evento `from(ph, coords(44.421833,12.911683))` che porta all'innesco della prima reazione, nel cui body viene invocato il predicato di osservazione `current_time(StartT)`.

Tramite questo predicato viene recuperato il tempo locale della macchina virtuale espresso in millisecondi allo scopo di acquisire il tempo al quale è cominciato il movimento. Supponendo che la variabile *StartT* unifichi con un tempo pari a 5150 ms, viene inserita una nuova tupla *start_log(5150, coords(44.421833,12.911683))*.

In seguito ad una breve attesa, viene invocato il metodo *notifyStopMovement(44.428833,12.912944)* che causa la generazione di un evento `to(ph, coords(44.428833,12.912944))`, il quale porta a sua volta all'innesco della seconda reazione. Nel body di quest'ultima viene invocato ancora una volta il predicato di osservazione `current_time(ArrT)`, allo scopo di recuperare il tempo al quale è terminato il movimento. Supponendo che siano trascorsi tre secondi dall'inizio dello spostamento, la variabile *ArrT* unifica con un tempo pari a 8150 ms. Fatto questo, viene inserita una nuova tupla *stop_log(8150,coords(44.428833, 12.912944))*, il cui inserimento causa l'innesco della terza reazione.

Nel body della terza reazione vengono recuperate le tuple `start_log/2` e `stop_log/2` al fine di inserirne una nuova che contiene tutte le informazioni relative al movimento effettuato dal nodo TuCSoN. Infatti, viene inserita la tupla `m_log(coords(44.421833,12.911683),coords(44.428833,12.912944),5150, 8150)` che riassume la traiettoria compiuta dal dispositivo che ospita il centro di tuple.

Capitolo 6

TuCSoN Mobile

Implementata l'estensione *space-aware* di ReSpecT ed estesa l'infrastruttura TuCSoN con il livello logico *Geolocation*, al fine di verificare concretamente il funzionamento dei predicati spaziali e del servizio di geolocalizzazione, si è deciso di sfruttare i porting di TuCSoN e dello strumento *Inspector* su dispositivo *Android*, già sviluppati in precedenza dagli ingegneri Ridolfi e D'Elia, integrandoli, estendendoli e migliorandoli al fine di sviluppare un'unica applicazione che permetta di avviare un nodo TuCSoN, fornendo anche diverse funzioni per interagire con esso e verificare il corretto funzionamento delle funzionalità che l'infrastruttura mette a disposizione, con particolare riguardo a quelle *space-aware*.

6.1 Caratteristiche generali

L'applicazione è stata sviluppata per dispositivi che eseguono le *API 17* (Android 4.2.2) o successive e presenta le seguenti sezioni, alcune migliorate ed altre sviluppate ex-novo:

Tucson Node Service – sezione tramite la quale è possibile avviare o interrompere il nodo TuCSoN.

CLI – sezione tramite la quale è possibile invocare tutte le primitive di coordinazione, richiedendone l'elaborazione al nodo TuCSoN in esecuzione sul dispositivo oppure ad uno remoto.

Inspector – sezione tramite la quale è possibile avviare lo strumento *Inspector*, agganciandolo al nodo TuCSoN in esecuzione sul dispositivo oppure ad uno remoto.

Geolocation – sezione tramite la quale è possibile avviare o interrompere il servizio di geolocalizzazione, agganciandolo al nodo TuCSoN in esecuzione locale sul dispositivo.

Tests – sezione tramite la quale è possibile eseguire i test per verificare il corretto funzionamento dell'applicazione, delle primitive invocate e dei predicati di osservazione, guardie ed eventi forniti dal linguaggio di coordinazione.

A differenza del porting di TuCSoN preesistente, ogni sezione è stata arricchita con un'area dedicata al *log* dell'applicazione relativo alle funzionalità

utilizzabili in essa. Nel *log* vengono mostrate, tramite semplici stampe, le operazioni che l'applicazione sta eseguendo nonché il loro esito, riportando anche gli eventuali errori che si verificano. Grazie a questo l'utente è quindi costantemente a conoscenza dello stato delle operazioni richieste ed è in grado di comprendere e risalire alla sorgente di eventuali problemi.

Come nel porting preesistente, tramite la pressione del tasto *menu* sul dispositivo utilizzato si può accedere alle impostazioni, tramite le quali è possibile impostare i parametri di esecuzione relativi alle funzionalità offerte dall'applicazione. Tali impostazioni sono state estese, aggiungendo anche i parametri relativi alle nuove funzionalità dell'applicazione. In generale si possono impostare i seguenti parametri:

- *Indirizzo IP*: l'indirizzo di rete sul quale è in esecuzione il nodo TuCSoN.
- *Nome del centro di tuple*: il nome del centro di tuple di destinazione.
- *Numero di porta*: il numero di porta sulla quale il nodo TuCSoN è in ascolto.
- *Identificatore del servizio di geolocalizzazione*: l'identificatore da associare al servizio di geolocalizzazione.

A differenza della versione precedente, in ogni sezione è stata inserita l'indicazione dei parametri impostati per le funzionalità fornite da essa. In questo modo, l'utente è in grado di essere perfettamente a conoscenza (in ogni momento) dei parametri che verranno utilizzati durante l'esecuzione delle operazioni richieste.

6.2 Tucson Node Service

In questa sezione è possibile visualizzare lo stato del nodo TuCSoN eventualmente attivo sul dispositivo.

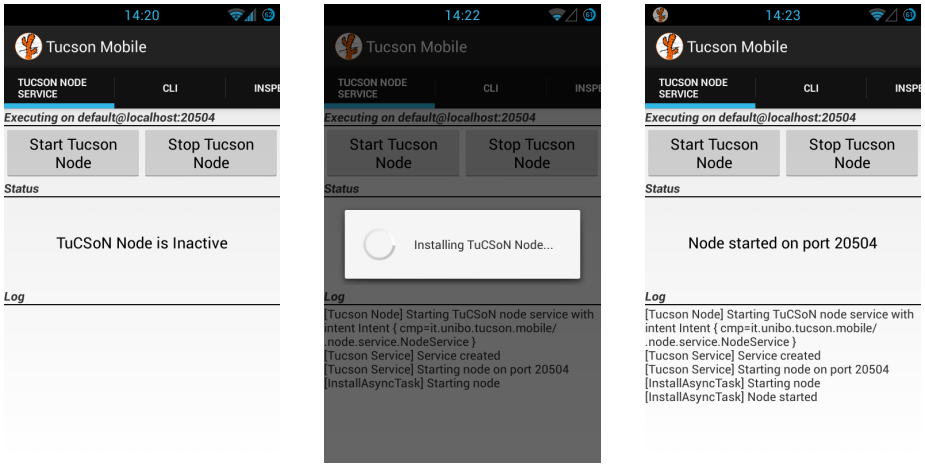


Figura 6.1: Tucson Node Service - All'avvio, in installazione ed in esecuzione

Osservando le catture, in alto vengono indicati i parametri di configurazione relativi al nodo ovvero il nome del centro di tuple, il suo indirizzo di rete e la porta sulla quale è in ascolto. L'unico parametro definibile tramite le impostazioni dell'applicazione è il numero di porta mentre indirizzo di rete e nome del centro di tuple vengono impostati di base a *localhost* e *default*, rispettivamente.

Alla pressione del pulsante *Start Tucson Node* l'applicazione effettua l'installazione di un nuovo nodo TuCSoN; questa operazione richiede alcuni secondi quindi, durante la sua esecuzione viene mostrata una finestra di dialogo che blocca l'interazione con l'applicazione fintanto che l'installazione non è terminata, momento in cui viene aggiornato lo stato del nodo indicando che esso risulta attivo sulla porta specificata.

Nel *log* vengono mostrati i vari passaggi effettuati dall'applicazione per l'avvio (e la terminazione) del nodo TuCSoN, nonché il suo stato ed eventuali errori che si verificano.

Differenze tecniche rispetto alla versione precedente

Nella versione precedente dell'applicazione, l'avvio del nodo TuCSoN veniva effettuato tramite l'utilizzo di un processo (*Thread*) che si occupava, una volta istanziato l'oggetto di tipo *TucsonNode*, di invocare il metodo `install()` implementato internamente alla libreria TuCSoN. Lo stesso processo restava poi bloccato su un semaforo (*Semaphore*) in attesa del comando di *stop*, ricevuto il quale effettuava l'interruzione del nodo invocando il metodo `shutdown()`.

Poiché entrambe queste operazioni risultano essere di breve durata, si è deciso di modificare l'implementazione sostituendo al *Thread* un oggetto di tipo *AsynchTask*, fornito proprio a questo scopo dalla libreria *Android*. L'*AsynchTask* permette di eseguire operazioni in background sul dispositivo e notificare i risultati sul processo che gestisce la UI, senza il bisogno di avere un *Thread* in esecuzione e bloccato in attesa di comandi: cosa chiaramente poco performante. Inoltre, utilizzando gli *AsynchTask* è possibile sapere il momento esatto in cui una data operazione è stata avviata o è stata completata. In particolare, sono stati implementati due *AsynchTask* distinti, uno incaricato di avviare il nodo TuCSoN e l'altro di interromperlo, avviati solo quando richiesto dall'utente e che terminano l'esecuzione non appena completata l'operazione assegnata.

Inoltre, sono stati utilizzati gli strumenti *BroadcastReceiver* e *Handler*, forniti dalla libreria *Android*, allo scopo di rendere più agevole ed estendibile l'interazione tra le funzionalità offerte da questa sezione e la UI stessa di *Android*.

6.3 CLI

In questa sezione è possibile sfruttare lo strumento *Command Line Interpreter* per invocare le primitive di coordinazione, richiedendone l'elaborazione al nodo TuCSoN in esecuzione sul dispositivo oppure ad uno remoto.

Osservando le catture, in alto vengono indicati i parametri di configurazione relativi allo strumento *CLI* ovvero il nome del centro di tuple, l'indirizzo di rete e la porta sui quali invocare le primitive di coordinazione. In questo caso è possibile impostare tutti e tre i parametri tramite le impostazioni dell'applicazione, in modo tale da poter effettuare operazioni anche su un nodo che non sia in esecuzione localmente sul dispositivo.

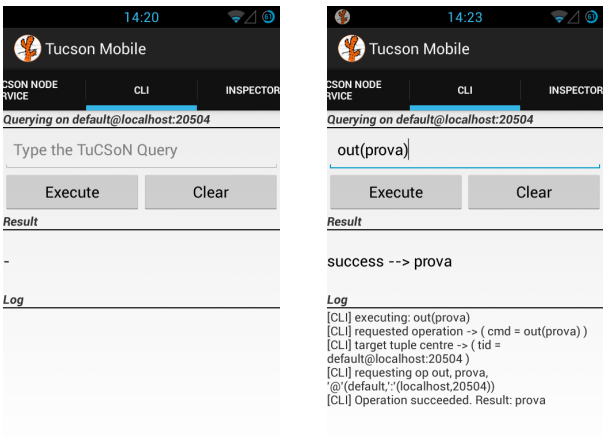


Figura 6.2: CLI - All'avvio ed in esecuzione

Innanzitutto l'utente deve specificare nella casella di testo l'operazione che vuole effettuare indicando anche la tupla (o il template) che il sistema deve utilizzare (e.g. *out(prova)*), dopodiché, alla pressione del pulsante *Execute* l'applicazione si occupa di avviare un nuovo agente TuCSoN incaricato di richiedere l'operazione specificata dall'utente. Nel caso in tutto sia corretto, la primitiva viene invocata sul nodo specificato, altrimenti (ad esempio l'utente ha commesso un errore di sintassi) l'applicazione indica che si è verificato un errore.

Inoltre, se l'utente digita i comandi *help*, *man* oppure *syntax*, viene stampata la guida di utilizzo dello strumento *CLI* nella quale vengono riportate le istruzioni di utilizzo.

Nel *log* vengono mostrati i vari passaggi effettuati dall'applicazione per l'invocazione di una primitiva di coordinazione, nonché il risultato dell'operazione ed eventuali errori che si verificano.

Differenze tecniche rispetto alla versione precedente

A differenza della versione precedente, in questa nuova sezione stati utilizzati gli strumenti *BroadcastReceiver* e *Handler*, forniti dalla libreria *Android*, allo scopo di rendere più agevole ed estendibile l'interazione tra le funzionalità offerte da questa sezione e la UI stessa di *Android*.

6.4 Inspector

In questa sezione è possibile sfruttare lo strumento *Inspector*, agganciandolo ad un nodo TuCSoN in esecuzione sul dispositivo (oppure ad uno remoto), allo scopo analizzare il contenuto del centro di tuple, gli eventi in attesa di essere elaborati e le reazioni innescate.

Osservando le catture, in alto vengono indicati i parametri di configurazione relativi allo strumento *Inspector* ovvero il nome del centro di tuple, l'indirizzo di rete e la porta relativi al nodo TuCSoN che si vuole analizzare. Anche in questo caso è possibile impostare tutti e tre i parametri tramite le impostazioni

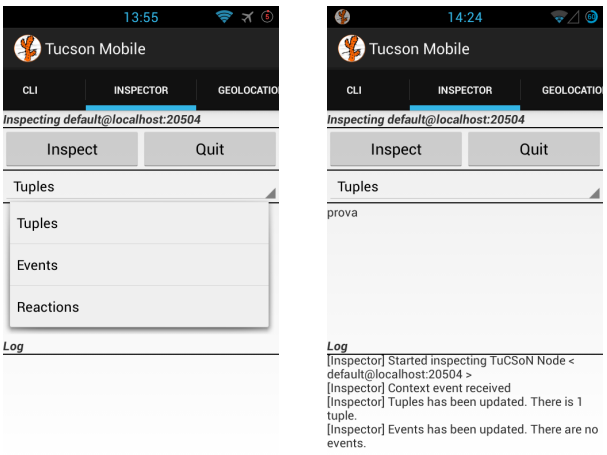


Figura 6.3: Inspector - All'avvio ed in esecuzione

dell'applicazione, in questo modo è possibile analizzare anche un nodo che non risulta in esecuzione localmente sul dispositivo.

Innanzitutto l'utente deve avviare un nodo TuCSoN tramite la sezione *Tucson Node Service* (Sezione 6.2) oppure specificare indirizzo di rete e porta di un nodo remoto. Fatto ciò, alla pressione del pulsante *Inspect*, l'applicazione effettua la creazione di un nuovo *Inspector* e lo registra al nodo specificato come ascoltatore degli aggiornamenti. Effettuando operazioni su tale nodo, l'*Inspector* collegato riceverà automaticamente notifiche riguardanti tuple, eventi e reazioni, aggiornando le aree di testo relative e mostrando lo stato del centro di tuple.

Nel *log* vengono mostrati i vari passaggi effettuati dall'applicazione per avviare o interrompere l'*Inspector*, nonché gli aggiornamenti di tuple, eventi e reazioni, indicando anche la quantità di ciascuno, ed eventuali errori che si verificano.

Differenze tecniche rispetto alla versione precedente

Come già accennato all'inizio di questo capitolo è stata effettuata l'integrazione del porting dello strumento *Inspector*, sviluppato dall'ingegnere D'Elia, con la nuova applicazione sviluppata in questa fase. Nello specifico, sono state conglobate tutte le funzionalità dello strumento *Inspector* in un'unica sezione tramite la quale, in ogni momento, è possibile scegliere il tipo di informazioni da visualizzare (tuple, eventi o reazioni), selezionando la voce opportuna tramite l'oggetto grafico *Spinner* inserito appositamente nell'interfaccia utente.

Effettuando l'integrazione stato necessario correggere alcuni malfunzionamenti ed aggiungere la funzionalità che permette di mostrare nel *log* il numero di tuple presenti nel centro di tuple, gli eventi in attesa di essere elaborati e le reazioni innescate.

Inoltre, anche in questo caso, sono stati utilizzati gli strumenti *BroadcastReceiver* e *Handler*, forniti dalla libreria *Android*, allo scopo di rendere più agevole ed estendibile l'interazione tra le funzionalità offerte da questa sezione e la UI stessa di *Android*.

6.5 Geolocation

In questa sezione è possibile creare (ed eliminare) un servizio di geolocalizzazione agganciandolo al nodo TuCSoN in esecuzione sul dispositivo. In questo caso è stata effettuata la progettazione ed implementazione completa della sezione e delle sue funzionalità in quanto, per ovvi motivi, nella versione precedente non era presente.

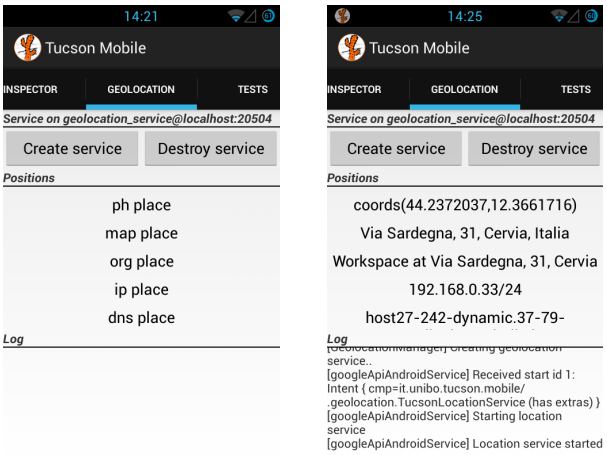


Figura 6.4: Geolocation - All'avvio ed in esecuzione

Osservando le catture, in alto vengono indicati i parametri di configurazione utilizzati per la creazione del servizio, ovvero il nome del centro di tuple, l'indirizzo di rete e la porta relativi al nodo TuCSoN che deve ricevere gli aggiornamenti di posizione e movimento. In questo caso è possibile impostare soltanto l'identificatore del servizio che si vuole creare e il nome del centro di tuple al quale agganciarlo; l'indirizzo di rete viene impostato di base al valore *localhost*, la porta viene invece recuperata dalla relativa impostazione associata al nodo TuCSoN in esecuzione localmente sul dispositivo.

Innanzitutto l'utente deve avviare un nodo TuCSoN tramite la sezione *Tucson Node Service* (Sezione 6.2) e, alla pressione del pulsante *Create service*, l'applicazione si occupa avviare un agente TuCSoN incaricato di effettuare la configurazione del servizio inserendo nel centro di tuple *geolocationConfigTC* una tupla nella forma *createGeolocationService(Sid, Sclass, Stcid)* la quale, seguendo la dinamica descritta nella Sezione 5.2, porta alla creazione e registrazione di un nuovo servizio di geolocalizzazione agganciato al centro di tuple specificato da *Stcid*.

Listing 6.1: CreateGeolocationServiceAgent.java

```
[...]

private void createServices()
    throws TucsonOperationNotPossibleException,
    UnreachableNodeException, OperationTimeOutException{

    this.say("Creating service..");
```

```
    LogicTuple t = new LogicTuple("createGeolocationService",
          new Value(this.serviceId),
          new Value(
              "it.unibo.tucson.mobile.geolocation" +
              ".TucsonGeolocationService"
          ),
          new Value(GEO_SERVICE.toString()) );
    this.acc.out(this.tcId, t, null);

    this.say("Service created");
}

[...]
```

Nel dettaglio, viene creato un servizio la cui implementazione specifica è definita nella classe `TucsonGeolocationService` che, alla ricezione degli aggiornamenti di posizione e movimento, si occupa di effettuare le relative notifiche alla `RespectVM` tramite i metodi definiti nella classe `GeolocationService` della quale è estensione.

La classe `TucsonGeolocationService` incapsula un'entità di tipo *Service* definita nella libreria di *Android* che permette di avviare un processo mantenuto in esecuzione in background sul dispositivo. Tale servizio, denominato `TucsonLocationService` e definito in una classe omonima, implementa l'interfaccia `LocationListener` (anch'essa definita in *Android*) tramite la quale è possibile ricevere gli aggiornamenti di posizione.

In particolare, implementando questa interfaccia, la libreria di *Android* impone l'implementazione specifica del metodo `onLocationChanged` che viene invocato in automatico ogni volta che è disponibile un aggiornamento. Risulta chiaro quindi che è proprio all'interno di quest'ultimo che devono essere generate le varie tipologie di posizione (Sezione 4.3.1) inviate poi all'entità *TucsonGeolocationService* (tramite *BroadcastReceiver*), la quale a sua volta si occupa di effettuare le relative notifiche alla macchina virtuale.

Alla ricezione del comando *start* da parte della macchina virtuale, il *TucsonGeolocationService* si occupa di avviare il relativo servizio utilizzando gli *Intent* di *Android*.

Listing 6.2: TucsonGeolocationService.java

```
[...]

@Override
public void start() {
    super.start();
    final Intent intent =
      new Intent(this.mContext, TucsonLocationService.class);
    intent.putExtra("serviceId",
                    this.getServiceId().getName());
    this.mContext.startService(intent);
}

[...]
```

Ricevuto il comando di avvio da parte della *TucsonGeolocationService*, l'entità *TucsonLocationService* si occupa di recuperare il miglior *provider* disponibile sul dispositivo e registrare il servizio agli aggiornamenti di posizione.

La possibilità di selezionare il miglior *provider* disponibile permette di ricevere aggiornamenti che siano il più precisi possibile, a seconda che il GPS sia attivo o meno sul dispositivo oppure che quest'ultimo risulti collegato ad internet tramite rete Wi-Fi od operatore telefonico.

Listing 6.3: TucsonLocationService.java

```
[...]

@Override
public int onStartCommand(Intent intent, int flags,
    int startId) {

    this.serviceId = intent.getStringExtra("serviceId");

    // Getting LocationManager object from
    // System Service LOCATION_SERVICE
    this.locationManager =
          (LocationManager) getSystemService(LOCATION_SERVICE);

    // Creating a criteria object to retrieve provider
    Criteria criteria = new Criteria();

    // Getting the name of the best provider
    String provider =
             locationManager.getBestProvider(criteria, true);

    // Getting Current Location
    Location location =
             locationManager.getLastKnownLocation(provider);
    this.oldPos = new Location(location);

    if(location!=null){
       onLocationChanged(location);
    }
    this.locationManager
          .requestLocationUpdates(provider,
                                  MIN_TIME_INTERVAL,
                                  MIN_DIST_INTERVAL,
                                  this);

    this.postLog("Location service started");

    return Service.START_STICKY;
}

[...]
```

La registrazione agli aggiornamenti di posizione avviene invocando il metodo `requestLocationUpdate`, specificando in ingresso i seguenti parametri:

provider – nome del *provider* con il quale registrarsi.

minTime – intervallo di tempo minimo tra gli aggiornamenti di posizione, in millisecondi.

minDistance – distanza minima che deve essere percorsa per ricevere un aggiornamento di posizione, in metri.

listener – oggetto di tipo `LocationListener` il cui metodo `onLocationChanged` verrà richiamato per ogni aggiornamento di posizione.

In particolare, i parametri *minTime* e *minDistance* risultano essere molto utili per impostare la granularità degli aggiornamenti di posizione, soprattutto relativamente al controllo della generazione degli eventi spaziali `from` e `to`, se richiesta dalla specifica di comportamento.

A questo proposito, all'interno della classe `TucsonLocationService` è stato definito il metodo `checkMovement`. Nello specifico se la distanza tra posizione attuale e precedente risulta maggiore di una certa quantità (definita dal parametro `EVENTS_GEN_GRANULARITY`), allora viene notificata la partenza di un movimento specificando la posizione precedente. Successivamente, se è cominciato un movimento e la distanza tra posizione attuale e precedente è minore di una certa quantità (`EVENTS_GEN_GRANULARITY`), allora viene notificata la terminazione del movimento in corso specificando la posizione corrente.

Listing 6.4: TucsonLocationService.java

```
[...]

private void checkMovement(Location location) {
   float distance = location.distanceTo(this.oldPos);
   if (!this.moveStarted
            && distance > EVENTS_GEN_GRANULARITY) {
      //movement started
      this.sendBroadcastStartMove(this.oldPos);
      this.moveStarted = true;
   } else if(this.moveStarted
            && distance < EVENTS_GEN_GRANULARITY) {
      //movement terminated
      this.sendBroadcastStopMove(location);
      this.moveStarted = false;
   }
   this.oldPos = new Location(location);
}

[...]
```

Infine, per quanto riguarda la geolocalizzazione degli agenti, è stata definita un'entità distinta, denominata `AgentGeolocationService`, alla quale viene associato un servizio (*Service*) separato denominato `AgentLocationService`. Questo è stato necessario poiché i *Service* di *Android* non permettono istanze multiple, quindi, per garantire il controllo separato dei servizi di geolocalizzazione del nodo e di un agente TuCSoN, l'unico approccio possibile è quello di definire servizi distinti. Inoltre, questo approccio permette, in linea di principio, il trattamento *ad-hoc* delle informazioni sulla posizione in modo tale che agenti e nodo siano eventualmente in grado di elaborare i dati in maniera differente.

6.6 Tests

In questa sezione è possibile eseguire alcuni test predefiniti utilizzando il nodo TuCSoN in esecuzione sul dispositivo (oppure uno remoto).

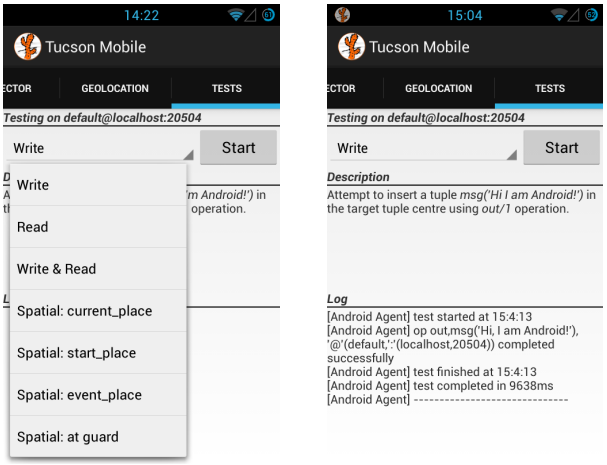


Figura 6.5: Tests - All'avvio ed in esecuzione

Osservando le catture, in alto vengono indicati i parametri di configurazione utilizzati per eseguire i test, ovvero il nome del centro di tuple, l'indirizzo di rete e la porta relativi al nodo TuCSoN da utilizzare. Anche in questo caso è possibile impostare tutti e tre i parametri tramite le impostazioni dell'applicazione, in questo modo è possibile eseguire test anche su un nodo che non risulta in esecuzione localmente sul dispositivo.

Innanzitutto l'utente deve selezionare il test che desidera eseguire, dopodiché nell'area *Description* viene visualizzata una breve descrizione del test e, alla pressione del pulsante *Start*, l'applicazione si occupa di avviare un nuovo agente TuCSoN incaricato di effettuare il test selezionato.

Nel *log* vengono mostrati i vari passaggi effettuati dall'applicazione durante l'esecuzione del test, nonché i tempi di avvio, terminazione e durata dello stesso ed eventuali errori che si verificano.

Differenze tecniche rispetto alla versione precedente

Nella versione precedente dell'applicazione, la sezione dedicata ai test sul nodo TuCSoN permetteva di eseguire solo semplici esecuzioni delle primitive `in` e `out`, quindi solamente test di lettura e scrittura dal/sul centro di tuple.

In questa nuova versione, la sezione di test è stata quasi completamente riprogettata fornendo un'interfaccia utente più intuitiva e tramite la quale viene fornita la possibilità di eseguire svariate tipologie di test, tra i quali quelli relativi ai predicati spaziali, selezionandoli tramite l'oggetto grafico *Spinner* inserito appositamente nell'interfaccia. Ad ogni test, inoltre, è stata associata una descrizione generale, visualizzata in un'apposita area, grazie alla quale l'utente è in grado di comprendere esattamente il test che verrà effettuato, nonché capire anche le informazioni stampate nel *log*.

Questa sezione è stata progettata per permettere allo sviluppatore di aggiungere o eliminare test nella maniera più agevole possibile, richiedendo pochi passi di programmazione.

Infine, anche in questo caso, sono stati utilizzati gli strumenti *BroadcastReceiver* e *Handler*, forniti dalla libreria *Android*, allo scopo di rendere più agevole ed estendibile l'interazione tra le funzionalità offerte da questa sezione e la UI stessa di *Android*.

Capitolo 7

Caso di studio

Allo scopo di dimostrare la semplicità e l'efficacia delle nuove caratteristiche spaziali di ReSpecT progettate e sviluppate in questo elaborato, nel seguito verranno mostrati sviluppo ed implementazione di un caso di studio che pone in evidenza le potenzialità principali della piattaforma *Space-Aware ReSpecT* relativamente alla gestione della posizione e degli eventi di movimento.

7.1 Presence System

7.1.1 Contesto

Il caso di studio si pone come obiettivo quello di sviluppare un'infrastruttura base che modelli uno spazio nel quale tutti gli agenti TuCSoN, collegati al sistema, vengano informati dell'arrivo di altri agenti, nonché della loro posizione, nel momento in cui questi ultimi raggiungono una specifica locazione spaziale all'interno del sistema stesso.

Ogni agente viene associato ad uno specifico nodo TuCSoN in esecuzione sullo stesso dispositivo sul quale esso si trova. Non appena quest'ultimo raggiunge una specifica posizione, l'agente deve procedere alla registrazione ad un entità server che si occupa di mantenere tutte le associazioni tra agente e parametri di identificazione relativi al centro di tuple associato. Tale server è inoltre incaricato di inoltrare a tutti i vicini (gli agenti già registrati) della presenza di un nuovo agente, in quale dovrà poi notificare ad essi la sua posizione attuale.

Un siffatto sistema realizza quello che si può definire *Presence System*, nel quale tutti gli utenti già registrati vengono notificati dell'arrivo di un nuovo utente non appena esso effettua la connessione. Gli scenari applicativi che possono essere presi in considerazione per un sistema di questo tipo sono svariati ma nel seguito ne verrà considerato uno specifico.

7.1.2 Scenario

Per l'implementazione del caso di studio, viene considerato un semplice scenario relativo ad un edificio contenente tre uffici (Figura 7.1) ed un server. In particolare, si suppone che solo in due uffici siano già presenti due impiegati aventi un dispositivo mobile ciascuno, sul quale sono in esecuzione un nodo ed un'agente in attesa di ricevere notifiche relative all'arrivo del terzo impiegato.

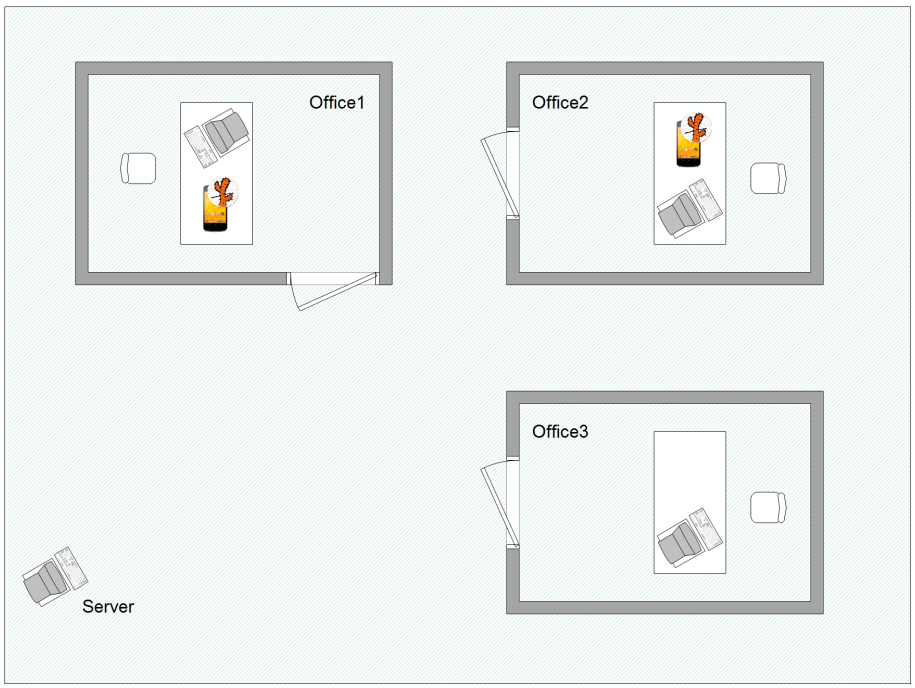


Figura 7.1: Scenario applicativo

Nel momento in cui l'impiegato mancante raggiunge il suo ufficio, l'agente in esecuzione sul suo dispositivo mobile si occupa di effettuare la registrazione al server, sul quale è in esecuzione un nodo TuCSoN dedicato alla gestione delle richieste di registrazione, espresse sotto forma di tuple, rappresentati l'associazione tra agente e nodo in esecuzione sul dispositivo dell'impiegato. Fatto ciò, lo stesso agente si occupa di notificare a tutti i nodi vicini, la cui lista viene recuperata dal server, la propria posizione. Quest'ultima viene specificata come posizione organizzativa e quindi indicherà l'ufficio specifico nel quale si trova l'impiegato appena arrivato.

7.2 Architettura e Interazione

In questa sezione vengono mostrate l'architettura generale che definisce le principali entità interagenti e la dinamica di interazione relativa all'obiettivo che si vuole raggiungere relativamente al caso di studio preso in esame.

7.2.1 Architettura generale

Dopo aver definito lo scenario specifico da considerare è stato possibile definire un'architettura generica che permettesse di strutturare il sistema proposto. In particolare, l'architettura è composta da tre tipologie di agente:

ServerBoot – agente posto sul nodo server incaricato di inizializzare il centro di tuple dello stesso, iniettando in esso una specifica di comportamento ReSpecT ben definita ed inizializzando le tuple necessarie.

InitAgent – agente posto su ogni dispositivo mobile incaricato di inizializzare il centro di tuple dello stesso, iniettando una specifica di comportamento ReSpecT ben definita ed inizializzando le tuple necessarie.

PeerAgent – agente posto su ogni dispositivo mobile, affiancato all'*InitAgent*, incaricato di attendere il comando di connessione, pervenuto al raggiungimento di una specifica posizione e ricevuto il quale deve registrarsi al server, recuperare la lista dei vicini e notificare loro la posizione organizzativa del nodo TuCSoN in esecuzione sul dispositivo mobile.

Gli agenti *ServerBoot* e *InitAgent*, vengono avviati contestualmente alla fase di configurazione del sistema e, dopo aver eseguito le operazioni a loro assegnate, terminano immediatamente la loro esecuzione. L'agente *PeerAgent* invece, una volta avviato, effettuata la connessione al server e notificata la posizione organizzativa del nodo a tutti i vicini, resta in attesa dell'arrivo di altri agenti.

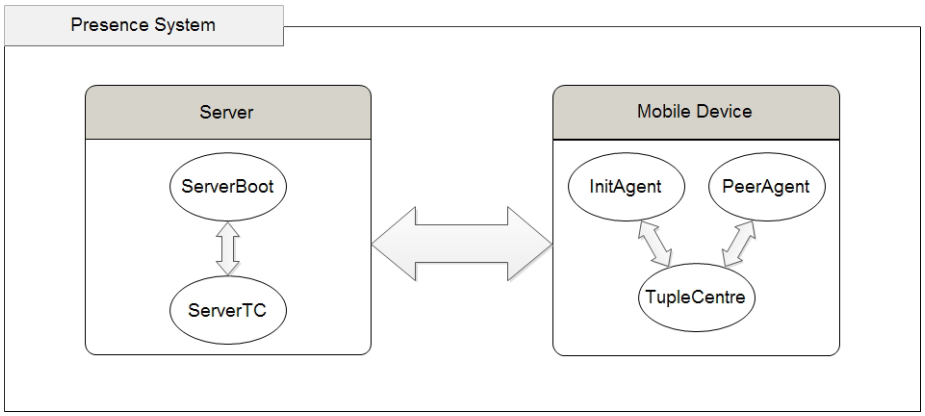


Figura 7.2: Architettura

In Figura 7.2 si può osservare la generica architettura appena descritta e si può notare che sul dispositivo, ad esempio un computer desktop, che ospita il server, è presente un centro di tuple all'interno del quale l'agente *ServerBoot* inietta la specifica di comportamento ReSpecT necessaria per elaborare le richieste di connessione da parte dei dispositivi mobili che si collegano al sistema, memorizzando i parametri di connessione a tali dispositivi in una lista recuperabile da parte degli agenti che la richiedono.

Per quanto riguarda il dispositivo mobile, anche in questo caso si può notare la presenza di un centro di tuple all'interno del quale l'agente *InitAgent* inizializza le tuple necessarie ed inietta la specifica di comportamento ReSpecT necessaria per garantire la corretta interazione con il server e gli altri agenti.

7.2.2 Interazione

Allo scopo di mostrare in maniera chiara l'interazione tra le entità definite nella sottosezione precedente, l'architettura in Figura 7.2 è stata modificata e resa più dettagliata, aggiungendo la specifica generica della dinamica di interazione che verrà poi descritta in dettaglio nella prossima sezione. La parte di architettura

relativa ai dispositivi mobili è stata duplicata al fine di differenziare tra i dispositivi già collegati al sistema e quelli che effettuano la registrazione e quindi per mostrare come avvengono le notifiche relative alla presenza di un nuovo agente e alla sua posizione organizzativa (Figura 7.3).

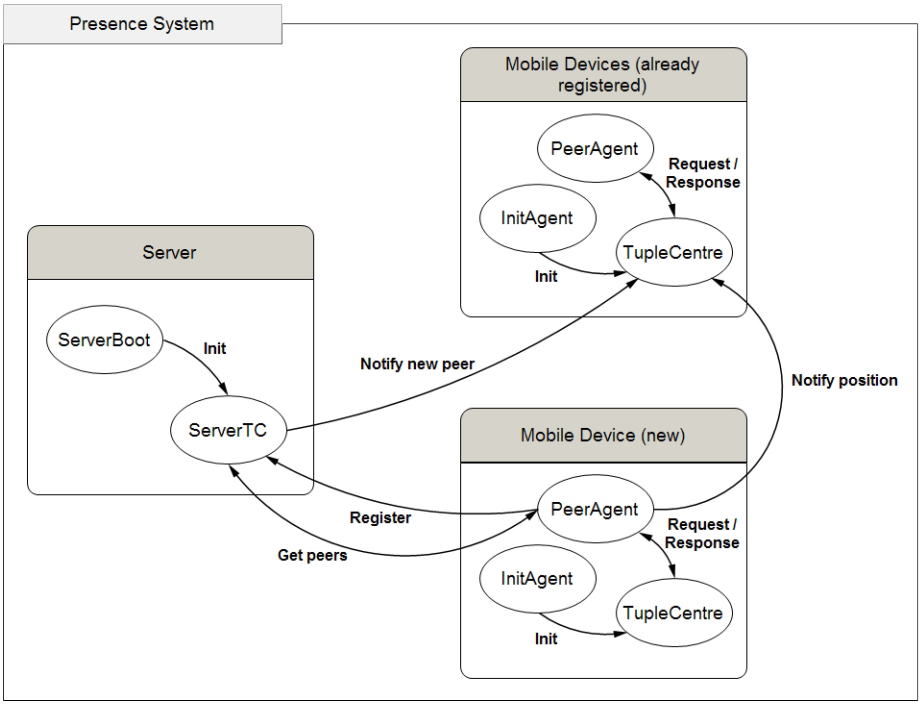


Figura 7.3: Interazione

Supponendo il sistema già avviato ed inizializzato e che vi sia un dispositivo mobile connesso al sistema e registrato sul server, il *PeerAgent*, in esecuzione su tale dispositivo, sarà in attesa di una tupla rappresentante la notifica di presenza relativa ad un nuovo vicino (*peer*).

Nel momento in cui un secondo dispositivo mobile raggiunge una certa posizione, il *PeerAgent* in esecuzione su di esso effettua la registrazione al server (*Register*) specificando in una tupla il suo nome, quello del centro di tuple associato, l'indirizzo IP e la porta del nodo TuCSoN in esecuzione sul dispositivo stesso. Ricevuta tale tupla, il server si occupa di notificare a tutti i vicini (in questo caso uno solo), contenuti in una lista aggiornata dinamicamente, la presenza di un nuovo *peer* (*Notify new peer*), inserendo una tupla direttamente nel suo centro di tuple. Il primo agente, che è bloccato in attesa di questa tupla, la recupera, elabora l'evento e attende la ricezione di un'altra tupla rappresentante la posizione organizzativa del nuovo *peer*.

Parallelamente, l'agente che ha appena effettuato la connessione, si occupa di recuperare la lista dei vicini richiedendola direttamente al centro di tuple del server (*Get peers*), dopodiché, iterando su tale lista, notifica ad essi (in questo caso uno solo) la propria posizione organizzativa (*Notify position*), inserendo una tupla direttamente nel relativo centro di tuple. Il primo agente, che è bloccato

in attesa di questa seconda tupla, la recupera, elabora l'evento e itera nel suo comportamento attendendo l'arrivo di altri *peers*.

Anche l'agente che ha appena effettuato la connessione, una volta notificata la propria posizione organizzativa ai vicini, si pone in attesa dell'arrivo di altri *peers*, uniformando il suo comportamento a quello dell'agente già presente nel sistema prima di lui.

7.2.3 La posizione organizzativa

Poiché nell'infrastruttura TuCSoN non è presente una meccanica di inserimento delle posizioni organizzative e dal servizio di geolocalizzazione non è chiaramente possibile venire a conoscenza, ad esempio, dell'ufficio in cui il nodo si trova, l'unico approccio possibile per notificare la posizione organizzativa corretta consiste nel lasciare al sistemista (colui che utilizza il middleware) il compito di impostare la natura organizzativa dello spazio che il livello applicativo (nel caso specifico, gli impiegati) andrà a sottintendere.

Per fare questo si è deciso di incaricare l'agente *InitAgent* di inserire, all'interno del centro di tuple del dispositivo mobile, una lista contenente le associazioni tra la posizione organizzativa e quella espressa in coordinate GPS (recuperata dal servizio di geolocalizzazione). Quando il dispositivo termina un movimento deve essere generato un evento (`to`) che causa l'innesco di una specifica reazione, all'interno della quale viene aggiornata la posizione organizzativa solo nel caso in cui il dispositivo si trovi nelle vicinanze di una delle posizioni presenti nella lista delle associazioni. Viene dunque inserita, e ad ogni movimento mantenuta aggiornata, una nuova tupla rappresentate la posizione organizzativa corrente del dispositivo. Tale comportamento deve essere definito all'interno della specifica ReSpecT che lo stesso *InitAgent* si occupa di iniettare all'interno del centro di tuple del dispositivo in fase di inizializzazione.

Quindi, con riferimento all'interazione descritta nella sottosezione precedente, l'agente che deve notificare la propria posizione organizzativa ai vicini dovrà recuperarla direttamente dal centro di tuple una volta che essa è stata correttamente impostata.

7.3 Implementazione

Definite le entità facenti parte del sistema da realizzare e le interazioni che devono sussistere tra esse, è stato possibile attuare l'implementazione concreta del *Presence System*, procedendo in due fasi. Nella prima vengono definite l'entità *ServerBoot* e la specifica di comportamento ReSpecT del server, nella seconda invece vengono definite le entità *InitAgent* e *PeerAgent*, nonché la specifica di comportamento ReSpecT del centro di tuple posto sul dispositivo mobile.

7.3.1 Server

Per l'implementazione del server è stata sviluppata una semplice applicazione Java da eseguire su un computer (desktop o notebook) designato come server. Tale applicazione risulta piuttosto banale e composta essenzialmente da un'entità *ServerBoot* e da un file contenente la specifica di comportamento del nodo server.

ServerBoot

L'entità *ServerBoot*, come già accennato nella Sottosezione 7.2.1, rappresenta un agente posto sul nodo server ed incaricato semplicemente di inizializzare il centro di tuple, dopodiché termina immediatamente la sua esecuzione.

In particolare, questa entità si occupa di iniettare la specifica di comportamento ReSpecT alla quale il nodo server deve attenersi ed inizializzare una tupla, espressa nella forma `neighbours(L)`, rappresentante la lista di tutti i dispositivi collegati al sistema. Tale lista conterrà tuple del tipo `peer(Name,Tc,Ip,Port)` che rappresentano i dispositivi collegati e contengono le informazioni relative al *peer*: il suo nome ed i parametri di connessione al centro di tuple ad esso associato.

Listing 7.1: ServerBoot.java

```
[...]

@Override
protected void main() {
  final SynchACC acc = this.getContext();
  try {
    final TucsonTupleCentreId tc =
        new TucsonTupleCentreId("peersLocationsTC",
                                this.ip, this.port);

    this.say("Injecting 'server_spec' ReSpecT specification
              in tc < " + tc.toString() + " >...");

    acc.out(tc, LogicTuple.parse("neighbours([])"), null);
    acc.setS(
        tc,
        Utils.fileToString("src/it/unibo/tucson/casestudy
                           /config/server_spec.rsp"),
        null);

    acc.exit();
  } catch (final Exception e) {
    e.printStackTrace();
  }
}

[...]
```

Specifica di comportamento

La specifica di comportamento che viene iniettata nel centro di tuple del nodo server si può suddividere logicamente in due parti.

Nella prima parte viene specificato il comportamento del server durante la registrazione di un nuovo *peer*. Quando un nuovo dispositivo raggiunge una specifica posizione, effettua la registrazione al server inserendo nel centro di tuple di quest'ultimo una tupla del tipo `peer(Name,Tc,Ip,Port)`, contenente le informazioni su nome e parametri di connessione al centro di tuple ad esso asso-

ciato. All'inserimento di tale tupla, sul nodo server viene innescata la seguente reazione:

Listing 7.2: server_spec.rsp

```
reaction(
  out(peer(Name,Tc,Ip,Port)),
  (completion),
  (
    Nbr = peer(Name,Tc,Ip,Port),
    in(neighbours(Nbrs)),
    out(neighbours([Nbr|Nbrs])),
    out(forward(Nbrs,Nbr))
  )
).
```

Innanzitutto viene costruito un termine (*Nbr*) rappresentante la tupla ricevuta in ingresso, dopodiché viene aggiornata la lista dei vicini (`neighbours(L)`), inserendo in essa il termine appena creato. Infine, allo scopo di inoltrare a tutti i vicini la notifica di presenza relativa ad un nuovo *peer*, viene inserita una tupla di forwarding, espressa nella forma `forward(Nbrs,Nbr)`, specificando la lista dei vicini precedente e il termine relativo al nuovo *peer*. Tale operazione porta all'innesco del seguente gruppo di reazioni:

Listing 7.3: server_spec.rsp

```
% 2) Forwards to each neighbour

  reaction(
    out(forward(Nbrs,Nbr)),
    (internal),
    (
      in(forward(Nbrs,Nbr))
    )
  ).

  reaction(
    out(forward([peer(Name,Tc,Ip,Port)|Nbrs],Nbr)),
    (internal),
    (
      Tc@Ip:Port ? out(newPeer(Nbr)),      % Forward
      out(forward(Nbrs,Nbr))               % Iterate
    )
  ).
```

La prima reazione esegue semplicemente il recupero della tupla di forwarding (`forward(Nbrs,Nbr)`), appena inserita, allo scopo di mantenere pulito il centro di tuple. La seconda reazione invece, innescata solo se la lista specificata come primo argomento non è vuota, effettua l'inoltro di una tupla `newPeer(Nbr)` verso il primo centro di tuple della lista, specificando il termine relativo al nuovo *peer* costruito nella reazione mostrata inizialmente. Fatto questo, viene inserita nuovamente la tupla di forwarding allo scopo di iterare sulla coda della lista ed inoltrare la notifica a tutti i vicini presenti. Tale processo ovviamente termina nel momento in cui la lista specificata come primo argomento risulta vuota.

Nella seconda parte di questa specifica, viene definito il comportamento del server durante la de-registrazione di un *peer* che per vari motivi decide di scollegarsi dal sistema. Quando questo accade, il *peer* deve rimuovere la tupla precedentemente inserita per la registrazione, operazione che causa l'innesco della reazione seguente:

Listing 7.4: server_spec.rsp

```
reaction(
  in(peer(Name,Tc,Ip,Port)),
  (completion),
  (
    in(neighbours(Nbrs)),
    delete(peer(Name,Tc,Ip,Port),Nbrs,Nbrs2),
    out(neighbours(Nbrs2))
  )
).
```

Innanzitutto viene recuperata la lista dei vicini tramite la primitiva `in`, allo scopo di bloccare qualsiasi agente che necessiti di leggere la lista e gestendo quindi la problematica di coordinazione *readers-writers*, dopodiché viene invocato il predicato built-in di Prolog `delete(Element,ListSource,ListDest)`, utilizzato per eliminare dalla lista dei vicini il termine relativo al *peer* che ha appena effettuato la de-registrazione. Infine, viene inserita nuovamente la lista dei vicini aggiornata.

7.3.2 Dispositivo mobile

Sul dispositivo mobile sono state definite due entità, *InitAgent* e *PeerAgent*, necessarie per garantire la corretta interazione tra i componenti del sistema.

InitAgent

L'entità *InitAgent*, come già accennato nella Sottosezione 7.2.1, rappresenta un agente posto su ogni dispositivo mobile ed incaricato di inizializzare il centro di tuple, dopodiché termina immediatamente la sua esecuzione.

In particolare, questa entità si occupa di iniettare la specifica di comportamento ReSpecT alla quale il nodo su dispositivo mobile deve attenersi ed inizializzare il centro di tuple, inserendo una tupla `init(OrgL)`. Quest'ultima causa l'innesco di una reazione nel cui body vengono inizializzate le tuple necessarie (la descrizione dettagliata di questa reazione viene mostrata nelle prossime sottosezioni). Fatto questo, l'*InitAgent* attende l'inserimento di una tupla `done`, che indica il completamento dell'inizializzazione, dopodiché termina l'esecuzione.

Listing 7.5: InitAgent.java

```
[...]

@Override
protected void main() {
  final SynchACC acc = this.getContext();
  try {
    this.tcId = new TucsonTupleCentreId("peerAgentTC",
                                        this.ip, this.port);
```

```
    this.say("Injecting 'peer_spec' ReSpecT specification
              in tc < " + this.tcId.toString() + " >...");
    this.acc.setS(
        tcId,
        Utils.fileToString("src/it/unibo/tucson/casestudy
                           /config/peer_spec.rsp"),
        null);

    this.say("Initializing tuples in tc
             < " + this.tcId.toString() + " >...");
    ITucsonOperation op = this.acc.out(
        this.tcId,
        LogicTuple.parse("init([" +
                "orgPosition(coords(44.134029,12.058843)," +
                             "'office1')," +
                "orgPosition(coords(44.134025,12.058376)," +
                             "'office2')," +
                "orgPosition(coords(44.134196,12.058443)," +
                             "'office3')"   + "])"),
        null);
    if(!op.isResultSuccess()) {
      this.say("problems during initialization");
    }

    op =
      this.acc.in(this.tcId, LogicTuple.parse("done"), null);
    if(op.isResultSuccess()) {
      this.say("Tuples initialized successfully");
    } else {
      this.say("problems during initialization");
    }

    acc.exit();
  } catch (final Exception e) {
    e.printStackTrace();
  }
}

[...]
```

Come si nota dallo stralcio di codice riportato sopra, all'interno della tupla `init(OrgL)` viene inserita una lista contenente tuple di tipo `orgPosition(Ph, Org)`. Tali tuple rappresentano le associazioni tra posizione fisica assoluta, espressa nella forma `coords(Lat,Lng)`, e posizione organizzativa espressa come stringa. Questo è necessario in quanto, come descritto nella Sottosezione 7.2.3, dev'essere il sistemista ad impostare la natura organizzativa dello spazio che il livello applicativo andrà a sottintendere, poiché il medium di coordinazione non può conoscerla a priori.

Specifica di comportamento

Anche in questo caso, la specifica di comportamento che viene iniettata nel centro di tuple si può suddividere logicamente in due parti. Nella prima parte

viene specificato il comportamento del nodo durante la fase di inizializzazione. Quando l'entità *InitAgent* viene eseguita, dopo aver iniettato la specifica di comportamento, inserisce la tupla `init(OrgL)` per inizializzare le tuple necessarie all'interazione tra i componenti del sistema. All'inserimento di tale tupla viene innescata la seguente reazione:

Listing 7.6: peer_spec.rsp

```
% 1) Initializes tuples in the tuple centre

  reaction(
    out(init(OrgL)),true,
    (
        in(init(OrgL)),
        current_place(ip,P),
        out(nodePosition(ip,P)),
        out(currentOrgPosition('NA')),
        out(orgPositionList(OrgL)),
        out(done)
    )
  ).
```

Innanzitutto viene rimossa la tupla `init(OrgL)` allo scopo di mantenere pulito il centro di tuple, dopodiché vengono effettuate le seguenti operazioni:

- recupero dell'indirizzo IP corrente del nodo, in esecuzione sul dispositivo mobile, tramite il predicato di osservazione spaziale `current_place(@S,?P)`, specificando come primo argomento l'atomo `ip`.
- inserimento di una tupla `nodePosition(ip,P)` rappresentate l'indirizzo IP recuperato con l'operazione precedente.
- inizializzazione di una tupla rappresentante la posizione organizzativa corrente, espressa nella forma `currentOrgPosition(Org)`.
- inserimento di una nuova tupla `orgPositionList(OrgL)`, rappresentate la lista delle associazioni tra posizione fisica assoluta ed organizzativa, specificando in ingresso la lista contenuta nella tupla `init`.
- inserimento della tupla `done` per notificare all'*InitAgent* che l'inizializzazione è stata completata con successo.

La seconda parte della specifica di comportamento viene mostrata in dettaglio dopo aver descritto l'implementazione dell'entità *PeerAgent*.

PeerAgent

L'entità *PeerAgent*, come già accennato nella Sottosezione 7.2.1, rappresenta un agente posto su ogni dispositivo mobile, affiancato all'*InitAgent*, ed incaricato di attendere il comando di connessione, pervenuto al raggiungimento di una specifica posizione e ricevuto il quale deve registrarsi al server, recuperare la lista dei vicini e notificare loro la posizione organizzativa corrente del nodo TuCSoN in esecuzione sul dispositivo mobile.

In particolare, questa entità, eseguita solo successivamente alla terminazione dell'*InitAgent*, attende inizialmente il comando di connessione rappresentato

dalla tupla connect, inserita nel momento in cui il dispositivo mobile termina il movimento in una posizione organizzativa conosciuta.

Listing 7.7: PeerAgent.java

```
[...]

  this.say("Waiting for connect command...");
  op =
    this.acc.in(this.tcId, LogicTuple.parse("connect"),null);
  if(op.isResultSuccess()) {
    this.say("Connect command received");
  } else {
    this.say("problems during connection");
  }

[...]
```

Recuperata la tupla connect, il *PeerAgent* si occupa leggere il contenuto della tupla nodePosition, inserita nel centro di tuple in fase di inizializzazione, allo scopo di effettuare la registrazione al server specificato l'indirizzo IP del nodo in esecuzione sul dispositivo mobile. Fatto questo infatti, il *PeerAgent* si occupa di inserire, direttamente nel centro di tuple del nodo server, una tupla del tipo peer(Name,Tc,Ip,Port), specificando: il suo nome, quello del centro di tuple a lui associato ed indirizzo IP e porta del nodo TuCSoN.

Listing 7.8: PeerAgent.java

```
[...]

  this.say("Retrieving my node ip position...");
  op = this.acc.rd(this.tcId,
                   LogicTuple.parse("nodePosition(ip,P)"),
                   null);

  String nodeIp = "";
  if(op.isResultSuccess()) {
    LogicTuple res = op.getLogicTupleResult();
    nodeIp = NetworkUtils.getIp(
                Tools.removeApices(res.getArg(1).toString())
             );
  } else {
    this.say("problems during node ip position retrieval");
  }

  acc.out(servertc,
          LogicTuple.parse(
              "peer(" + this.myName()       + "," +
                        tcId.getName()      + "," +
                        "'" + nodeIp + "'" + "," +
                        tcId.getPort()      + ")" ),
            null);

[...]
```

A questo punto il *PeerAgent* si occupa di recuperare la lista di tutti i vicini tramite la primitiva di coordinazione `rd`. Fatto questo, effettua un controllo sul risultato di tale operazione: se la lista non è vuota oppure contiene un elemento che corrisponde all'agente stesso, significa che non sono presenti vicini a cui inviare la posizione organizzativa. In caso contrario, viene richiamato il metodo `notifyOrgPosToAll(List<Term>)`, tramite il quale tale posizione viene notificata a tutti i vicini.

Listing 7.9: PeerAgent.java

```
[...]

  this.say("Getting peers...");
  op = acc.rd(servertc,
              LogicTuple.parse("neighbours(Nbrs)"),
              null);

  LogicTuple res = op.getLogicTupleResult();
  List<Term> peers = res.getArg(0).toList();

  if (peers.size() > 0 &&
      !( peers.size() == 1
         && ( (Struct) peers.get(0).getTerm() ).getArg(0)
             .toString().equals(this.myName())
       )
  ) {
    this.say("Notifing my organisational position to
              others...");
    this.notifyOrgPosToAll(peers);
  }

[...]
```

All'interno del metodo `notifyOrgPosToAll(List<Term>)` viene effettuata l'iterazione sulla lista dei vicini ricevuta in ingresso, recuperando le tuple `peer(Name,TcId,Ip,Port)` una per volta e, nel caso in cui l'argomento *Name* corrente non corrisponda al nome dell'agente stesso, viene costruito un nuovo `TucsonTupleCentreId` partendo dagli altri argomenti della tupla `peer` corrente. Fatto questo, viene inserita una tupla `currentPosition(Name,OrgPos)` direttamente all'interno del centro di tuple identificato dal `TucsonTupleCentreId` appena creato. In questo modo gli agenti, in esecuzione su altri dispositivi mobili ed in attesa di tale tupla, sono in grado di recuperare la posizione organizzativa del nuovo *peer* direttamente dal loro centro di tuple.

Listing 7.10: PeerAgent.java

```
[...]

private void notifyOrgPosToAll(List<Term> peers)
  throws [...] {

  String orgPos = this.getOrgPos();

  for (Term peer : peers) { // peer(Name, TcId, Ip, Port)
    Struct currPeer = (Struct)peer.getTerm();
```

```
    if(!currPeer.getArg(0).toString().equals(this.myName())){
      TucsonTupleCentreId target = new TucsonTupleCentreId(
            currPeer.getArg(1).toString(),
            currPeer.getArg(2).toString(),
            currPeer.getArg(3).toString());
      this.acc.out(target,
                   LogicTuple.parse(
                    "currentPosition("+ this.myName() + "," +
                                        orgPos + ")"),
                   null);
    }
  }
}

[...]
```

Come si nota dallo stralcio di codice riportato sopra, il *PeerAgent* recupera la posizione organizzativa del nodo invocando il metodo `getOrgPos()`, definito proprio a questo scopo. All'interno di tale metodo viene semplicemente letta, tramite la primitiva di coordinazione `rd`, la tupla `currentOrgPosition(OrgPos)` che è stata inserita nel centro di tuple in fase di inizializzazione ed aggiornata in automatico alla terminazione di ogni movimento. Le modalità con cui questa tupla viene mantenuta aggiornata verranno descritte in dettaglio nella prossima sottosezione.

Listing 7.11: PeerAgent.java

```
[...]

private String getOrgPos()
  throws [...] {

  ITucsonOperation op =
    this.acc.rd(this.tcId,
                LogicTuple.parse("currentOrgPosition(Org)"),
                null);
  String orgPos = "";
  if(op.isResultSuccess()) {
    orgPos = op.getLogicTupleResult().getArg(0).toString();
    if(orgPos.equals("NA")) {
      this.say("The node is not at a known org position!");
    } else {
      this.say("My current org position is: " + orgPos);
      }
  } else {
    this.say("problems during current org position
              retrieval");
  }

  return orgPos;
}

[...]
```

A prescindere che vi siano, oppure no, vicini a cui notificare la posizione organizzativa, il *PeerAgent* a questo punto entra in un loop, all'interno del quale attende ciclicamente l'arrivo di nuovi vicini e della notifica relativa alla loro posizione organizzativa.

Listing 7.12: PeerAgent.java

```
[...]

  while (true) {
    this.say("Waiting for new neighbours...");
    op = this.acc.in(this.tcId,
                     LogicTuple.parse("newPeer(N)"),
                     null);
    if (op.isResultSuccess()) {
      res = op.getLogicTupleResult();
    } else {
      this.say("problems during new peer retrieval");
          break;
    }

    this.say("New neighbour is arrived: " + res.getArg(0));

    this.say("Waiting for organisational position from
              the new neighbour...");
    op = this.acc.in(this.tcId,
                     LogicTuple.parse(
                      "currentPosition(Name,OrgPos)"),
                     null);
    if (op.isResultSuccess()) {
      res = op.getLogicTupleResult();
    } else {
      this.say("problems during organisational position
                retrieval");
      break;
    }

    this.say("The new neighbour " + res.getArg(0) +
             " is at " + res.getArg(1));
  }

[...]
```

Specifica di comportamento

Proseguendo la descrizione della specifica di comportamento del nodo in esecuzione su dispositivo mobile, la seconda parte di tale specifica definisce il comportamento del nodo in fase di esecuzione.

Come descritto in precedenza l'entità *PeerAgent* resta in attesa del comando di connessione, rappresentato dalla tupla `connect`. Quest'ultima deve essere inserita nel centro di tuple solo nel momento in cui il dispositivo mobile termina il movimento in una posizione organizzativa nota al sistema. A tale scopo è stata sfruttata la capacità della macchina virtuale (a valle dell'estensione spaziale

effettuata nel corso di questo elaborato) di generare eventi spaziali in risposta all'avvio o alla terminazione di un movimento. Nello specifico, viene generato un evento `to(?S,?P)` ogni volta che il dispositivo mobile che ospita il nodo TuCSoN termina un movimento.

Per sfruttare questa potenzialità, è stata definita una specifica reazione che viene innescata ogni volta che l'evento di tipo `to` viene generato. Tale reazione, a sua volta, causa l'innesco di altre reazioni definite al duplice scopo di inserire il comando di connessione (`connect`) solo quando viene raggiunta una posizione nota al sistema ed aggiornare la tupla `currentOrgPosition(Org)` rappresentante la posizione organizzativa corrente. Questo gruppo di reazioni è riportato di seguito:

Listing 7.13: peer_spec.rsp

```
% 2) When a to(ph,P) motion event is generated

  % 2.1)
  reaction(
    to(ph,P),
    true,
    (
      rd(orgPositionList(OrgL)),
      out(checkOrgPosition(OrgL))
     )
   ).

  % 2.2)
  reaction(
    out(checkOrgPosition(OrgL)),
    (internal),
    (
      in(checkOrgPosition(OrgL))
    )
  ).

  % 2.3)
  reaction(
    out(checkOrgPosition([orgPosition(Pos,Org)|Orgs])),
    (internal),
    (
      out(checkOrgPosition(Orgs))
    )
  ).

  % 2.4)
  reaction(
    out(checkOrgPosition([orgPosition(Pos,Org)|Orgs])),
    (internal, near(ph,Pos,2)),
    (
      no(connect),
      out(connect)
    )
  ).
```

```
% 2.5)
reaction(
  out(checkOrgPosition([orgPosition(Pos,Org)|Orgs])),
  (internal, near(ph,Pos,2)),
  (
    in(currentOrgPosition(OldOrg)),
    out(currentOrgPosition(Org))
  )
).
```

Innanzitutto, quando termina un qualsiasi movimento e viene generato un evento `to`, si innesca la reazione `2.1`, nel cui body viene letta la lista delle posizioni organizzative note ed inserita una tupla di checking espressa nella forma `checkOrgPosition(OrgL)`. Tale operazione causa l'innesco delle reazioni `2.2`, `2.3`, `2.4` e `2.5`.

Più nello specifico:

- la reazione `2.2` si occupa semplicemente di rimuovere la tupla di checking allo scopo di mantenere pulito il centro di tuple.
- la reazione `2.3`, innescata solo se la lista in ingresso risulta non vuota, si occupa di inserire nuovamente la tupla di checking specificando sempre la lista delle posizioni organizzative note ma privata, ad ogni iterazione, del primo elemento. Questo viene effettuato allo scopo di iterare sull'intera lista fino a che essa non risulta vuota.
- la reazione `2.4` si occupa di controllare se il nodo in esecuzione sul dispositivo si trova nelle immediate vicinanze della prima posizione organizzativa contenuta nella lista di ingresso. Questo controllo viene effettuato tramite l'invocazione della guardia spaziale `near(@S,@P,@R)`, specificando in ingresso l'atomo `ph`, la posizione fisica assoluta inserita come primo argomento nella tupla `orgPosition(Pos,Org)` e recuperata dalla lista in ingresso e il raggio entro cui il nodo deve trovarsi. Se la guardia viene valutata con successo, la reazione si occupa di inserire la tupla `connect`, rappresentate il comando di connessione, che il *PeerAgent* attende per continuare la sua esecuzione.
- la reazione `2.5` si occupa di effettuare lo stesso controllo della precedente ma, se la guardia viene valutata con successo, questa reazione si occupa di aggiornare la tupla `currentOrgPosition(Org)`. Questo risulta necessario al fine di mantenere aggiornata l'informazione sulla posizione organizzativa corrente.

Le reazioni `2.4` e `2.5` devono essere obbligatoriamente mantenute separate in quando, mentre la reazione `2.5`, allo scopo di mantenere aggiornata la posizione organizzativa corrente, deve venire innescata ogni volta che il dispositivo termina un movimento, la reazione `2.4`, successivamente all'inserimento della tupla `connect`, non risulta più necessaria. In particolare, quest'ultima viene direttamente rimossa, tramite la primitiva `in_s`, dall'entità *PeerAgent* non appena essa ha ricevuto il comando di connessione.

Listing 7.14: PeerAgent.java

```
[...]

  this.say("Waiting for connect command...");
  op =
    this.acc.in(this.tcId, LogicTuple.parse("connect"),null);
  if(op.isResultSuccess()) {
    this.say("Connect command received");
  } else {
    this.say("problems during connection");
  }

  this.acc.inS(
    this.tcId,
    LogicTuple.parse(
        "out(checkOrgPosition([orgPosition(Pos,Org)|Orgs]))"
    ),
    LogicTuple.parse("( internal, at(ph,Pos) )"),
    LogicTuple.parse("( nop(connect), out(connect) )"),
    null);

[...]
```

L'implementazione mostrata fino ad ora rappresenta il generico comportamento delle entità che fanno parte del sistema *Presence System* e ne è stato verificato il funzionamento sviluppando semplici applicazioni Java che simulassero la presenza del dispositivo mobile e del servizio di geolocalizzazione. Nella prossima sezione verrà descritta l'implementazione specifica realizzata su piattaforma *Android*.

7.4 Estensione dell'applicazione Android

Allo scopo di testare il caso di studio nella sua completezza, si è deciso ancora una volta di sfruttare la piattaforma *Android*. In particolare è stata estesa l'applicazione descritta nel Capitolo 6, aggiungendo una nuova sezione, denominata *Case Study*, tramite la quale è possibile avviare un *PeerAgent* direttamente su dispositivo mobile.

7.4.1 Case Study

In questa sezione è possibile inizializzare il centro di tuple, con specifica e tuple relative al caso di studio, ed avviare un *peer* che si occupa di registrarsi al server ed inviare a tutti i vicini la propria posizione organizzativa corrente.

In particolare sono state definite due classi:

`InitTupleCentreAgent` – classe rappresentate l'entità *InitAgent* incaricata di inizializzare il centro di tuple del nodo in esecuzione sul dispositivo *Android*, iniettando in esso una specifica di comportamento ben definita ed inizializzando le tuple necessarie.

`CaseStudyAgent` – classe rappresentante l'entità di tipo *PeerAgent* incaricata di attendere il comando di connessione, pervenuto al raggiungimento di

una specifica posizione e ricevuto il quale deve registrarsi al server, recuperare la lista dei vicini e notificare loro la posizione organizzativa corrente del nodo TuCSoN in esecuzione sul dispositivo mobile.

L'implementazione di queste due classi e la specifica di comportamento iniettata nel centro di tuple risultano esattamente le stesse descritte nella sezione precedente.

Assunto che sia già presente un server in attesa di registrazioni su un computer remoto, per la corretta configurazione ed esecuzione del caso di studio l'utente deve effettuare alcune operazioni preliminari. In particolare deve:

1. impostare correttamente i parametri di connessione al server tramite le impostazioni dell'applicazione.
2. avviare un nodo TuCSoN locale dalla sezione *Tucson Node Service*.
3. avviare il servizio di geolocalizzazione dalla sezione *Geolocation*.

Effettuate queste operazioni, è possibile inizializzare il caso di studio ed avviare l'agente `CaseStudyAgent`.

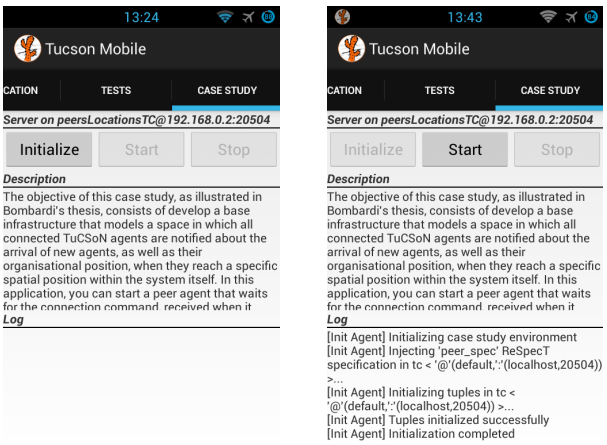


Figura 7.4: Case Study - All'avvio ed in inizializzazione

Osservando le catture, in alto vengono indicati i parametri di configurazione utilizzati per interfacciarsi con l'entità server posta in remoto su un computer (desktop o notebook), ovvero il nome del centro di tuple, l'indirizzo di rete e la porta relativi al nodo TuCSoN in esecuzione sul server stesso. É possibile impostare tutti e tre i parametri tramite le impostazioni dell'applicazione, in modo tale da poter definire agevolmente il server al quale il `CaseStudyAgent` deve registrarsi per risultare connesso al sistema.

Inoltre, dalle impostazioni dell'applicazione è possibile impostare il nome con cui l'agente `CaseStudyAgent` deve registrarsi ed essere conosciuto dagli altri agenti connessi al sistema.

Innanzitutto l'utente deve procedere all'inizializzazione del centro di tuple locale tramite la pressione del pulsante *Initialize*, azione che provoca l'avvio

dell'agente `InitTupleCentreAgent`. Questo agente si occupa di iniettare la specifica di comportamento ReSpecT (definita nel file *peer_spec.rsp* presente nella cartella *assets* dell'applicazione) all'interno del centro di tuple ed inizializzare le tuple necessarie inserendo la tupla `init(OrgL)`, specificando in ingresso la lista delle posizioni che definiscono la natura organizzativa dello spazio che il livello applicativo deve sottintendere.

Completata l'inizializzazione, l'utente può procedere con l'esecuzione dell'agente `CaseStudyAgent` premendo il pulsante *Start*. Questo agente, come già detto, attende il comando di connessione pervenuto nel momento in cui il dispositivo termina un movimento in una posizione organizzativa conosciuta, dopodiché si occupa di recuperare la lista dei vicini, interfacciandosi con il server, allo scopo di inserire nel loro centro di tuple una tupla `currentPosition(Name,OrgPos)` rappresentate la sua posizione organizzativa corrente. Fatto questo, si pone anch'esso in attesa dell'arrivo di altri *peer*.

Nel *log* vengono mostrati i vari passaggi effettuati dall'applicazione durante l'inizializzazione del centro di tuple e l'esecuzione dell'agente `CaseStudyAgent`, nonché eventuali errori che si verificano durante queste fasi.

7.4.2 Terminazione dell'esecuzione

Durante lo sviluppo della sezione *Case Study* è nata anche l'idea di permettere all'utente di terminare l'esecuzione dell'agente `CaseStudyAgent`. Per fare questo è stato inserito, direttamente nell'interfaccia utente, un pulsante *Stop*, alla pressione del quale viene avviato un agente che si occupa semplicemente di inserire le tuple necessarie ad interrompere il loop infinito all'interno del quale l'agente attende ciclicamente i nuovi *peer* e le relative posizioni organizzative. In particolare, questa nuova entità, denominata `StoppingAgent` ed implementata in una classe omonima, si occupa di inserire due tuple: `newPeer(stop)` e `currentPosition(stop,stop)`. Queste due tuple rappresentano esattamente quelle sulle quali l'agente `CaseStudyAgent` si blocca per ricevere, rispettivamente, la notifica relativa all'arrivo di un nuovo *peer* e la posizione organizzativa dello stesso.

Listing 7.15: StoppingAgent.java

```
[...]

@Override
protected void main() {
  try{
    this.acc.out(this.tcId,
                 LogicTuple.parse("newPeer(stop)"),
                 null);
    this.acc.out(this.tcId,
                 LogicTuple.parse(
                    "currentPosition(stop,stop)"
                 ), null);
  } catch (Exception e) { this.postError(e.getMessage()); }
}

[...]
```

L'inserimento di queste due tuple permette dunque di sbloccare l'agente `CaseStudyAgent`, il quale dovrà occuparsi di controllarne il contenuto per verificare che sia stato ricevuto o meno il comando di terminazione. In particolare, se il primo argomento della tupla recuperata corrisponde all'atomo `stop`, significa che l'agente deve interrompere il loop infinito e de-registrarsi dal server.

Listing 7.16: CaseStudyAgent.java

```
[...]

  while (!this.interrupted) {
    this.postLog("Waiting for new neighbours...");
    op = this.acc.in(this.tcId,
                     LogicTuple.parse("newPeer(N)"), null);
    if (op.isResultSuccess()) {
      res = op.getLogicTupleResult();
    } else {
      this.postError("problems during new peer retrieval");
      break;
    }
    if(!res.getArg(0).toString().equals("stop")){
      this.postLog("New neighbour is arrived: " +
                   res.getArg(0));
    } else {
      this.interrupted = true;
      break;
    }
    this.postLog("Waiting for organisational position from
                 the new neighbour...");
    op = this.acc.in(this.tcId,
                     LogicTuple.parse(
                     "currentPosition(Name,OrgPos)"),
                     null);
    if (op.isResultSuccess()) {
      res = op.getLogicTupleResult();
    } else {
      this.postError("problems during organisational position
                     retrieval");
      break;
    }
    if(!res.getArg(0).toString().equals("stop")){
      this.postLog("The new neighbour " + res.getArg(0) +
                   " is at " + res.getArg(1));
      this.sendNotifyCmd("showNotification",
                         res.getArg(0).toString() +
                         " is arrived at " +
                         res.getArg(1).toString() + "!" );
    } else {
      this.interrupted = true;
      break;
    }
  }

[...]
```

La de-registrazione avviene, come già accennato nella Sottosezione 7.3.1, semplicemente recuperando la tupla `peer(Name,Tc,Ip,Port)`, inserita in fase di registrazione, direttamente dal centro di tuple del nodo in esecuzione sul server. Quindi l'agente `CaseStudyAgent`, dopo aver interrotto il suo loop di attesa, si occupa di recuperare tale tupla tramite la primitiva di coordinazione `in`, dopodiché procede con la pulizia del centro di tuple locale invocando l'apposito metodo `deleteGarbageTuples()`.

Listing 7.17: CaseStudyAgent.java

```
[...]

  this.postLog("Stopping..");
  this.acc.in(this.SERVER,
        LogicTuple.parse(
            "peer(" + this.myName() + "," +
                      this.tcId.getName() + "," +
                      "'" + nodeIp + "'" + "," +
                      this.tcId.getPort() +")"),
        null);

  this.deleteGarbageTuples();
  this.sendCmd("enableInit");
  this.postLog("Stopped");

[...]
```

Listing 7.18: CaseStudyAgent.java

```
[...]

private void deleteGarbageTuples()
  throws [...] {

  this.acc.inAll(this.tcId,
                 LogicTuple.parse("newPeer(X)"),
                 null);
  this.acc.inAll(this.tcId,
                 LogicTuple.parse("currentPosition(X,X)"),
                 null);
  this.acc.inAll(this.tcId,
                 LogicTuple.parse("currentOrgPosition(X)"),
                 null);
  this.acc.in(this.tcId,
              LogicTuple.parse("orgPositionList(X)"),
              null);
  this.acc.in(this.tcId,
              LogicTuple.parse("nodePosition(S,X)"),
              null);
}

[...]
```

7.4.3 Notifiche

Allo scopo di rendere l'applicazione più interattiva al ricevimento delle notifiche relative all'arrivo di un nuovo *peer* e alla sua posizione organizzativa, si è deciso di utilizzare le *Notification* messe a disposizione dalla libreria *Android*. Questo strumento è strettamente correlato all'entità *NotificationManager* che permette di gestire la visualizzazione delle notifiche sulla barra di stato del terminale.

Non appena l'agente `CaseStudyAgent` riceve la posizione organizzativa del nuovo *peer*, viene inviato un comando (tramite lo strumento *Handler*) al processo principale che gestisce la UI di *Android*, specificando in ingresso il messaggio che si vuole mostrare nella notifica.

Listing 7.19: CaseStudyAgent.java

```
[...]

  this.sendNotifyCmd("showNotification",
            res.getArg(0).toString() + " is arrived at " +
            res.getArg(1).toString() + "!");
[...]
```

Tale comando viene ricevuto dall'*Handler*, incapsulato nella classe `CaseStudy`, che lo elabora richiamando un apposito metodo per la creazione e visualizzazione della notifica sulla barra di stato.

Listing 7.20: CaseStudy.java

```
[...]

public void showNotification(String notificationText) {
  NotificationCompat.Builder mBuilder =
        new NotificationCompat.Builder(CaseStudy.mContext)
        .setSmallIcon(R.drawable.tucson_icon)
        .setContentTitle("Case Study")
        .setContentText(notificationText);

  Intent resultIntent = new Intent(CaseStudy.mContext,
                                   MainActivity.class);
  [...intent flags setting...]
  PendingIntent resultPendingIntent =
      PendingIntent.getActivity(CaseStudy.mContext, 0,
          resultIntent, PendingIntent.FLAG_UPDATE_CURRENT);
  mBuilder.setContentIntent(resultPendingIntent);

  Notification notification = mBuilder.build();
  notification.flags |= Notification.FLAG_AUTO_CANCEL;
  notification.defaults = Notification.DEFAULT_ALL;

  NotificationManager notificationManager =
      (NotificationManager) CaseStudy.mContext
      .getSystemService(Context.NOTIFICATION_SERVICE);
  notificationManager.notify(1, notification);
}

[...]
```

7.5 Testing

Allo scopo di testare il funzionamento del sistema descritto in questo capitolo, con riferimento allo scenario mostrato nella Sottosezione 7.1.2, e poiché la disponibilità di dispositivi mobili è limitata ad un solo smartphone *Android*, si è deciso di sviluppare due semplici applicazioni Java rappresentanti due ulteriori *peer* che si connettono al sistema.

Queste due applicazioni, denominate *Office2Agent* e *Office3Agent*, non sono altro che l'unione dell'applicazione utilizzata per il testing del servizio di geolocalizzazione, mostrata nella Sezione 5.5, e di quella utilizzata per verificare il funzionamento dell'implementazione delle entità *InitAgent* e *PeerAgent*, descritta nella Sezione 7.3. In particolare, quest'ultima è composta da una singola classe, denominata `OfficeXAgent`, dove `X` rappresenta il numero dell'ufficio associato al *peer* rappresentato da essa. Essendo un'applicazione utilizzata puramente a scopo di testing, tale classe incapsula il comportamento di entrambe le entità *InitAgent* e *PeerAgent*.

Come si evince dal nome delle due applicazioni, queste rappresentano il dispositivo mobile degli impiegati associati agli uffici 2 e 3.

Per quanto riguarda il dispositivo mobile dell'impiegato associato all'ufficio 1, esso viene rappresentato dall'unico terminale disponibile, sfruttando la funzionalità apposita implementata nell'applicazione *Android* e descritta nella sezione precedente.

Infine, relativamente all'entità server, è stata sviluppata una terza applicazione Java, denominata *OfficeServer*, composta dalla sola entità *ServerBoot*, implementata nella classe omonima ed incaricata di inizializzare il centro di tuple del nodo in esecuzione sul server.

Nel seguito vengono mostrate le specifiche tecniche dei dispositivi hardware utilizzati, le configurazioni software applicate per costruire il sistema *Presence System* proposto e i vari passi effettuati per verificare il corretto funzionamento del sistema stesso.

7.5.1 Configurazione hardware

Per l'esecuzione del test sono stati utilizzati tre componenti differenti, associati ciascuno ad una specifica mansione:

Computer Desktop – rappresenta il server, sul quale è stata caricata ed eseguita l'applicazione *OfficeServer*, associata ad un nodo TuCSoN dedicato.

Computer Notebook – rappresenta i dispositivi mobili degli impiegati associati agli uffici 2 e 3. Su questo computer quindi sono state caricate ed eseguite le applicazioni *Office2Agent* e *Office3Agent*, associate a due nodi TuCSoN distinti per meglio simulare l'aspetto distribuito del sistema.

Smartphone Android – rappresenta il dispositivo mobile dell'impiegato associato all'ufficio 1. Sul terminale è stata caricata ed eseguita l'applicazione *Android* (Capitolo 6) e ne sono state sfruttate le funzionalità utili al caso di studio.

Nel seguito vengono riportate le specifiche tecniche dei componenti utilizzati.

Computer Desktop

Modello – assemblato dal proprietario

Schema madre – ASUS P5B Deluxe (Intel Broadwater P965)

Processore – QuadCore Intel Core 2 Quad Q9550, 2833 MHz

Memoria RAM – 4096 MB (DDR2-800)

Scheda grafica – AMD Radeon HD 5850 (1024 MB)

Sistema operativo – Windows 7 Professional 64 bit (Service Pack 1)

Computer Notebook

Modello – ASUS N61Jq

Schema madre – ASUS N61Jq Series (Intel Lynnfield HM55)

Processore – QuadCore Intel Core i7 720QM, 1900 MHz

Memoria RAM – 4096 MB (DDR3-1333)

Scheda grafica – AMD Radeon HD 5750 (1024 MB)

Sistema operativo – Windows 7 Professional 64 bit (Service Pack 1)

Smartphone Android

Modello – HTC Desire HD

Processore – Qualcomm Snapdragon 1024 MHz (ARMv7)

Memoria RAM – 768 MB

Scheda grafica – Ardeno 205

Sistema operativo – Android 4.2.2

Sensori – GPS, accelerometro, bussola, prossimità, luminosità

7.5.2 Configurazione software

La configurazione software è riferita all'impostazione degli indirizzi IP e delle porte dei nodi TuCSoN utilizzati e delle relative applicazioni, nonché alla corretta configurazione dei servizi di geolocalizzazione e delle posizioni organizzative conosciute.

Nella Tabella 7.1 è mostrata la configurazione effettuata su nomi, indirizzi IP e porte associati ai nodi TuCSoN e agli agenti in esecuzione nel sistema. Come già detto, sul computer notebook sono stati avviati due nodi TuCSoN differenti per simulare al meglio la caratteristica distribuita propria del sistema proposto.

Tabella 7.1: CONFIGURAZIONE SOFTWARE

Computer Desktop	
Server	Nome del centro di tuple: *peersLocationsTC* Indirizzo IP: *192.168.0.2* Porta di ascolto: *20504*

Tabella 7.1: continua nella prossima pagina

Tabella 7.1: continua dalla pagina precedente

Computer Notebook	
Office 2	Nome del centro di tuple: *default* Indirizzo IP: *192.168.0.10* Porta di ascolto: *20504* Nome del peer: *office2Agent*
Office 3	Nome del centro di tuple: *default* Indirizzo IP: *192.168.0.10* Porta di ascolto: *20505* Nome del peer: *office3Agent*
Smartphone Android	
Office 1	Nome del centro di tuple: *default* Indirizzo IP: *192.168.0.7* Porta di ascolto: *20504* Nome del peer: *office1Agent*

Per la corretta configurazione dei servizi di geolocalizzazione, è stato necessario impostare, come nodo TuCSoN al quale agganciare il servizio, lo stesso utilizzato dall'entità *PeerAgent*. In questo caso, è risultato sufficiente variare solo il numero di porta del servizio da agganciare al nodo TuCSoN rappresentate l'ufficio 3 mentre gli altri parametri, relativi a questo nodo e ai nodi degli altri due uffici, sono stati lasciati ai valori di base, ovvero *default* per il nome del centro di tuple, *localhost* per l'indirizzo IP e 20504 per il numero di porta.

Per quanto riguarda l'impostazione delle posizioni organizzative, per renderle il più possibile veritiere, è stato utilizzato il servizio *iTouchMap* (*itouchmap.com*), tramite il quale è possibile selezionare uno specifico punto sulla mappa ed ottenere in risposta le coordinate espresse in latitudine e longitudine. In particolare sono stati selezionati tre punti per simulare la posizione dei tre uffici:

- *Office 1*: latitudine `44.134029` e longitudine `12.058843`.
- *Office 2*: latitudine `44.134025` e longitudine `12.058376`.
- *Office 3*: latitudine `44.134196` e longitudine `12.058443`.

Infine, per permettere agevolmente di verificare il funzionamento del sistema, i tre servizi di geolocalizzazione sono stati programmati per notificare la terminazione di un movimento nelle immediate vicinanze delle tre posizioni organizzative prescelte. Questo ha permesso alla `RespectVM` di generare l'evento spaziale `to` necessario per permettere agli agenti di interagire con il server e con i vicini, nelle modalità descritte in questo capitolo.

7.5.3 Esecuzione

Effettuata la configurazione software appena descritta è stato possibile procedere con l'esecuzione del caso di studio allo scopo di verificarne il funzionamento. A questo scopo sono stati effettuati i seguenti passi:

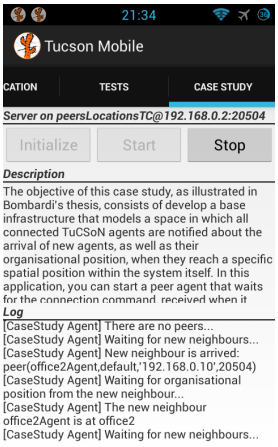
1. Avvio dei nodi TuCSoN necessari su tutti i componenti del sistema.
2. Avvio dell'applicazione *OfficeServer* sul computer desktop per l'inizializzazione del centro di tuple server.

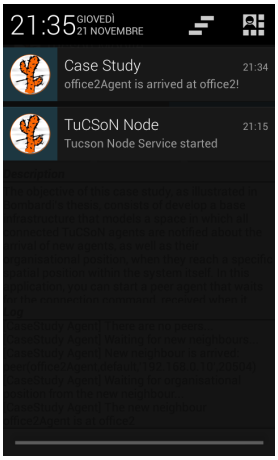
3. Avvio del servizio di geolocalizzazione sullo smartphone (*Office 1*), tramite la sezione *Geolocation* dell'applicazione *Android*.

4. Inizializzazione del centro di tuple dell'*Office 1*, ed avvio dell'entità *PeerAgent* relativa, tramite la sezione *Case Study* dell'applicazione *Android*. Questa operazione porta alla registrazione dell'agente *office1Agent* e, non essendoci altri *peer*, tale agente non notifica a nessuno la sua posizione organizzativa e si pone in attesa dell'arrivo di nuovi vicini.

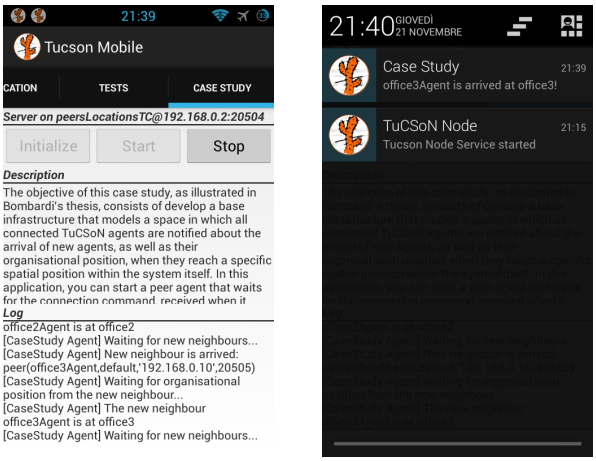
5. Avvio dei servizi di geolocalizzazione agganciati ai nodi TuCSoN, relativi ad *Office 2* e *Office 3*, in esecuzione sul computer notebook.

6. Avvio dell'applicazione *Office2Agent* sul notebook per l'inizializzazione del centro di tuple e l'avvio dell'entità *PeerAgent*, entrambi associati all'*Office 2*. Questa operazione porta alla registrazione dell'agente *office2Agent* che si occupa di notificare a *office1Agent*, già connesso al sistema, la propria posizione organizzativa. Infatti, sul terminale *Android* viene ricevuta una notifica sulla barra di stato che informa l'utente dell'arrivo di un nuovo vicino e della sua posizione organizzativa corrente.





7. Avvio dell'applicazione *Office3Agent* sul notebook per l'inizializzazione del centro di tuple e l'avvio dell'entità *PeerAgent*, entrambi associati all'*Office 3*. Questa operazione, analogamente alla precedente, porta alla registrazione dell'agente *office3Agent* che si occupa di notificare a *office1Agent* ed *office2Agent*, già connessi al sistema, la propria posizione organizzativa. Ancora una volta, sul terminale *Android* viene ricevuta una notifica sulla barra di stato che informa l'utente dell'arrivo di un nuovo vicino e della sua posizione organizzativa corrente. Parallelamente, sulla console dell'applicazione *Office2Agent*, viene visualizzata una semplice stampa della stessa informazione. Quindi entrambi gli agenti connessi al sistema, sono stati informati dell'arrivo di un nuovo vicino e della sua posizione organizzativa attuale.



8. Infine, allo scopo di verificare il corretto funzionamento anche in fase di terminazione, è stato premuto il pulsante *Stop* sull'interfaccia dell'applicazione *Android*. Tale operazione ha portato allo sblocco dell'agente *office1Agent* che si è occupato di de-registrarsi dal server e di effettuare la pulizia del centro di tuple a lui associato, dopodiché ha terminato la sua esecuzione.

Conclusioni e sviluppi futuri

In questo elaborato è stato preso in considerazione il medium e linguaggio di coordinazione ReSpecT e se ne è effettuata l'estensione con alcuni costrutti e meccanismi di base, richiesti per lo sviluppo di medium di coordinazione *space-aware*, in grado di affrontare e risolvere le problematiche legate allo spazio e alla topologia di sistemi pervasivi complessi. É stata quindi fornita una struttura efficiente per garantire comunicazione e coordinazione, basate sullo spazio, tra sistemi software distribuiti nell'ambiente ed in continuo movimento, implementando concretamente tutti i costrutti spaziali definiti dal modello *st*ReSpecT, che risulta il tassello fondamentale alla base di tutto il lavoro svolto in questo elaborato.

É stata inoltre proposta l'estensione del middleware di coordinazione, definendo un'architettura a livelli dedicata alla creazione e gestione dei servizi di geolocalizzazione. A tal proposito, sono state definite ed integrate nel sistema nuove entità, necessarie per poter agganciare l'infrastruttura TuCSoN a servizi di geolocalizzazione specifici, che possono essere definiti concretamente dal sistemista in relazione alla tecnologia utilizzata.

Grazie alla recente disponibilità dell'infrastruttura TuCSoN su dispositivi *Android*, è stato inoltre possibile agganciare al middleware un servizio di geolocalizzazione specifico che ha permesso di sperimentare concretamente la coordinazione spaziale in scenari pervasivi. Il caso di studio, in particolare, mette in evidenza una possibile applicazione dell'infrastruttura, ora *space-aware*, in un ambito realistico nel quale ciascun individuo, in possesso di un dispositivo mobile, desidera interagire con tutti gli altri individui connessi al sistema, in qualsiasi momento, in qualunque luogo si trovi e nella maniera più automatica possibile.

Tutti questi elementi, unitamente alle capacità già consolidate relative alla gestione delle problematiche temporali ed ambientali, permettono di definire l'infrastruttura TuCSoN ed il linguaggio di coordinazione ReSpecT, come completamente *situati*.

Nonostante la proprietà di *situatedness* risulti a questo punto completamente supportata dall'infrastruttura TuCSoN e gli obiettivi enunciati nell'introduzione siano stati raggiunti, vi è ancora spazio per ulteriori argomenti di approfondimento emersi durante la progettazione e lo sviluppo dei vari elementi trattati in questo elaborato.

Uno di questi è relativo alla rappresentazione della posizione organizzativa che ora assume la forma *Workspace at (map address)*. Al riguardo si dovrebbe ragionare su quali strumenti, in particolare quali API, fornire al sistemista

TuCSoN per permettergli di implementare una specifica natura organizzativa del sistema, consentendogli quindi di definire rappresentazioni ad-hoc delle posizioni organizzative che compongono lo spazio che il livello applicativo andrà a sottintendere.

Inoltre, durante la progettazione del caso di studio, è emersa la necessità di poter conoscere tutti i nodi TuCSoN presenti nella rete, allo scopo di poter comunicare e, se richiesto, coordinarsi per portare a compimento compiti specifici oppure obiettivi comuni. Dunque, invece di appoggiarsi ad un nodo TuCSoN facente funzioni di server, sarebbe necessario elaborare una metodologia efficiente di *discovery*, ad esempio tramite scambio di pacchetti UDP, che permetta ad un qualsiasi nodo di scoprire eventuali altri nodi presenti nella rete, negoziare con essi i parametri di comunicazione e, successivamente, dialogare e coordinarsi per il raggiungimento di obiettivi comuni.

Bibliografia

[1] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. *CoRR*, abs/1202.5509, 2012.

[2] M. Casadei and A. Omicini. Situated tuple centres in ReSpecT. In S. Y. Shin, S. Ossowski, R. Menezes, and M. Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, volume III, pages 1361–1368, Honolulu, Hawai'i, USA, 8–12 Mar. 2009. ACM.

[3] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agent interaction and mobility. *IEEE Transaction on Software Engineering*, 24(5):315–330, May 1998.

[4] M. B. Julien Pauty, Paul Couderc and Y. Berbers. Geo-linda: a geometry aware distributed tuple space. *Advanced Information Networking and Applications, 2007*, pages pp. 370–377, Mag. 2007. 21st International Conference on.

[5] A. N. Leontjev. *Activity, Consciousness, and Personality*. 1978.

[6] M. Mamei and F. Zambonelli. Self-star properties in complex information systems. chapter Spatial computing: the TOTA approach, pages 307–324. Springer-Verlag, Berlin, Heidelberg, 2005.

[7] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engineering Methodologies*, 18(4):15:1–15:56, July 2009.

[8] S. Mariani and A. Omicini. Promoting space-aware coordination: ReSpecT as a spatial-computing virtual machine. In J.-L. Giavitto, S. Dulman, A. Spicher, and M. Viroli, editors, *6th International Spatial Computing Workshop (SCW 2013)*, pages 29–34, AAMAS 2013, Saint Paul, Minnesota, USA, 6 May 2013. Proceedings.

[9] S. Mariani and A. Omicini. TucSoN Coordination Model & Technology. Italy, 2013.

[10] S. Mariani and A. Omicini. Space-aware coordination in ReSpecT. Torino, Italy, 2013.

[11] A. Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007. 5th International

Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 Aug. 2006. Post-proceedings.

[12] A. Omicini, A. Ricci, and M. Viroli. Time-aware coordination in ReSpecT. In J.-M. Jacquet and G. P. Picco, editors, *Coordination Models and Languages*, volume 3454 of *LNCS*, pages 268–282. Springer-Verlag, Apr. 2005. 7th International Conference (COORDINATION 2005), Namur, Belgium, 20–23 Apr. 2005. Proceedings.

[13] A. Omicini, A. Ricci, and N. Zaghini. Distributed workflow upon linkable coordination artifacts. In P. Ciancarini and H. Wiklicky, editors, *Coordination Models and Languages*, volume 4038 of *LNCS*, pages 228–246. Springer, June 2006. 8th International Conference (COORDINATION 2006), Bologna, Italy, 14–16 June 2006. Proceedings.

[14] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Sept. 1999. Special Issue: Coordination Mechanisms for Web Agents.

[15] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In M. Sirjani, editor, *Coordination Languages and Models*, volume 7274 of *LNCS*, pages 212–229. Springer-Verlag, June 2012. Proceedings of the 14th Conference of Coordination Models and Languages (Coordination 2012),Stockholm (Sweden), 14-15 June.

Ringraziamenti

E infine eccoci qua, al capitolo meno importante della tesi ma in un certo senso il più sentito di tutti ed insieme anche il più problematico. Un capitolo fatto anche per inviare messaggi alle persone che lo leggono, un capitolo che costringe chi lo scrive a pensare seriamente su quali sono state le persone più importanti in questi lunghi anni di studio, quelle che hanno fatto davvero la differenza e senza le quali oggi non sarebbe qui a scrivere queste cose.

Innanzitutto ringrazio i miei genitori e mio fratello Simone che per primi hanno creduto in me e nelle mie capacità, sostenendomi durante gli studi e consigliandomi saggiamente nei momenti di crisi, quando il mio cervello mi diceva di scappare via.

Un immenso ringraziamento è rivolto alla mia ragazza (e collega), Chiara, per avermi incoraggiato a studiare quando di voglia non ce n'era, per avermi tirato su il morale quando gli esiti degli esami non erano quelli sperati e, soprattutto, per avermi messo, quel giorno di qualche anno fa, quella benedetta pulce nell'orecchio che si è rivelata essere più saggia di quanto potesse sembrare. Senza di te, Chiara, ora di certo non sarei qui.

Un sentito ringraziamento è dedicato a tutti i miei Amici. Anche se non ne sono consapevoli, quell'unica sera a settimana nella quale si esce per un giro in centro o per un cinema, ha permesso che io staccassi la spina, non grazie alla birra (certo quella c'era) ma alla compagnia, alle chiacchiere e risate che durante i periodi d'esame sono stati un toccasana per rigenerare le mie energie.

Un sincero ringraziamento è rivolto ai miei colleghi universitari, triennali e magistrali, per quei momenti in cui la mia testa sembrava stesse per scoppiare, grazie per esservi girati verso di me dicendomi la prima cavolata che vi passava per la testa, distogliendomi da quello che in quel momento sembrava un intricato labirinto di parole e, per quei momenti di pausa tra una lezione e l'altra, grazie per avermi regalato tanti sorrisi mentre se ne dicevano di cotte e di crude.

Infine, ma assolutamente non meno importante, un ringraziamento grandissimo è rivolto al Prof. Andrea Omicini, per avermi dato l'opportunità di affrontare un argomento tanto interessante ed attuale e per la disponibilità e simpatia con la quale mi ha seguito durante il suo svolgimento, e al Dott. Stefano Mariani, per il suo sostegno tecnico, per i suoi consigli e per aver gentilmente risposto alle mie e-mail sempre ed a qualsiasi ora, fornendomi indicazioni mirate e dettagliate che mi hanno guidato fino alla fine di questo lavoro.