ALMA MATER STUDIORUM · UNIVERSITA' DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA ED ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

TuCSoN on Android: coordinazione event-driven e geolocalizzata su dispositivi mobili

Relatore:
Chiar.mo Prof.
Andrea Omicini

Presentata da:
Lorenzo Forcellini Reffi

Correlatore:
Dott. Ing. Stefano Mariani

Sessione II
Anno Accademico 2014 - 2015

...ai miei genitori, Claudio e Francesca
alla mia famiglia
a Martina

a tutti coloro ai quali non è mai importata quella vocina
che dice: "Non ce la farai"...

" Se vuoi farti buono, pratica queste tre cose e tutto andrà bene: allegria, studio, preghiera. E' questo il grande programma per vivere felice, e fare molto bene all'anima tua e degli altri "

-Don Bosco-

INTRODUZIONE

Scienza ed immaginazione si muovono su piani diversi ma sarebbe improprio negare che l'una ispira l'altra e viceversa. La forte propensione alla ricerca tecnologica, alle volte ancora generata dalla curiosità più che dal denaro, modifica il mondo a tal punto da rendere inscindibili i termini "futuro" e "tecnologia".

E' ormai normale raggiungere virtualmente qualsiasi parte del pianeta in millisecondi o guardare video in alta definizione su dispositivi portatili spessi meno di un centimetro mentre lo diventerà "dialogare" con oggetti che risponderanno alle nostre domande o agiranno per conto nostro. La diffusione di strumenti dalle dimensioni contenute, economicamente avvicinabili, perennemente connessi e dotati della più avanzata sensoristica fornisce "un'infrastruttura" completa, immediatamente fruibile, in grado di estendere infinitamente le funzionalità messe a disposizione dell'uomo. Quest'ultimo però non è il solo componente del mondo come lo vediamo, esso infatti interagisce continuamente con l'ambiente circostante che deve quindi essere altrettanto interfacciato a strumenti tecnologici.

Senza entrare nello specifico, quanto detto finora lascia trasparire l'importanza dello studio dei sistemi distribuiti dovendosi confrontare sempre di più con entità eterogenee, magari autonome, sparse nell'ambiente e necessitanti di coordinarsi fra loro e con quest'ultimo.

Definendo queste entità come "agenti", considerando quindi l'ambito dei MAS (Multi-Agent Systems), in questo lavoro di tesi viene affrontato l'aggiornamento di TuCSoN (Tuple Centre Spread over the Network), un'infrastruttura di coordinazione che mette a disposizione spazi condivisi (centri di tuple) utili agli agenti sopra citati a coordinarsi o comunicare.

A tal proposito viene in aiuto ReSpecT (Reaction Specification Tuple), un linguaggio utile a programmare i centri di tuple, in grado di conferirgli un determinato comportamento al verificarsi di un evento.

Essendo questi strumenti creati per realizzare sistemi distribuiti devono necessariamente essere progettati garantendo versatilità, abbracciando il concetto "multipiattaforma", presentando quindi, fra le altre, una soluzione "Desktop" ed una soluzione mobile "Android".

Lo sviluppo di TuCSoN, nel tempo, ha reso necessario il rilascio di nuove versioni contenenti migliorie utili alla stabilità e all'efficienza del sistema, non consentendo però l'aggiornamento parallelo su piattaforma Android. Questo ha comportato l'attuale coesistenza di due versioni TuCSoN significative.

La prima obsoleta ma:

- compatibile con l'App Android
- implementante un servizio di Geo-localizzazione
- parzialmente convertita al modello event-driven.

La seconda completamente aggiornata ma non comprendente tali caratteristiche.
L'obiettivo di questa tesi quindi consiste nell'aggiornare l'Applicazione in grado di sfruttare le caratteristiche di TuCSoN all'ultima release disponibile, aggiungere il servizio di geo-localizzazione e completare la migrazione al modello event-driven, verificando il funzionamento dei componenti attraverso l'adattamento e l'esecuzione di test specifici.
In corso d'opera, a causa delle modifiche richieste dal modello event-driven, si è intervenuto anche sulle funzionalità in grado di concretizzare la proprietà di "situatedness" mettendo mano al codice relativo alla gestione di sensori ed attuatori.

L'elaborato viene quindi suddiviso in diversi capitoli. Nel primo di questi viene presentata la teoria che descrive TuCSoN e ReSpecT andandone a specificare la struttura interna, il flusso di chiamate oltre al modello a cui si riferiscono.
Nel secondo capitolo viene presentata l'App implementante TuCSoN on Android ed i passaggi utili ad aggiornarla all'ultima release disponibile di tale infrastruttura di coordinazione, non considerando gli aspetti avanzati.
Nel terzo capitolo viene introdotta concettualmente la geo-localizzazione, per poi presentare gli strumenti messi a disposizione da Android. A questo punto si entra nella fase più importante del lavoro descrivendone la prima implementazione su TuCSoN, la migrazione all'ultima release per poi spostarsi sull'aggiornamento della geo-localizzazione Android side. Il tutto viene completato con l'adattamento ed esecuzione dei test.
Nel quarto capitolo viene descritto il modello ad eventi e si prosegue con le modifiche effettuate per conformare il progetto a tali specifiche.
Per finire nel quinto capitolo viene introdotto il concetto di situatedness e le modifiche apportate su tale proprietà, oltre ai relativi test.

CAPITOLO 1 - TuCSoN e ReSPeCT

1.1 - Il Meta-Modello Agents&Artifacts

Il meta-modello A&A interpreta i Sistemi Multi Agente (MAS) come sistemi computazionali composti da due astrazioni di base: agenti e artefatti. Gli agenti, entità attive del sistema, incapsulano il controllo e portano a termine obiettivi definendo il comportamento dell'intero sistema. Gli artefatti rappresentano invece entità passive e reattive che forniscono servizi e funzioni utili agli agenti a completare 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 ed interagire con l'ambiente.
Data la visione generale del modello si scende in dettaglio.

AGENTI

Come anticipato, una delle astrazioni fondamentali del meta-modello A&A sono gli agenti, componenti di natura pro-attiva, progettati per svolgere uno o più compiti con lo scopo di raggiungere un determinato obiettivo. Per poter portare a termine i compiti a loro assegnati gli agenti necessitano quindi di specifiche capacità di ragionamento, unite all'abilità di sfruttare gli oggetti resi a loro disponibili (artefatti). Nel contesto del sistema sono entità autonome che incapsulano il flusso di controllo e che non possono essere invocate non fornendo ne interfacce ne metodi. Saranno essi stessi ad invocare ed utilizzare gli strumenti a loro necessari incapsulando un criterio per autogovernarsi.
Inoltre gli agenti sono dinamici, possono cioè decidere per la loro esecuzione (non più passivi ne semplicemente reattivi bensì attivi) ma soprattutto impredicibili poiché si possono specificare criteri così articolati da non poterne prevedere l'esito dell'esecuzione oppure perché implementanti algoritmi di apprendimento che ne cambiano la percezione dell'ambiente che li circonda.
Essendo pensati come entità orientate al raggiungimento di un obiettivo e il cui operato determina il comportamento dell'intero sistema, gli agenti hanno anche capacità sociali ma essendo entità ingovernabili la comunicazione tra due agenti non può modificarne il comportamento in maniera assoluta. Infatti tale comunicazione è basata su scambio di dati quindi le informazioni hanno l'unico effetto di condurre l'agente a ragionare e deliberare in base agli strumenti che ha a disposizione.

Questo dialogo è un interazione fondamentale del meta-modello A&A, nel quale si considera il sistema multi-agente nel suo complesso piuttosto che i singoli agenti.

ARTEFATTI

La seconda astrazione fondamentale del meta-modello A&A sono gli artefatti. Essi sono componenti passivi e reattivi costruiti ed utilizzati dagli agenti durante le loro attività, individuali, quando vengono utilizzati da un singolo agente, oppure sociali, quando vengono sfruttati come entità di coordinazione delle interazioni per la cooperazione di più agenti.

Gli artefatti mediano da un lato le interazioni tra i componenti e l'ambiente in cui vivono, dall'altro incapsulano la porzione dell'ambiente progettata per supportare le attività dei componenti del sistema. Possono essere inoltre monitorati come parte osservabile del sistema allo scopo, ad esempio, di valutare le prestazioni complessive. E' possibile identificare anche una terza tipologia di artefatti, ovvero gli artefatti ambientali come mediatori tra l'ambiente e gli agenti del sistema.

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

- **inspectability:** gli agenti devono poter sfruttare al meglio l'artefatto quindi si deve rendere il suo comportamento visibile e osservabile.

- **controllability:** un artefatto può garantire il controllo sulla propria struttura, stato e comportamento.

- **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 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.

- **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 è uno strumento senza attività che viene progettato allo scopo di fornire funzioni aggiuntive agli agenti che lo utilizzano. Questo deve presentare quindi un'interfaccia che rende disponibili tutte le funzioni utilizzabili dagli agenti per interagire con l'ambiente o altri agenti.
Tornando in ottica MAS, a questo punto si può dichiarare che gli artefatti di coordinazione sono l'astrazione primaria per la coordinazione all'interno di un sistema complesso. Tali artefatti mediano l'interazione tra gli attori coinvolti nello stesso contesto sociale e abilitano l'interazione agente-agente e agente-ambiente.
Il meta-modello A&A ridefinisce il concetto di spazio di interazione all'interno di un MAS in modo tale che i componenti 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.

Allo scopo di gestire i problemi di coordinazione, una delle possibili alternative è rappresentata dall'utilizzo del modello di coordinazione TuCSoN e del linguaggio di coordinazione ReSpecT su cui esso si basa. Vengono introdotti quindi questi due elementi poiché sono il framework concettuale di base utilizzato in questo lavoro.

TuCSoN

TuCSoN (Tuple Centres Spread over the Network) è un modello orientato agli agenti quindi utile alla coordinazione nei Sistemi Multi-Agente. Esso fornisce spazi di tuple come astrazione principale per progettare e sviluppare artefatti di coordinazione. Contrariamente allo spazio di tuple standard, TuCSoN adotta il linguaggio di specifica ReSpecT che permette la programmazione del comportamento insito nel centro di tuple.
Gli agenti interagiscono con quest'ultimo e si coordinano tramite lo scambio di tuple attraverso un insieme di primitive Linda-like (in, rd, inp, rdp).

Le principali caratteristiche di questo approccio sono:

- **comunicazione generativa:** le informazioni inserite nel centro di tuple sono disaccoppiate dal ciclo di vita di chi le ha generate causando il disaccoppiamento spaziale e temporale degli agenti.
- **accesso associativo:** l'accesso alle informazioni è basato sul meccanismo di tuple matching, relativo alla 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.

Un sistema basato su TuCSoN è composto da 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 formato da uno spazio condiviso per la comunicazione basata su tuple (tuple space) e uno spazio di specifica che contiene la logica di comportamento.
Ogni centro di tuple viene identificato con l'ausilio di un nome:

tcname@netid:portno

Dove:

- **tcname** è il nome del centro di tuple definito come un termine Prolog di prim'ordine.

- **netid** è l'indirizzo IP del dispositivo che ospita il centro di tuple.

- **portno** è il numero di porta sulla quale il servizio di coordinazione è in ascolto.

L'interazione tra gli agenti e il centro di tuple è concretizzata mediante un linguaggio di coordinazione che permette di eseguire operazioni eseguite in due fasi. Per prima cosa un agente richiede l'esecuzione di un'operazione su uno specifico centro di tuple (*invocation*), successivamente, elaborato il risultato, esso risponde con le informazioni richieste (*completion*).
La sintassi delle operazioni sul centro di tuple è:

tcname@netid:portno?op

tramite la quale viene invocata una primitiva su un centro di tuple (locale o remoto).

PRIMITIVE

Le primitive messe a disposizione da TuCSoN 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 ne viene trovata almeno una 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, 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.

In aggiunta esistono primitive, le primitive *bulk*, che possono gestire più tuple con una singola operazione:

- 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 definire comportamenti probabilistici all'interno del meccanismo di coordinazione degli agenti.
Per concludere 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.

ReSPeCT

ReSpecT (Reaction Specification Tuples) è un linguaggio utile a programmare i centri di tuple TuCSoN.
In sintesi esso permette di definire attività computazionali, dette reazioni, all'interno dei centri di tuple le quali vengono eseguite in seguito al verificarsi di eventi associati all'interno del centro di tuple stesso. Ne risulta la separazione di due componenti, quella dichiarativa in cui si

dichiarano reazioni associate agli eventi tramite apposite tuple di specifica e quella procedurale che permette di definire reazioni sotto forma di obiettivi da raggiungere.
Si possono dunque identificare due tipi di tuple:

- **tuple ordinarie**: un termine Prolog di prim'ordine.

- **tuple di specifica:** un termine Prolog, nella forma *reaction(E, R)*. Dato un evento *Ev*, tale tupla associa un evento di comunicazione *E* ad una reazione *R*. (ReactionGoals, ovvero l'insieme di operazioni ReSpecT che devono essere eseguite in risposta agli eventi). Definiscono quindi il comportamento di uno spazio di tuple in termini di reazioni di interazione, 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 semantica transazionale e tutte le reazioni innescate da un evento di comunicazione vengono terminate prima che ne venga servito un altro. Gli agenti quindi percepiscono l'avvenuta elaborazione di un evento (e di tutte le reazioni associate) come una singola transizione.
Insieme, TuCSoN e ReSpecT, formano quindi un framework concettuale che soddisfa al meglio le necessità del *meta-modello A&A* .
Il linguaggio di coordinazione ReSpecT è stato poi esteso definendo una nuova sintassi per il meta-modello A&A introducendo la specifica delle guardie a quella che era la struttura delle reazioni. Viene aggiunto così un nuovo termine G e le tuple di specifica in A&A ReSpecT assumono la forma:

reaction(E,G,R)

Dato un evento *Ev*, tale tupla associa un evento di comunicazione rappresentato dal termine E ad una reazione R se le condizioni espresse dalla guardia G sono soddisfatte:

- **E** è l'evento al quale la reazione è associata

- **G** è la guardia associata all'evento, ovvero l'insieme di condizioni da soddisfare affinché tutte la reazioni presenti in R possano essere eseguite

- **R** è il corpo della reazione espressa sotto forma di primitive ReSpecT

Velocemente, l'implementazione di tutto questo implica la classe principale *RespectVMContext* che si occupa di fornire un'interfaccia tra TuCSoN e le operazioni ReSpecT. Questa classe è incaricata di gestire inserimento, rimozione e lettura delle tuple nel centro di tuple e 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 alla classe *Respect2PLibrary*, che rappresenta una libreria attraverso la quale è possibile definire il comportamento delle primitive ReSpecT. Questa libreria viene utilizzata lato core, all'interno della *RespectVM*.

Allo stesso modo, 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.

1.2 - RespectVM e TuCSoN: la comunicazione

Affrontate le caratteristiche di base vengono presentate le modalità di comunicazione fra le entità del sistema.
Un sistema basato su TuCSoN è costituito da un'insieme di nodi, interconnessi tramite la rete, che 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.
Esso 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, TuCSoN struttura i centri di tuple governando l'accesso associando ad ogni agente un ruolo specifico. Questo aspetto viene realizzato secondo il modello Role-Based Access Control (RBAC) che richiede la presenza di un centro di tuple adibito a contenere le regole di accesso. A tal fine, e proprio in merito alla comunicazione oggetto di questa sezione, è stato introdotto *l' Agent Coordination Context (ACC)*, 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 sia in modalità sincrona che asincrona.
Inoltre, TuCSoN fornisce un'altro ACC, Specification (Synch/Asynch)ACC, utilizzato allo scopo di permettere l'accesso alla specifica ReSpecT.

Vengono ora presentate le fasi principali della comunicazione.
Un'agente TuCSoN, posto in un nodo della rete, effettua una richiesta ad un centro di tuple TuCSoN, situato nello stesso nodo o in remoto. Quest'ultimo elabora la richiesta effettuando le relative operazioni e rispondendo al chiamante con il risultato.
Il protocollo utilizzato per stabilire la connessione tra un agente A ed uno specifico centro di tuple T, indipendentemente da dove si trovino, è basato su TCP ed è suddiviso in tre fasi, che sono:

1. A prova l'autenticazione; se ha successo, viene creata un'entità chiamata *ACCProxyAgentSide*, che ha il compito di mantenere attiva la comunicazione.

2. L'entità *ACCProxyAgentSide* comunica 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. *L'ACCProvider* analizza tipo,

contenuto e chiamante della richiesta e, nel caso tutto sia corretto, crea una *ACCProxyNodeSide* avente il compito di mantenere attiva la comunicazione con l'agente A. Viene quindi stabilito un canale di comunicazione tra *ACCProxyAgentSide* e *ACCProxyNodeSide* utile all' 'interazione tra l'agente ed il centro di tuple.

3. 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.

Viene quindi riportato il flusso dettagliato delle operazioni:

LATO NODO [1]:

`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.

Andando in dettaglio sulla macchina virtuale [1]:

`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`**.

CAPITOLO 2 - TuCSoN ON ANDROID

I capitoli finora affrontati avevano l'obiettivo di fornire la descrizione dei principali strumenti oggetto del lavoro di tesi.
Dopo aver illustrato le caratteristiche primarie di TuCSoN/ReSpecT, avendone analizzato quindi la struttura ed il comportamento, risulta opportuno completare la parte teorica attraverso una panoramica dell'applicazione Android utile ad ottenerne un implementazione mobile.

2.1 - App: Lo stato attuale

L'app, realizzata in precedenti lavori di tesi, è strutturata mediante cinque fragment:

1. **Tucson Node Service**: Fragment che permette di avviare o interrompere il nodo TuCSoN.

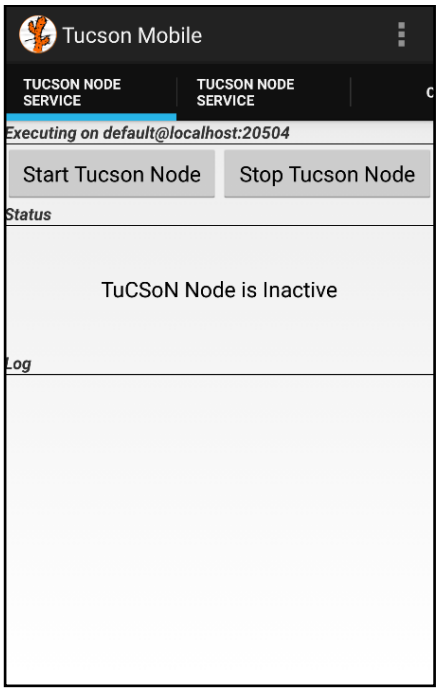
2. **CLI**: Fragment tramite il quale è possibile invocare tutte le primitive di coordinazione, richiedendone l'elaborazione al nodo TuCSoN in esecuzione sul dispositivo oppure ad uno remoto.

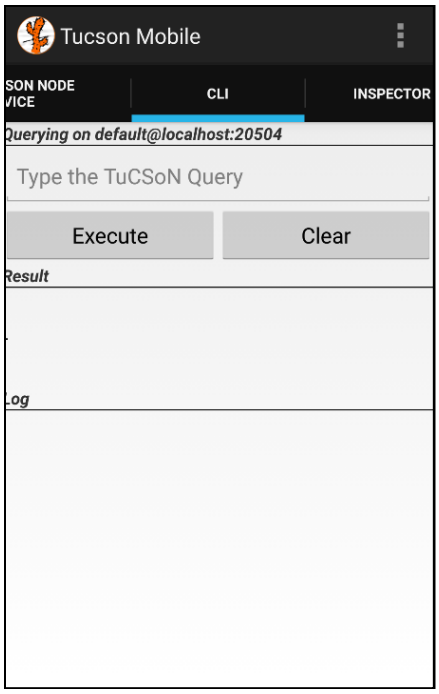
3. **Inspector**: Fragment tramite il quale è possibile avviare lo strumento Inspector, agganciandolo al nodo TuCSoN in esecuzione sul dispositivo oppure ad uno remoto in modo da monitorarlo.

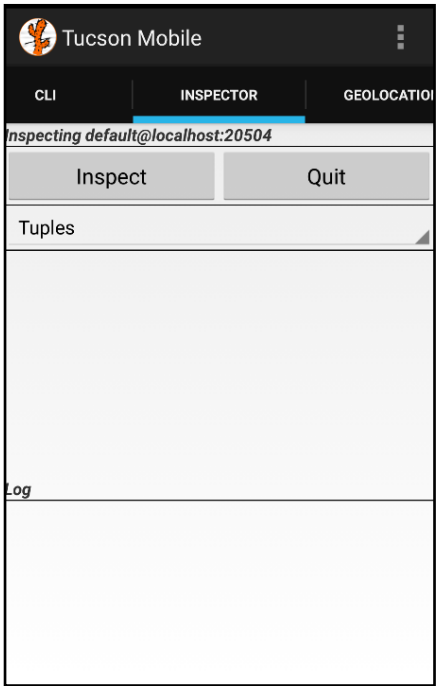
4. **Geolocation**: Fragment tramite il quale è possibile avviare o interrompere il servizio di geolocalizzazione, agganciandolo al nodo TuCSoN in esecuzione locale sul dispositivo

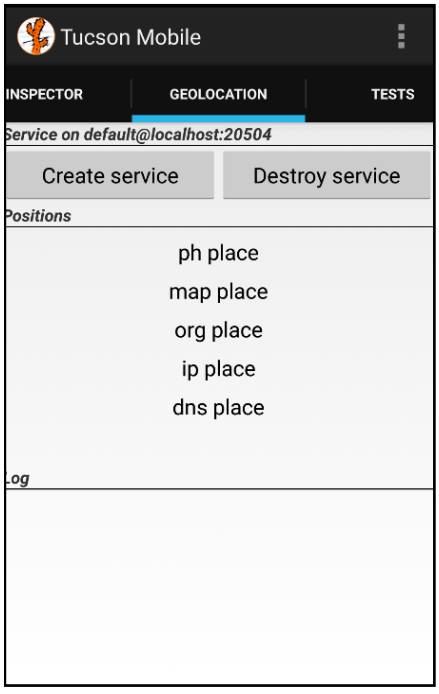
5. **Tests**: Fragment tramite il quale è possibile eseguire i test per verificare il corretto funzionamento dell'applicazione, delle primitive invocate e dei predicati di osservazione, guardie ed eventi forniti dal linguaggio di coordinazione.

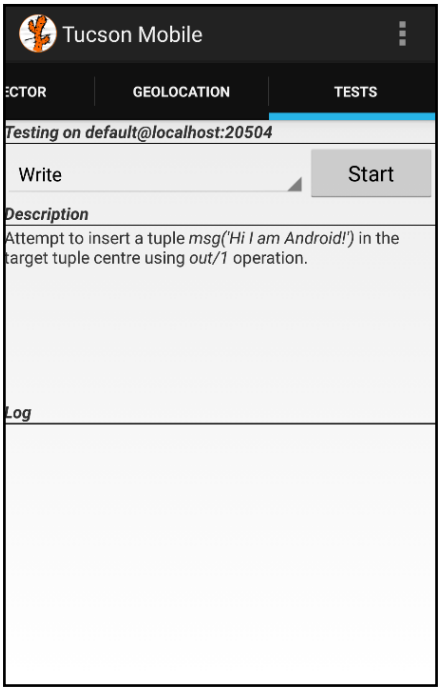
Screenshot:


Tucson Mobile
TUCSON NODE SERVICE
TUCSON NODE SERVICE
Executing on default@localhost:20504
Start Tucson Node
Stop Tucson Node
Status
TuCSoN Node is Inactive
Log


Tucson Mobile
CLI
INSPECTOR
Querying on default@localhost:20504
Type the TuCSoN Query
Execute
Clear
Result
Log


Tucson Mobile
CLI
INSPECTOR
Inspecting default@localhost:20504
Inspect
Quit
Tuples
Log


Tucson Mobile
INSPECTOR
GEOLOCATION
TESTS
Service on default@localhost:20504
Create service
Destroy service
Positions
ph place
map place
org place
ip place
dns place
Log


Tucson Mobile
GEOLOCATION
TESTS
Testing on default@localhost:20504
Write
Start
Description
Attempt to insert a tuple msg('Hi I am Android!') in the target tuple centre using out/1 operation.
Log

2.2 - App: Allineamento di TuCSoN

Giunti a questo punto diventa più facilmente comprensibile il primo obiettivo dichiarato all'inizio del documento che viene così introdotto.
Lo sviluppo di TuCSoN, nel tempo, ha reso necessario il rilascio di nuove versioni contenenti migliorie utili alla stabilità e all'efficienza del sistema, non consentendo però l'aggiornamento parallelo su piattaforma Android. Questo ha comportato così l'attuale coesistenza di due versioni TuCSoN significative.
La prima obsoleta ma:

- compatibile con l'App Android
- implementante un servizio di Geo-localizzazione
- parzialmente convertita al modello event-driven.

La seconda completamente aggiornata ma non comprendente tali caratteristiche.
A partire da queste considerazioni il primo obiettivo posto consiste nell'allineare il progetto mobile sviluppato con *TuCSoN 1.11.0.0209* all'ultima release *TuCSoN 1.12.0.0301*.

2.2.1 - Workplan

Come citato nella descrizione degli obiettivi, avendo Tucson su Android funzionalità aggiuntive, l'allineamento dell'App non può comprendere solamente l'aggiornamento di API modificate nel tempo ma si propone di migrare anche gli aspetti di geo-localizzazione e conversione al modello puramente event-driven. E' chiara la necessità di organizzare il lavoro.

Al fine quindi di evitare il più possibile bug o problematiche di vario tipo si è deciso di procedere isolando, in un primo momento, la parte "core" del Framework. Eliminando temporaneamente le funzionalità aggiuntive (geo-localizzazione) sia da TuCSoN sia dall'App viene garantita l'esclusione di possibili problemi relativi a queste. Una volta accertato che il sistema in configurazione base è funzionante si procederà al reintegro e al test delle parti disabilitate.

Per tali motivazioni il lavoro è stato suddiviso in questo modo:

1. **Configurazione dell'ambiente di lavoro**

2. **Eliminazione degli aspetti avanzati**
    a. Eliminazione Geo-localizzazione App
    b. Eliminazione Test Geo-localizzazione App
    c. Eliminazione parte grafica App (Fragment non necessari quali "Caso di Studio" e "Geo-localizzazione")

3. **Allineamento dell'App alla nuova Release TuCSoN**
    d. Aggiornamento libreria TuCSoN presente nell'App all'ultima Release disponibile
    e. Risoluzione dei problemi derivanti dall'aggiornamento (modifica API Tucson)

4. **Implementazione dei test**
    a. Modifica dei test di funzionamento standard di TuCSoN in modo che siano eseguibili su piattaforma Android

5. **Esecuzione dei test**

Prima di passare alla descrizione punto per punto del workplan soprastante vengono messe in evidenza le librerie e le relative versioni utilizzate.

2.2.2 - Librerie e versionamento dei sistemi

- Ambiente di sviluppo:

  Eclipse Luna 4.4.1

- Tucson:

  Tucson-1.12.0.0301

  (Ultima release attualmente disponibile)

- TuProlog:

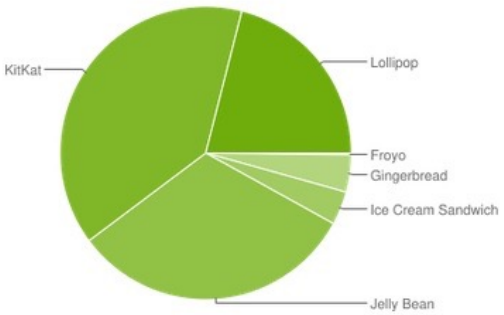
  2p-JDK7.jar

  (La libreria è compilata con Java 7. Java 8 non è al momento supportato da Android)

- Android:
  - MinSDKversion: API 17 (Android 4.2.2)
  - TargetSDKversion: API 19 (Android 4.4.2)
  - android-support-v4.jar

Secondo gli ultimi dati resi noti da Google il 39.2% dei dispositivi adottano Android KitKat (API 19) contro un 5.1% di Android Lollipop (API 22) [2].

Version	Codename	API	Distribution
2.2	Froyo	8	0.2%
2.3.3 - 2.3.7	Gingerbread	10	4.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	3.7%
4.1.x	Jelly Bean	16	12.1%
4.2.x		17	15.2%
4.3		18	4.5%
4.4	KitKat	19	39.2%
5.0	Lollipop	21	15.9%
5.1		22	5.1%



Nel caso in futuro si riveli necessario passare ad API Android più recenti non sembrano esserci particolari problemi di funzionamento. (Considerando le versioni attualmente disponibili).

2.2.3 - Configurazione dell'ambiente di lavoro

La configurazione di tutti gli strumenti di supporto potrebbe apparire di secondaria importanza ma è utile riportare i principali problemi riscontrati e le soluzioni adottate in questa fase per semplificare il lavoro di chi dovesse scontrarsi con problematiche simili o replicare il procedimento.
Tutto il codice sorgente è depositato in due repository messi a disposizione da un apposito servizio (bitbucket), accessibili solo previa autorizzazione.

REPOSITORY 1:

TuCSoN: https://bitbucket.org/smariani/tucson/

suddiviso in due branch di interesse:

1. ../feature/Android:
   https://bitbucket.org/smariani/tucson/branch/feature/Android contenente il codice aggiornato di TuCSoN v1.12.0.0301

2. ../feature/hovering:
   https://bitbucket.org/smariani/tucson/branch/feature/hovering
   contenente il codice della versione obsoleta di TuCSoN

REPOSITORY 2:

TuCSoNAndroid: https://bitbucket.org/smariani/tucsonandroid/

con un branch di interesse:

1. ../feature/hovering:
   https://bitbucket.org/smariani/tucsonandroid/branch/feature/hovering
   contenente il codice relativo all'App Android.

La prima fase di configurazione consiste quindi nel download dei repository git.

Completata la prima fase si passa al download ed integrazione delle librerie.

LIBRERIE

- android-support-v4.jar
- 2p-JDK7

Problematiche di configurazione

Dopo aver incluso le librerie sopra citate all'interno della cartella *"libs"* dell'App (da creare se non presente), il passaggio più ovvio è quello di collegare il progetto contenente Tucson aggiornato, ottenuto dal rispettivo repository, al progetto contenente l'App mediante "Build Path" di Eclipse.

Agendo in questo modo però possono essere generati una serie di errori in fase di avvio dell'app, non riconducibili all'allineamento, che non ne permettono l'esecuzione. Significativo il fatto che tutto funzioni eliminando il caricamento del fragment dedicato al CLI ma di cui non si vuole chiaramente fare a meno.

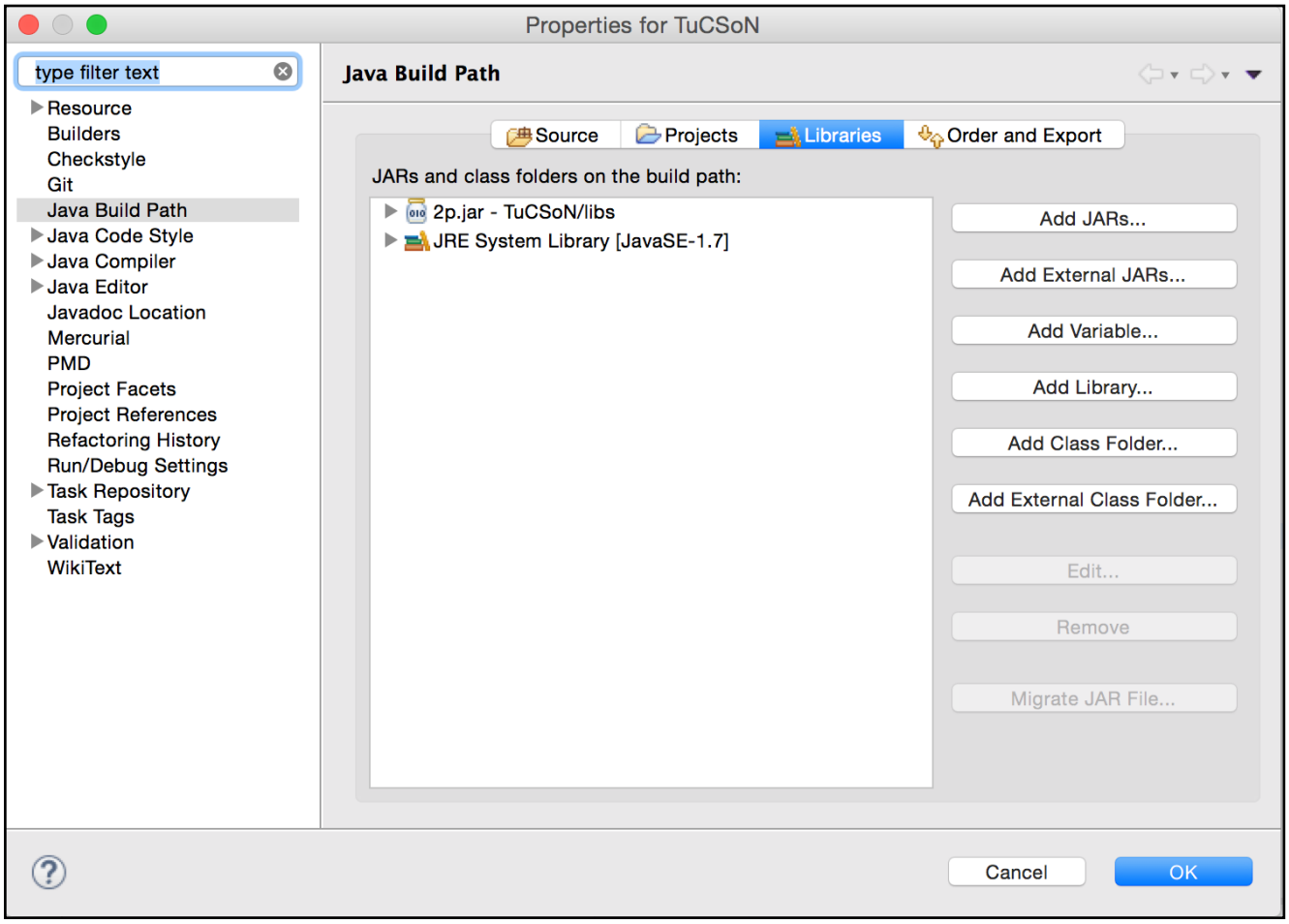
Soluzioni

I problemi sono stati totalmente risolti scaricando TuCSoN in formato ".jar" (https://bitbucket.org/smariani/tucson/downloads) ed includendolo all'interno della cartella "libs" al pari delle altre librerie.

Il motivo risiede nel fatto che tale *.jar* è attualmente compilato con *Java 7*, compatibile con Android.

Nel caso sia però necessario accedere alle classi TuCSoN sarà sufficiente collegare il progetto mediante *buildPath* di Eclipse (come spiegato precedentemente) ma ricompilando con *Java 7* per renderlo compatibile alla piattaforma Android.

Qui di seguito viene riportato il pannello Eclipse relativo alle librerie:



2.2.4 - Eliminazione aspetti avanzati ed allineamento

Una volta configurato l'ambiente di lavoro si è proceduto ad eliminare temporaneamente la geo-localizzazione dall'App per poterla affrontare separatamente in un secondo momento.

In particolare sono state eliminate:

- le classi appartenenti al package *"it.unibo.tucson.android. geolocation"*

- il codice relativo ai test sulla geo-localizzazione nelle classi *"Tests"* e *"TestAgent"* appartenenti al package *"it.unibo.tucson.android.tests"*

- il codice relativo alla visualizzazione del Fragment inerente alla geo-localizzazione e al vecchio caso di studio all'interno della *"MainActivity"* nel package *"it.unibo.tucson.android"*

Risolte le dipendenze minori l'App è stata momentaneamente epurata della geo-localizzazione.
Le dipendenze irrisolte rimanenti comprendono la chiamata di metodi TuCSoN aggiornati e modificati principalmente nel nome.
In particolare sono state modificate semplicemente le chiamate a metodi appartenenti a poche classi contenuti nei package:

- alice.tucson.parsing
- alice.tucson.network

A questo punto il codice viene compilato ed il focus passa all'aggiornamento e all'esecuzione dei test.

2.2.5 - Test

Una volta risolti tutti gli errori ed avendo l'applicazione funzionante è stato necessario adattare i test standard di TuCSoN in modo che fossero eseguibili su dispositivo Android.
I test adattati sono i seguenti:

- **write**
- **read**
- **write&read**
- **helloWorld**
- **diningPhilosophers**
- **timedDiningPhilosophers**
- **distributedDiningPhilosophers**
- **asynchAPI**
- **rbac**
- **spawnedWorkers**
- **spawnedWorkers2P**

write:

Inserisce la tupla *"Hi, I'm Android"* nel centro di tuple avviato usando l'operazione *"out"*.
Implementazione originale non modificata e funzionante dopo l'aggiornamento.

read:

Legge una tupla corrispondente al template *"msg(who)"* dal centro di tuple avviato usando l'operazione *"in"*.
Implementazione originale non modificata e funzionante dopo l'aggiornamento.

write&read:

Concatenazione dei due test precedenti.
Implementazione originale non modificata e funzionante dopo l'aggiornamento.

helloWorld:

Test basilare. Porta a termine una *"out"* ed una *"read"*.
Non ci sono particolari modifiche a livello architetturale, lievi modifiche a livello di codice causa interfaccia grafica e stampa del log.

diningPhilosophers:

Classico test di coordinazione che prevede N filosofi i quali possono “pensare” o “mangiare”.
Non è stato necessario apportare modifiche radicali. Ovviamente si è proceduto con l’adattamento all’interfaccia grafica Android che ha richiesto cambiamenti a livello di codice oltre al caricamento di file eseguito sul dispositivo.
E’ stato introdotto un meccanismo temporizzato per poter terminare il test che altrimenti iterava all’infinito.

timedDiningPhilosophers:

Classico test di coordinazione che prevede N filosofi i quali possono “pensare” o “mangiare”. In questo caso viene scambiato con il centro di tuple anche il tempo massimo in cui un filosofo può “mangiare” ovvero occupare il “lock”.
Non è stato necessario apportare modifiche radicali. Ovviamente si è proceduto con l’adattamento all’interfaccia grafica Android che ha richiesto cambiamenti a livello di codice oltre al caricamento di file eseguito sul dispositivo.
E’ stato introdotto un meccanismo temporizzato per poter terminare il test che altrimenti iterava all’infinito.

distributedDiningPhilosophers:

Classico test di coordinazione che prevede N filosofi i quali possono “pensare” o “mangiare” eseguito su più nodi.
In questo caso è stato necessario apportare modifiche più importanti all’app che fortunatamente non comprendevano ostacoli derivanti dal sistema operativo. Una volta verificato infatti che più “service” possono essere eseguiti su porte differenti dello stesso dispositivo si è proceduto a duplicare il nodo, il fragment e le relative impostazioni in modo da avere il controllo su due centri di tuple differenti simulando un sistema distribuito.
In seguito il programma di test è stato modificato a livello di codice per adattarsi all’applicazione. L’esecuzione presuppone ovviamente l’avvio dei due centri di tuple.

asynchAPI:

Le primitive per accedere al centro di tuple locale sono testate in maniera asincrona introducendo un elemento di non determinismo.
Per il corretto funzionamento di questo test all’interno dell’app è stato necessario introdurre un meccanismo di sincronizzazione in modo tale che l’app potesse rilevare il reale momento in cui ha termine il test.

rbac:

Classico test di sicurezza qui adottato per la verifica delle credenziali degli agenti che interagiscono con il nodo TuCSoN.

spawnedWorkers:

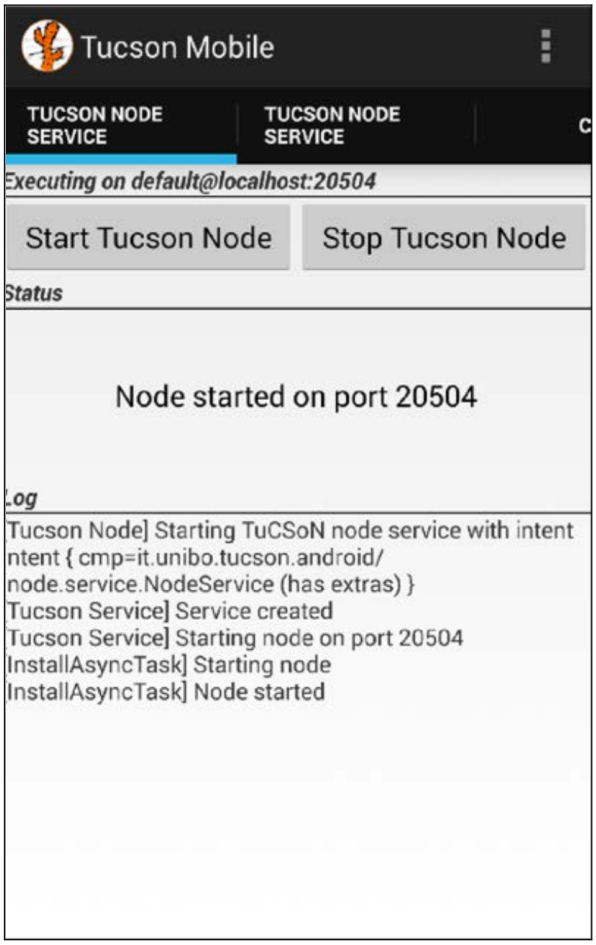
Implementazione dell'architettura Master/Worker che utilizza il nodo TuCSoN come uno spazio di memoria condiviso. Il Master delega al Worker la computazione di alcuni fattoriali.
Allo stato iniziale, così come preso da *alice.tucson.examples.spawnedWorkers*, la versione java di questo test era non terminante.
Nella versione android (*it.unibo.tucson.android.tests.spawnedWorkers*) il comportamento è stato completamente riscritto.

spawnedWorkers2P:

La stessa architettura del test precedente ma data in pasto al motore tuProlog.
Per renderlo eseguibile è stato necessario eseguire un riallineamento con la v3.0 di tuProlog poiché una signature era cambiata. Inoltre il logger di tuProlog è stato redirottato verso l'handler dei log di android.

ESECUZIONE DEI TEST

Tutti i test sono stati eseguiti e completati con successo



CAPITOLO 3 - AGGIORNAMENTO TuCSoN: GEO-LOCALIZZAZIONE

Ottenuta una versione dell'App allineata alla release TuCSoN più recente si procede reintegrando gli aspetti avanzati temporaneamente disabilitati, ovvero nello specifico la geo-localizzazione.
Viene presentata quindi una panoramica di tale strumento, la sua gestione su piattaforma Android per poi procedere alle modifiche TuCSoN.

3.1 - Global Positioning System

3.1.1 - Descrizione

[4] Il sistema di posizionamento globale (Global Positioning System o GPS) è un sistema di posizionamento e navigazione satellitare civile. Esso, attraverso una rete dedicata di satelliti artificiali in orbita, fornisce ad un terminale mobile o ricevitore GPS informazioni sulle sue coordinate geografiche ed orario, indipendentemente dalla posizione sulla Terra o
dalle condizioni meteorologiche, ove quindi vi sia un contatto privo di ostacoli con almeno quattro satelliti del sistema. La localizzazione avviene tramite la trasmissione di un segnale radio da parte di ciascun satellite e l'elaborazione dei segnali ricevuti da parte del ricevitore.
Il sistema GPS è gestito dal governo degli Stati Uniti d'America ed è liberamente accessibile da chiunque sia dotato di un ricevitore GPS.
Il suo grado attuale di accuratezza è dell'ordine dei metri, dipendente dalle condizioni meteorologiche, dalla disponibilità e dalla posizione dei satelliti rispetto al ricevitore, dalla qualità e dal tipo di ricevitore, dagli effetti di radiopropagazione del segnale in ionosfera e troposfera (riflessione) e dagli effetti della relatività.

Funzionamento

Il principio di funzionamento si basa su un metodo di posizionamento sferico (trilaterazione), che parte dalla misura del tempo impiegato da un segnale radio a percorrere la distanza satellite-ricevitore.
Poiché il ricevitore non conosce quando è stato trasmesso il segnale dal satellite, per il calcolo della differenza dei tempi il segnale inviato dal satellite è di tipo orario, grazie all'orologio

atomico presente sul satellite: il ricevitore calcola l'esatta distanza di propagazione dal satellite a partire dalla differenza (dell'ordine dei microsecondi) tra l'orario pervenuto e quello del proprio orologio sincronizzato con quello a bordo del satellite, tenendo conto della velocità di propagazione del segnale.
L'orologio a bordo dei ricevitori GPS è però molto meno sofisticato di quello a bordo dei satelliti e deve essere corretto frequentemente, non essendo altrettanto accurato sul lungo periodo. In particolare la sincronizzazione di tale orologio avviene all'accensione del dispositivo ricevente, utilizzando l'informazione che arriva dal quarto satellite, venendo così continuamente aggiornata. Se il ricevitore avesse anch'esso un orologio atomico al cesio perfettamente sincronizzato con quello dei satelliti, sarebbero sufficienti le informazioni fornite da tre satelliti, ma non è così e dunque il ricevitore deve risolvere un sistema di quattro incognite (latitudine, longitudine, altitudine e tempo) in quattro equazioni.

3.1.2 - Diffusione

Con la diffusione di Internet su dispositivi mobili è innegabile che questi siano diventati strumenti fondamentali alle attività quotidiane delle persone. Se nel recente passato la geo-localizzazione veniva sfruttata principalmente e quasi esclusivamente attraverso il navigatore satellitare (a livello civile), attualmente questa sta assumendo un ruolo rilevante su smartphone in diversi ambiti, dal rintracciamento del dispositivo alla localizzazione di post, video e foto sui social network così come al supporto alle applicazioni ed alla realtà aumentata.
I sistemi di localizzazione possono essere:

- **Network-based**: rileva le reti mobili Wi-Fi e GSM disponibili nella zona ed in base a questi calcola la propria posizione. Non molto accurato ma immancabile nei dispositivi. Utile anche nella localizzazione interna.

- **GPS**: Il sistema descritto precedentemente basato sull'intercettazione di messaggi inviati da satelliti che ruotano attorno alla Terra. Tali comunicazioni contengono l'informazione oraria ed altri dati relativi all'orbita percorsa. Il dispositivo intercettando i segnali di almeno quattro di questi satelliti con l'applicazione di formule matematiche

riesce a calcolare la propria posizione. Accurato e diffusissimo tranne che in alcuni dispositivi di fascia bassa. Praticamente il sistema di localizzazione per antonomasia.

- **Ibridi:** La combinazione dei due sistemi sopra descritti. Attualmente una soluzione molto diffusa ed efficace.

3.2 - La Geo-localizzazione su Android

Android, come tutti i maggiori sistemi operativi mobile integra la gestione della geo-localizzazione con un sistema ibrido.

3.2.1 - Strumenti

Le Api messe a disposizione per determinare la posizione corrente sono contenute nell'apposito package *"android.location"*[5].
In particolare:

- **Location Manager:** E' la classe che mette a disposizione l'accesso al servizio di localizzazione Android. Questi servizi permettono di comunicare con i "location providers", registrare posizioni, aggiornare appositi listener, gestire avvisi di prossimità ed altro.

- **Location Provider:** E' la super-classe dei diversi location providers i quali distribuiscono le informazioni sulla posizione attuale. Quest'ultima è rappresentata dalla classe "Location". Il dispositivo Android deve poter comunicare con diversi "Location provider" attivi dando la possibilità di selezionare quello voluto. In sostanza sono gli strumenti che gestiscono fisicamente la geolocalizzazione quindi saranno ad esempio la rete WI-FI o il ricevitore GPS

- **Criteria:** E' un oggetto che permette di scegliere il miglior "Location provider" a disposizione permettendo molta flessibilità dato che permette di impostare come il provider deve essere scelto.

- **Location Listener:** Registrato ad un Location Manager permette di ricevere aggiornamenti periodici sulla posizione

- **Proximity Alert:** Avviso attivato nel caso il dispositivo entri in un area settata mediante longitudine, latitudine e raggio.

- **Geocoder:** Classe che permette di determinare le coordinate geografiche (longitudine, latitudine) per un indirizzo oppure il possibile indirizzo date le coordinate. Questa funzionalità è ovviamente garantita da servizi Google chiamati all'occorrenza.

E' importante ricordare che per poter utilizzare i servizi di geolocalizzazione è necessario abilitare i relativi permessi sull'applicazione:

- ACCESS_FINE_LOCATION (GPS + NETWORK)
- ACCESS_COARSE_LOCATION (NETWORK)

L'utente infine può controllare se un "Location Manager" è attivo o meno. Nel caso negativo è possibile seguire l'esempio riportato sotto.

```
LocationManager service = (LocationManager)
getSystemService(LOCATION_SERVICE);
boolean enabled = service
  .isProviderEnabled(LocationManager.GPS_PROVIDER);

// check if enabled and if not send user to the GSP settings
// Better solution would be to display a dialog and suggesting to
// go to the settings
if (!enabled) {
  Intent intent = new
Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
  startActivity(intent);
}
```

3.2.2 - Esempio esplicativo

Viene riportato in seguito un esempio di applicazione in grado di richiedere informazioni GPS e mostrarle [5]. Oltre a latitudine e longitudine l'Activity recupererà l'indirizzo cui corrisponde la posizione (utilizzando il Geocoder sopra descritto):

Layout:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <TableRow android:padding="5dp">
        <TextView android:text="Abilitato" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        <TextView android:id="@+id/enabled" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
    </TableRow>
    <TableRow android:padding="5dp">
        <TextView android:text="Data ora" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        <TextView android:id="@+id/timestamp" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
    </TableRow>
    <TableRow android:padding="5dp">
        <TextView android:text="Latitudine" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        <TextView android:id="@+id/latitude" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
    </TableRow>
    <TableRow android:padding="5dp">
        <TextView android:text="Longitudine" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        <TextView android:id="@+id/longitude" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
    </TableRow>
    <TableRow android:padding="5dp">
        <TextView android:text="Località" android:padding="5dp"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        <TextView android:id="@+id/where" android:padding="5dp"
            android:lines="2"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
    </TableRow>
</TableLayout>
```

Impostazione dei permessi (GPS + NETWORK):

```
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Registrazione Listener presso il relativo Location Manager (Operazione nel metodo onResume().
L'annullamento in onPause()):

```
public class MainActivity extends Activity
{
    private String providerId = LocationManager.GPS_PROVIDER;
    private Geocoder geo = null;
    private LocationManager locationManager=null;
    private static final int MIN_DIST=20;
    private static final int MIN_PERIOD=30000;

    private LocationListener locationListener = new
LocationListener()
    {
        . . .
        . . .

    };

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    protected void onResume()
    {
        super.onResume();
        geo=new Geocoder(this, Locale.getDefault());
        locationManager = (LocationManager)
getSystemService(LOCATION_SERVICE);
        Location location =
locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
        if (location!=null)
            updateGUI(location);
        if (locationManager!=null &&
locationManager.isProviderEnabled(providerId))
            updateText(R.id.enabled, "TRUE");
        else
            updateText(R.id.enabled, "FALSE");
        locationManager.requestLocationUpdates(providerId,
MIN_PERIOD,MIN_DIST, locationListener);
    }

    @Override
    protected void onPause()
    {
        super.onPause();
        if (locationManager!=null &&
locationManager.isProviderEnabled(providerId))
            locationManager.removeUpdates(locationListener);
    }
. . .
```

Il metodo "*requestLocationUpdates*" effettua la vera registrazione del listener.
I parametri che utilizza sono:

- l'ID del provider: la costante stringa che individua il tipo di provider da usare;
- il minimo intervallo di tempo, in millisecondi, che deve trascorrere tra aggiornamenti della posizione;
- la minima distanza in metri che deve intercorrere tra due misurazioni;
- l'oggetto che svolge il ruolo di listener.

Implementazione listener:

```
private LocationListener locationListener = new LocationListener()
    {
        @Override
        public void onStatusChanged(String provider, int status,
Bundle extras)
        {

        }
        @Override
        public void onProviderEnabled(String provider)
        {
            // attivo GPS su dispositivo
            updateText(R.id.enabled, "TRUE");
        }
        @Override
        public void onProviderDisabled(String provider)
        {
            // disattivo GPS su dispositivo
            updateText(R.id.enabled, "FALSE");
        }
        @Override
        public void onLocationChanged(Location location)
         {
            updateGUI(location);
         }
    };
```

I primi tre metodi "*onStatusChanged*", "*onProviderEnabled*", "*onProviderDisabled*" notificano, rispettivamente, se il provider è disponibile o meno, se è abilitato, se è stato disabilitato.
L'ultimo metodo "*onLocationChanged*" è la parte principale del listener e viene invocato ogni volta che nuove informazioni di posizione sono state recapitate.
L'oggetto "Location" contiene tutto ciò che è stato appreso dall'ultima misurazione del posizionamento e viene inviata al metodo "updateGUI" per riflettere gli aggiornamenti sulla interfaccia utente:

```
private void updateGUI(Location location)
    {
        Date timestamp = new Date(location.getTime());
        updateText(R.id.timestamp, timestamp.toString());
        double latitude = location.getLatitude();
        updateText(R.id.latitude, String.valueOf(latitude));
        double longitude = location.getLongitude();
        updateText(R.id.longitude, String.valueOf(longitude));
        new AddressSolver().execute(location);
    }

private void updateText(int id, String text)
    {
        TextView textView = (TextView) findViewById(id);
        textView.setText(text);
    }
```

All'interno di "updateGUI", oltre al codice di modifica delle TextView, è presente l'invocazione al Geocoder per la conversione delle coordinate in un indirizzo vero e proprio.
Il Geocoder viene consultato in maniera asincrona mediante "AsyncTask". Nel metodo "doInBackground", la Location sarà convertita in una stringa frutto della concatenazione delle informazioni reperite:

```
private class AddressSolver extends AsyncTask<Location, Void,
String>
    {

        @Override
        protected String doInBackground(Location... params)
        {
            Location pos=params[0];
            double latitude = pos.getLatitude();
            double longitude = pos.getLongitude();

            List<Address> addresses = null;
            try
            {
                addresses = geo.getFromLocation(latitude,
longitude, 1);
            }
            catch (IOException e)
            {

            }
            if (addresses!=null)
            {
                if (addresses.isEmpty())
                  {
                    return null;
                  }
                  else {
                     if (addresses.size() > 0)
                     {
                         StringBuffer address=new StringBuffer();
                         Address tmp=addresses.get(0);
                         for (int
y=0;y<tmp.getMaxAddressLineIndex();y++)
                            address.append(tmp.getAddressLine(y)
+"\n");
                         return address.toString();
                     }
                  }
            }
            return null;
        }

        @Override
        protected void onPostExecute(String result)
        {
            if (result!=null)
                updateText(R.id.where, result);
            else
                  updateText(R.id.where, "N.A.");

        }
```

3.2.3 – API

Infine vengono riportate esaustivamente le API messe a disposizione da Android per la geo-localizzazione [6]:

CLASSI

- **Address**: classe che rappresenta un indirizzo, cioè un insieme di stringhe che descrivono una posizione. Il formato address è una versione semplificata di xAL

- **Criteria**: classe che indica i criteri di applicazione per la selezione di un fornitore di posizione. Provider forse ordinate in base alla precisione, il consumo di energia, capacità di relazione, altitudine, velocità e portamento e costo monetario

- **Geocoder**: classe per la gestione di geocoding e il geocoding inverso. Geocoding è il processo di trasformazione di un indirizzo stradale o un'altra descrizione di un luogo in coordinate (latitudine, longitudine). Geocoding inverso è il processo di trasformazione di coordinare (latitudine, longitudine) in un indirizzo (parziale) . La quantità di dettagli può variare, ad esempio si può contenere l'indirizzo stradale di un edificio, mentre un altro potrebbe contenere solo il nome della città e il codice postale . La classe Geocoder richiede un servizio di back-end che non è incluso nel core di Android.

- **GpsSatellite**: questa classe rappresenta lo stato corrente di un satellite GPS e viene utilizzata in combinazione con la classe GpsStatus .

- **GpsStatus**: questa classe rappresenta lo stato corrente del motore GPS . e viene utilizzata in combinazione con l'interfaccia GpsStatus.Listener.

- **Location**: classe dati che rappresenta una posizione geografica. Una posizione può essere costituita da latitudine, longitudine, timestamp e altre informazioni come posizione, altitudine e velocità. Tutti i luoghi generati dal *LocationManager* sono garantiti avere una valida latitudine, longitudine, e timestamp, tutti gli altri parametri sono opzionali.

- **LocationManager**: questa classe fornisce l'accesso ai servizi di localizzazione del sistema . Questi servizi consentono alle applicazioni di ottenere aggiornamenti periodici sulla posizione geografica del dispositivo o innescare una Intent quando il dispositivo entra nella vicinanza di una determinata posizione geografica.

- **LocationProvider**: una superclasse astratta per i fornitori di posizione . Un provider di posizione fornisce rapporti periodici sulla posizione geografica del dispositivo. Ogni fornitore ha una serie di criteri in base ai quali può essere utilizzato, per esempio, alcuni provider richiedono hardware GPS e visibilità ad un numero di satelliti mentre altri richiedono l'uso della radio cellulare, o l'accesso alla rete specifica o ad Internet. Essi possono anche avere diverse caratteristiche di consumo della batteria o costi monetari per l'utente. La classe Criteria consente ai fornitori di specificare queste scelte.

- **SettingInjectorService**: classe che dinamicamente specifica lo stato di attivazione di una preferenza iniettato nella lista delle impostazioni dell'applicazione visualizzata dall'app impostazioni di sistema. Utilizzata soltanto da applicazioni che sono incluse nell'immagine di sistema per preferenze che riguardano applicazioni multiple. Posizioni che riguardano una sola applicazione dovrebbero apparire soltanto all'interno di tale applicazione, piuttosto che nelle impostazioni di sistema. Questa classe è stata introdotta soltanto a partire dal livello di API 19.

INTERFACCE

- **GpsStatus.Listener**: interfaccia utilizzata per ricevere le notifiche quando lo stato del GPS è cambiato.

- **GpsStatus.NmeaListener**: interfaccia utilizzata per ricevere messaggi NMEA dal GPS. NMEA 0183 è uno standard per la comunicazione con i dispositivi elettronici marini ed è un metodo comune per ricevere dati da un GPS, tipicamente attraverso una porta seriale. È possibile implementare questa interfaccia e chiamare *addNmeaListener* (GpsStatus.NmeaListener) per ricevere dati NMEA dal motore GPS.

- **LocationListener**: interfaccia utilizzata per ricevere le notifiche dal LocationManager quando la posizione è cambiata. Questi metodi sono chiamati se il *LocationListener* è stato registrato con il servizio di location manager utilizzando i *requestLocationUpdates* (String, long, float, LocationListener).

3.2.4 – Una possibile applicazione

La geo-localizzazione sopra descritta potrebbe essere applicata allo sviluppo di un'infrastruttura annoverante uno spazio nel quale tutti gli agenti TuCSoN vengano informati della posizione di nuovi agenti in avvicinamento 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 server che si occupa di mantenere tutte le associazioni tra agente e parametri di identificazione relativi al centro di tuple associato. Tale server ha anche la funzione di inoltrare a tutti gli agenti già registrati la nuova presenza e della sua posizione.

SCENARIO

Per l'implementazione del caso di studio, viene considerato un semplice scenario relativo ad un museo contenente tre sale ed un server condiviso. In particolare si suppone che in due sale siano presenti due custodi aventi un dispositivo mobile ciascuno. Per ogni dispositivo sono in esecuzione un nodo ed un'agente in attesa di ricevere notifiche relative all'arrivo del terzo custode.
Nel momento in cui il custode mancante raggiunge la sua sala, l'agente in esecuzione sul suo dispositivo 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. 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 la sala specifica nel quale si trova il custode appena arrivato.

ARCHITETTURA

L'architettura generica che permette di strutturare il sistema proposto è composta da:

- Server: agente posto sul nodo server incaricato di inizializzare il centro di tuple remoto

- InitAgent : agente posto su ogni dispositivo mobile incaricato di inizializzare il centro di tuple sul dispositivo smart.

- 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.

Gli agenti *Server* e *InitAgent* dovrebbero essere avviati contestualmente alla fase di configurazione del sistema. 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. Una possibile rappresentazione che rispecchi l'architettura appena descritta può essere data da un computer desktop che ospita il server sul quale è presente un centro di tuple necessario per gestire 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 immaginare la presenza di un centro di tuple all'interno del quale l'agente *InitAgent* inizializza le tuple necessarie e configura il comportamento necessario per garantire la corretta interazione con il server e gli altri agenti.

3.3 - La Geo-localizzazione su TuCSoN

Presentata la geo-localizzazione su piattaforma Android diventa necessario, prima di agire sulla modifica di TuCSoN, introdurre gli aspetti concettuali utili alla gestione dello spazio con cui si andrà ad arricchire il modello [1].
La coordinazione spaziale richiede che il media di coordinazione supporti in particolare la proprietà di spatial situatedness.

Spatial Situatedness: *La proprietà di un ente di essere strettamente in relazione con l'ambiente, tipicamente espressa in termini di reazioni ai cambiamenti di quest'ultimo. [12]*

La proprietà di situatedness richiede che un'astrazione di coordinazione space-aware sia sempre associata ad un posizionamento assoluto 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 non riguarda solamente la posizione ma anche qualsiasi tipo di movimento. Le astrazioni software, infatti, potrebbero muoversi all'interno di uno spazio virtuale (la rete), oppure fisico (lo spazio tridimensionale), oltre alla possibilità di essere ospitate da un dispositivo mobile condividendo con esso la posizione.
Risulta chiara la necessità di capacità di movimento e vengono messi in evidenza due tipi di posizionamento:

- Posizionamento fisico
  - assoluto (latitudine, longitudine, altitudine)
  - geografico (Via Genova, Cesena)
  - organizzativo (Ufficio 2 del DISI, sede di Cesena).

- Posizionamento virtuale (rete)
  - assoluto (indirizzo IP)
  - relativo (localizzazione del dominio tramite DNS).

3.3.1 - Estensione del modello: lo spazio

Così come le nozioni temporali potenziano i centri di tuple dotandoli di cognizione di tempo, le nozioni spaziali devono estendere i centri di tuple allo scopo di affrontare le problematiche enunciate precedentemente.
Per definire la posizione di un centro di tuple si deve introdurre la nozione di posizione corrente [1]. 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.
Analogamente agli eventi temporali, è possibile specificare reazioni innescate da eventi spaziali, quindi 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.
Poiché la maggior parte dei sistemi software e hardware odierni possono accedere ai dati GPS, diventa disponibile, anche nei sistemi distribuiti, una descrizione fisica, globale e assoluta di qualsiasi proprietà spaziale.
Allo scopo di consentire leggi di coordinazione che permettano ad agenti e artefatti di dialogare a proposito di spazio, vengono introdotti quindi nuovi eventi:

- ***from(⟨Place⟩)*:** rappresenta la reazione innescata dall'evento spaziale generato dal movimento di un'entità che lascia una data locazione fisica rappresentata da ⟨*Place*⟩ (L'evento spaziale generato quando il dispositivo che ospita il centro di tuple comincia a muoversi).

- ***to(⟨Place⟩)*:** rappresenta la reazione innescata dall'evento spaziale generato dal movimento di un'entità che raggiunge una data locazione fisica rappresentata da ⟨*Place*⟩ (L'evento spaziale generato quando il dispositivo termina il movimento)

Nuovi predicati di osservazione utili per l'accesso alle proprietà spaziali e la loro valutazione:

- **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.

Infine nuove guardie:

- **at(⟨Place⟩):** innesca una reazione quando il centro di tuple si trova nella locazione fisica definita da ⟨Place⟩.

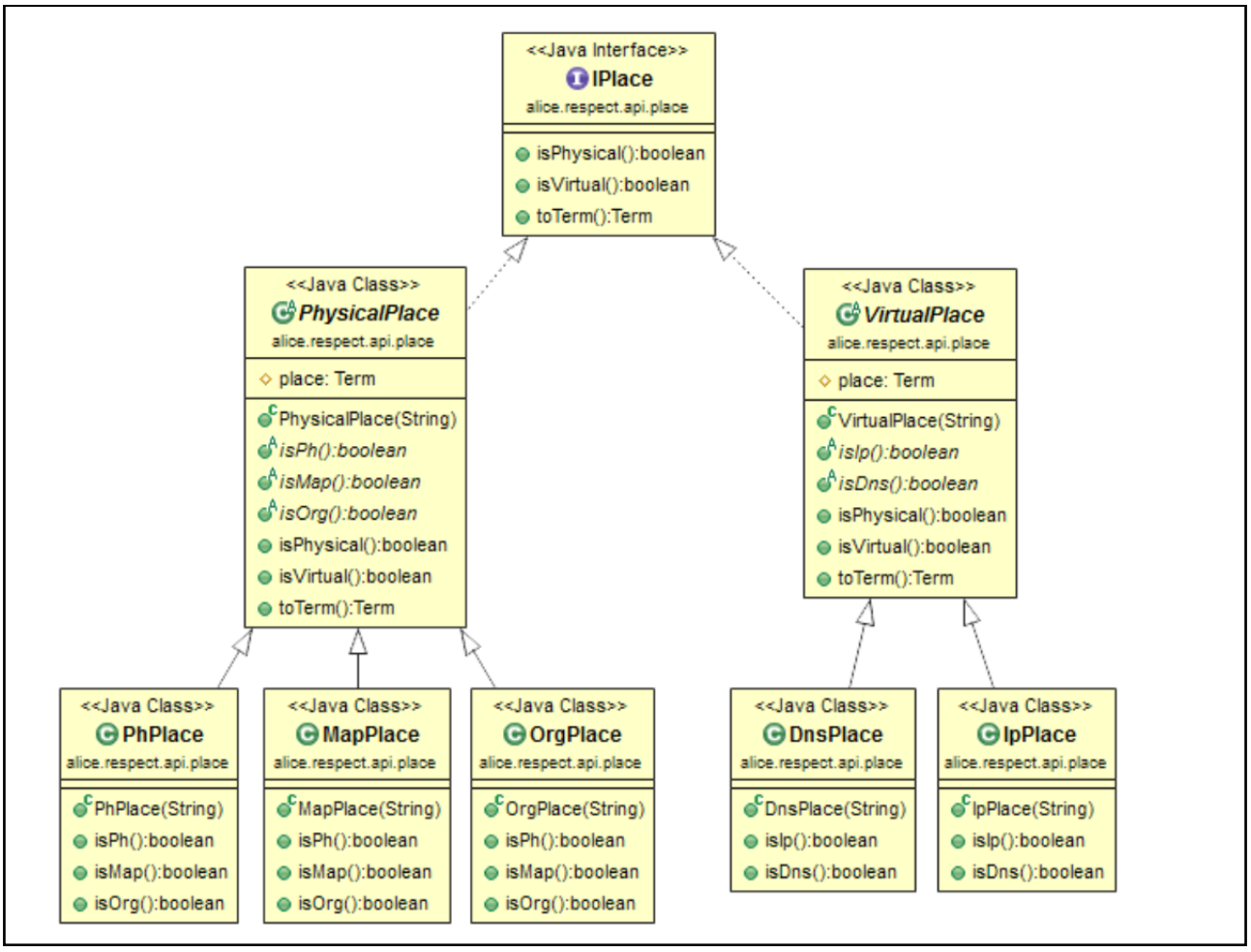
- **near(⟨Place⟩,⟨Radius⟩):** innesca una reazione quando il centro di tuple si trova in prossimità della locazione fisica definita da ⟨*Place*⟩ a patto che essa sia entro un raggio definito da ⟨*Radius*⟩.

Tali costrutti potenziano quindi il medium di coordinazione rendendolo, una volta programmato opportunamente, in grado di acquisire a tutti gli effetti la proprietà space-aware riconoscendo eventi, accedendo alle informazioni e valutando condizioni facenti parte della dimensione spaziale in un sistema coordinato e situato.

A questo punto non resta che concretizzare tutto questo tramite una rapida descrizione dell'implementazione di posizione, predicati e guardie nel codice TuCSoN per poi arrivare ad introdurre gli strumenti di geo-localizzazione che si basano sull'estensione del modello verso lo spazio appena presentato.

Prima di qualunque aspetto spaziale è necessario disporre di un'entità che rappresenti la posizione ed a questo scopo è stata definita la classe *Position* che incapsula tutte le informazioni riguardanti la posizione virtuale o fisica.

Le varie tipologie di posizione sono state definite tramite una gerarchia di classi specifiche sotto riportata [1]:



Tale gerarchia può essere brevemente spiegata come:

- **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.

Definita la posizione vengono implementati i predicati e le guardie sopra riportate agendo direttamente sulla classe *Respect2PLibrary*, inserendo i metodi *current_place_2*, *event_place_2*, start_place_2, *near_3* e *at_2* descritti dal progettista nel precedente lavoro di tesi che qui viene parzialmente riportato per quel che riguarda i predicati [1]:

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.

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.

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 averne discriminato il tipo e recuperato la posizione corrispondente, si procede all'unificazione di questa con quella specificata in ingresso.

3.3.2 - La prima implementazione

Giunti a questo punto, avendo definito il modello architetturale che formalizza i componenti utili alla gestione dello spazio e, allo stesso fine, l'estensione del linguaggio di coordinazione, diventa importante descrivere l'implementazione dell'architettura che sfrutterà tali estensioni: la geo-localizzazione [1].
E' importante sottolineare che tale struttura è stata progettata, in una tesi precedente, per espletare connessioni e interazioni tra il sistema TuCSoN e la piattaforma di geo-localizzazione specifica del sistema ospitante.

Visto l'ottimo lavoro svolto e la documentazione piuttosto dettagliata è stato deciso di non riprogettare tutto ma piuttosto di agire aggiornando ed adattando le parti incompatibili con l'ultima versione di TuCSoN. Viene fornita quindi una prima descrizione delle soluzioni adottate per arrivare alle modifiche necessarie all'aggiornamento.

La geo-localizzazione in TuCSoN è stata pensata e suddivisa principalmente in tre livelli:

- **GeolocationService:** entità astratta che rappresenta un servizio di geo-localizzazione 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 geo-localizzazione.

Al fine di garantire la corretta interazione tra il servizio di geo-localizzazione ed il middleware di coordinazione è stato adottato il pattern EventListener. Tale pattern prevede un ascoltatore in attesa di notifiche provenienti da altre entità, nel caso specifico dal servizio di geo-localizzazione.
E' importante riportare una breve descrizione delle classi sopra citate:

GeolocationService

L'entità astratta *GeolocationService* rappresenta il generico servizio di geo-localizzazione, caratterizzato da un identificatore *(GeoServiceId)*, una piattaforma di esecuzione ed un centro di tuple a cui riferirsi relativamente agli aggiornamenti su posizione e movimento. Questo componente incapsula inoltre una lista di ascoltatori associati a *GeolocationServiceListener* che ricevono 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 geo-localizzazione deve implementare. Ne viene riportata la struttura.

IGeolocationService:

void notifyLocationChanged(double lat, double lng):

Notifica agli ascoltatori collegati che la posizione fisica assoluta del dispositivo 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 geo-localizzazione 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 geo-localizzazione.

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 Geo-coding di un indirizzo geografico dato in ingresso.

Da notare il metodo *"generateSpatialEvents(boolean generate)"*. Esso è importante per impostare il valore di una variabile di controllo, denominata *"genSpatialEvents"* all'interno della classe *GeolocationService*. Nel caso siano presenti reazioni spaziali nella specifica di comportamento, tale variabile assume valore true, altrimenti false.

In questo modo è possibile, per il servizio di geo-localizzazione, sapere se sarà necessario generare gli eventi spaziali *from* e *to*.

Il motivo per cui *GeolocationService* rappresenta un'entità astratta risiede nel fatto che deve essere concretizzato da un'implementazione specifica del servizio di geo-localizzazione, la quale dovrà fornire il supporto necessario per rendere possibile l'interazione fra la tecnologia propria del dispositivo utilizzato e il middleware di coordinazione.

Tale implementazione corrisponde alle API specifiche per la tecnologia del dispositivo e dovrà interfacciarsi con esse, recuperare le informazioni di posizione richieste e notificarle agli ascoltatori collegati sfruttando i metodi già definiti in *GeolocationService*.

Questi metodi si occupano semplicemente di scorrere la lista degli ascoltatori collegati ed invocare il metodo opportuno, indipendentemente da ciò che si deve notificare.

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

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

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

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

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

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

GeolocationServiceListener

Come riportato nella documentazione del progettista il *"GeolocationServiceListener"* rappresenta l'entità ascoltatore che attende notifiche riguardanti aggiornamenti di posizione e generazione degli eventi di movimento provenienti dal servizio di geo-localizzazione. 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. 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:

IGeolocationServiceListener:

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 riguardanti la posizione. Il contesto spaziale implementa l'interfaccia *ISpatialContext* che ne definisce la seguente struttura:

ISpatialContext:

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()* della 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 geo-localizzazione.
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.

Analogamente, quando l'ascoltatore riceve una notifica relativa all'inizio o alla fine di un movimento, recupera 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*. *Tale evento* viene notificato alla macchina virtuale tramite il metodo *notifyInputEnvEvent(InputEvent ev)*. A questo punto la *RespectVM* lo elaborerà valutando anche le reazioni innescate da esso.

```java
@Override
public AbstractGeolocationService getService() {
    return this.service;
}

@Override
public GeoServiceId getServiceId() {
    return this.service.getServiceId();
}

@Override
public TucsonTupleCentreId getTcId() {
    return this.tcId;
}

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

GeolocationServiceListener.java

GeolocationServiceManager

Il *GeolocationServiceManager* rappresenta l'entità responsabile della creazione, registrazione e rimozione dei servizi di geo-localizzazione.

Questa entità, definita adottando il pattern Singleton, incapsula una mappa chiave-valore, rappresentante la lista dei servizi di geo-localizzazione attivi nell'infrastruttura di coordinazione.

Inoltre fornisce i metodi necessari per:

- **creare servizi** sfruttando il metodo *createNodeService*. 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 tramite il metodo addService, il quale, preso in ingresso un *GeolocationService* si occupa di aggiungerlo alla lista dei servizi attivi.

- **rimuovere servizi** in modalità selettiva, tramite il metodo *destroyService(id)*, oppure globale, tramite il metodo *destroyAllServices()*.

- **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 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. E' possibile inoltre interrogarne i metodi, o invocarli, come se ci si trovasse in un normale flusso di esecuzione.

Il compito della creazione del servizio di geo-localizzazione viene assegnato al nodo TuCSoN. Questo delegherà l'azione al *GeolocationServiceManager*.

A questo punto sarebbe opportuno riportare anche il comportamento del sistema in fase di configurazione.

Avendo affrontato più avanti le problematiche di importazione del concetto di *situatedness* e relativi test, sezione in cui è fondamentale conoscere le dinamiche di avvio di un servizio di

geo-localizzazione ci si riserva di richiamare quella parte per descrivere il flusso di chiamate utili ad avviare e configurare il sistema.

Per concludere è necessario introdurre l'ultima modifica fatta.
Quanto mostrato fin ora riguarda la registrazione di servizi di geo-localizzazione ad un nodo TuCSoN, dunque permette di aggiornare la posizione della sola RespectVM. Poiché un agente potrebbe essere in esecuzione su un dispositivo nel quale la macchina virtuale non è in esecuzione, è necessario separare gli ambiti fornendo la possibilità di agganciare personali servizi di geo-localizzazione agli agenti TuCSoN, anche se un nodo non è in esecuzione.
A tale scopo sono stati aggiunti i seguenti metodi alla classe *ACCProxyAgentSide.java*.

```
public void attachGeolocationService(final String className,
        final TucsonTupleCentreId tcId) {
    final GeolocationServiceManager geolocationManager = GeolocationServiceManager
            .getGeolocationManager();
    if (geolocationManager.getServices().size() > 0) {
        final AbstractGeolocationService 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);
    }
}
```

```
private void createGeolocationService(final TucsonTupleCentreId tcId,
        final String className) {
    try {
        final int platform = PlatformUtils.getPlatform();
        final GeoServiceId sId = new GeoServiceId(this.aid.getAgentName()
                + "_GeoService");
        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 (final SecurityException e) {
        this.log("Error during service creation: " + e.getMessage());
    } catch (final NoSuchMethodException e) {
        this.log("Error during service creation: " + e.getMessage());
    } catch (final IllegalArgumentException e) {
        this.log("Error during service creation: " + e.getMessage());
    } catch (final InstantiationException e) {
        this.log("Error during service creation: " + e.getMessage());
    } catch (final IllegalAccessException e) {
        this.log("Error during service creation: " + e.getMessage());
    } catch (final InvocationTargetException e) {
        this.log("Error during service creation: " + e.getMessage());
    } catch (final ClassNotFoundException e) {
        this.log("Error during service creation: " + e.getMessage());
    }
}
```

```
public void setPosition(final IPlace place) {
    if (this.position != null) {
        this.position.setPlace(place);
    }
}
```

3.3.3 - Migrazione alla release aggiornata

Ora che è stato descritto il sistema di geo-localizzazione implementato sulla versione di TuCSoN obsoleta non resta che migrarlo sulla nuova versione adattando e correggendo eventuali dipendenze irrisolte.

SPAZIO

Il primo passaggio della migrazione consiste nell'iniettare il concetto di spazio all'interno del Framework aggiungendo il package:

"alice.respect.api.place"

e le rispettive classi:

- AbstractPhysicalPlace.java
- AbstractVirtualPlace.java
- DnsPlace.java
- IPlace.java
- IpPlace.java
- MapPlace.java
- OrgPlace.java
- PhPlace.java

Tali classi non necessitano di modifiche mentre la spiegazione concettuale viene rimandata al capitolo precedente.

POSIZIONE

In secondo luogo è stato iniettato il concetto di "posizione", modellata attraverso il package:

"alice.respect.api.geolocation"

e le rispettive classi:

- *GeoUtils.java*
- *Platforms.java*
- *PlatformUtils.java*
- *Position.java*
- *GeolocationConfigAgent.java*

Tali classi non necessitano di modifiche.

Da sottolineare che "*GeolocationConfigAgent.java"* non si limita a definire la posizione ma è una classe descritta in dettaglio successivamente, le classi *"Platform.java" e "PlatformUtils.java"* invece permettono l'adattamento a diverse piattaforme (Windows, Android, iOS….) mentre "*GeoUtils.java"* mette a disposizione metodi di conversione delle coordinate geografiche.

POSIZIONE NELL'EVENTO

Avendo inserito spazio e posizione si è provveduto a modificare le classi modellanti gli eventi in modo da renderle abili a contenere informazioni riguardanti tali concetti.

Inserendo un campo di tipo *"Position"* all'interno della classe astratta *"alice.tuplecentre.core.AbstractEvent.java"* diventa necessario modificare anche *"InputEvent.java"* e *"OutputEvent.java"* (all'interno dello stesso package) le quali effettivamente rappresentano eventi in ingresso ed in uscita e che estendono l'evento generico.

MESSAGGI

La trasmissione della posizione dovrebbe essere incapsulata all'interno di classi modellanti messaggi che vengono modificate a causa dell'approccio event-driven descritto in seguito. Tutta la parte riguardante quindi la modifica della comunicazione della posizione viene descritta nel capitolo successivo.

ACC

Il concetto di spazio e quindi di posizione è stato aggiunto all'ACCProxyAgentSide (*alice.tucson.service.ACCProxyAgentSide.java*), il quale deve gestire la memorizzazione e la trasmissione della propria posizione ed all'ACCProxyNodeSide (*alice.tucson.service.ACCProxyNodeSide.java)* il quale riceve, tramite apposito messaggio, la posizione dell'agente mittente.

CONTESTO

Il contesto *"spatial"* descritto precedentemente viene inserito aggiungendo le classi *alice.respect.api.ISpatialContext .java* e *alice.respect.core.SpatialContext.java* oltre all'apposito metodo in *RespectTC.java* utile a recuperare tale contesto.

NODO

La modifica soprastante non è in realtà sufficiente dato che è fondamentale iniettare il concetto di spazio anche all'interno del nodo stesso. Questo perché la posizione non riguarda solo un

eventuale agente mittente ma anche il nodo stesso che memorizza l'informazione all'interno della propria macchina virtuale.

Le modifiche apportate includono così un campo *"Position"* e relativi metodi *"getPosition()"* e *"setPosition(Position p)"* all'*AbstractRespectVM* (appartenente al package "*alice.tuplecentre.core*") e alla classe che lo implementa ovvero *"alice.respect.core.RespectVMContext.java"*.
Riassumendo un messaggio arriva all'ACCProxyNodeSide che chiama il TupleCentreContainer il quale passa il controllo alla RespectVM la quale fra i suoi compiti ha anche quello di poter salvare la posizione.

PREDICATI

Sono stati aggiunti i nuovi predicati descritti nel capitolo precedente, quindi la classe *alice.respect.api.Respect2PLibrary.java* è stata arricchita con "*current_place_2*", *"event_place_2"*, *"start_place_2"*.
L'AbstractTupleCentreVMContext è stato arricchito dei metodi *getDistanceTollerance()* e *setDistanceTollerance()* chiamati all'interno del RespectVMContext.

GUARDIE

Allo stesso modo sono state aggiunte le nuove guardie *"at_2"* e *"near_3"*

OPERAZIONI

I tipi di operazione from e to vengono aggiunti a *RespectOperation.java*

NOTE

Vengono aggiunti i metodi *findFromReactions()* e *findToReactions()* ad *"alice.respect.core.RespectVMContext"* così come l'estensione di codice spatial ai metodi *setReactionSpecHelper()* e *addReactionSpecHelper()*.
Causa aggiornamento alla nuova versione di TuCSoN le signature delle operazioni respect (*alice.respect.core.RespectOperation*) non prevedevano più la specifica del motore prolog come parametro di ingresso. Necessariamente quindi sono state aggiornate tutte le chiamate a quei metodi in particolare nell'*ACCProvider, ACCProxyNodeSide* e *ACCProxyAgentSide* così come è stata adattata la gestione delle eccezioni.

SERVIZI

Ora che sono state iniettate le classi modellanti i principali concetti relativi alla geo-localizzazione si prosegue importando nel progetto le classi astratte utili ad implementare quelli che saranno i servizi veri e propri di geo-localizzazione.

Il package specifico è:

"alice.respect.api.geolocation.service"

con le rispettive classi:

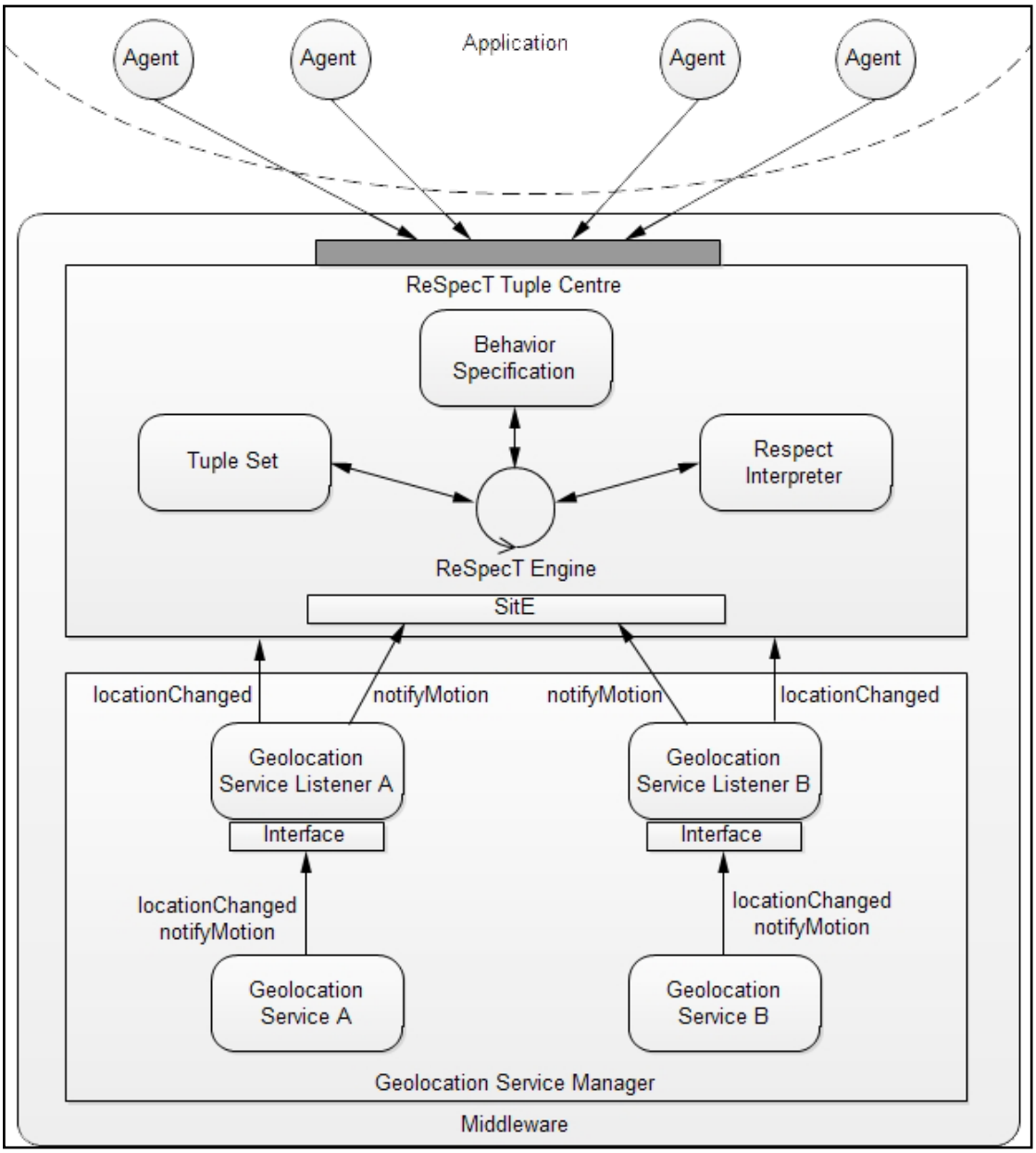
- AbstractGeolocationService.java
- AgentGeolocationServiceListener.java
- GeolocationServiceListener.java
- GeolocationServiceManager.java
- GeoServiceId.java
- IGeolocationService.java
- IGeolocationServiceListener.java

mentre le specifiche respect sono state aggiunte mediante il file

"alice.tucson.service.config.geolocation_spec.rsp".

NOTE

E' stata aggiunta la classe di supporto *"alice.tucson.network.NetworkUtils.java"* e sistemati diversi "import" di librerie così come la gestione delle eccezioni.



(Schema riassuntivo del servizio di geo-localizzazione [1])

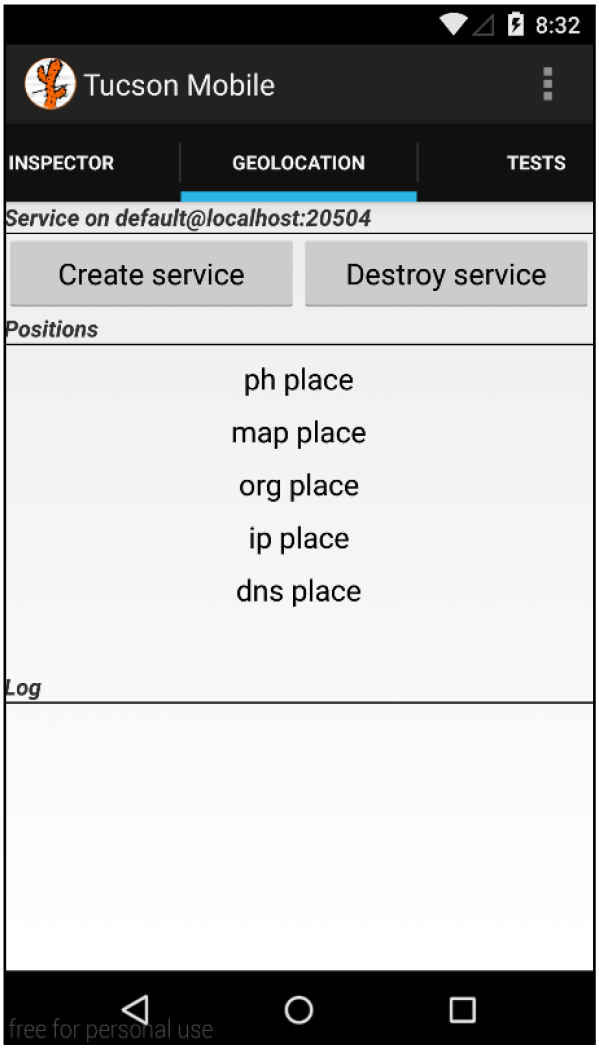
3.4 - La Geo-Localizzazione sull'App TuCSoN Android

Ora che l'ultima versione di TuCSoN è stata aggiornata, è stata integrata la geo-localizzazione ed i test terminano con esito positivo è possibile introdurre l'ultima parte del lavoro ovvero l'utilizzo della geo-localizzazione "Android side."

Al fine di una maggiore comprensione verrà descritto il flusso di chiamate utili ad avviare tale servizio, verrà mostrata la migrazione degli strumenti utili ad ottenere la posizione, le modifiche effettuate e l'App funzionante.

3.4.1 - Lo stato attuale

All'avvio dell'App e selezionato il Fragment relativo alla geo-localizzazione si presenta questa situazione:



L'Activity mostra il nodo e la porta su cui verrà istanziato il servizio di geo-localizzazione impostabile dal menù in alto a destra.

Prima di poter procedere con la creazione di tale servizio è necessario però avviare un *TucsonNodeService* mediante apposito Fragment.

Questo infatti, all'interno delle sue operazioni di inizializzazione predispone anche un centro di Tuple in attesa di ricevere un eventuale comando utile a configurare la geo-localizzazione.

In dettaglio:

ANDROID SIDE:

1. Click su ***"Start Tucson Node"*** del Fragment *"Tucson Node Service"*

NODE SIDE:

2. Viene chiamato ***TucsonNodeService.install()*** che avvia le operazioni utili a creare il nodo. Fra queste vi è la chiamata a *"setupGeolocationConfigTupleCentre()"* che deve configurare il sistema per renderlo in grado di istanziare un futuro servizio di geo-localizzazione.

3. "***TucsonNodeService.setupGeolocationConfigTupleCentre()***" crea quindi un centro di tuple ***"geolocationConfigTC"***, configurato tramite il file *"geolocation_spec.rsp"* che viene riportato per mostrare le reazioni messe a disposizione:

```
% ReSpecT specification of tuple centre geolocationConfigTC, used with geolocationConfigAgent for the configuration of the geolocation service.

reaction(
    out(boot),
    true,
    (
        in(boot),
        out(done)
    )
).

% Used to create a geolocation service.
%  Sid: service id
%  STclass: service transducer class path
%  Stcid: service tuple centre id
reaction(
    out( createGeolocationService(Sid,Sclass,Stcid) ),
    response,
    out( cmd(createGeolocationService) )
).

% Used to stop and destroy a geolocation service.
%  Sid: service id
reaction(
    out( destroyGeolocationService(Sid) ),
    response,
    out( cmd(destroyGeolocationService) )
).

```

Da sottolineare che *"createGeolocationService"* si aspetta tre parametri:

- Un service id
- La classe contenente l'implementazione del servizio di geo-localizzazione
- id del tuple centre

Infine viene chiamato "*TucsonNodeService.bootManagementAgents()*"

4. "***TucsonNodeService.bootManagementAgents()***" è il metodo utile ad avviare l'agente TuCSoN ***"geolocationConfigAgent"*** che si mette in attesa (tramite l'operazione *"in"*) sul *"geolocationConfigTC"* dell'arrivo di una tupla *"cmd(X)"*

Abbiamo un centro di tuple avviato sul nodo che può ricevere un' operazione "createGeolocationService(Sid, Sclass, Stcid)". Quando questa arriverà, la reazione associata prevede una out(cmd(createGeolocationService)) che inserisce la tupla cmd(createGeolocationService) nel suddetto centro di tuple. Parallelamente l'agente "geolocationConfigAgent", che è in attesa di una tupla (cmd(X)) (Con X=Operazione da avviare) si sbloccherà, recupererà l'operazione (createGeolocationService) e la invocherà lanciando il servizio di geo-localizzazione.

Una volta avviato quindi il Tucson node service, si torna al Geolocation Fragment.

ANDROID SIDE:

5. Click su ***"Create service"*** il quale avvia un *"CreateGeolocationServiceAgent"* su cui viene richiamata la *"createServices()"*.

6. ***"CreateGeolocationServiceAgent.createServices()"*** crea la tupla *"createGeolocationService(Sid, Sclass, Stcid)"* specificando soprattutto la classe che implementa il servizio di geolocalizzazione *(Sclass:it.unibo.tucson.android.geolocation.TucsonGeolocationService)* e la invia al centro di tuple "*geolocationConfigTC*" sbloccando il "*geolocationConfigAgent" i*n attesa.

NODE SIDE:

7. Ricevendo *"createGeolocationService(Sid, Sclass, Stcid)"* ed eseguendo la out(*createGeolocationService*) viene quindi sbloccato *"alice.respect.api.geolocation.GeolocationConfigAgent"* il quale esegue execCmd()

8. "***GeolocationConfigAgent.execCmd()***" si occupa di chiamare il *"GeolocationServiceManager.createNodeService()"* il quale istanzia la classe implementante il servizio di geo-localizzazione vero e proprio passata nella tupla ovvero *"it.unibo.tucson.android.geolocation.TucsonGeolocationService"*

ANDROID SIDE:

9. "***TucsonGeolocationService.start()***" avvia il "Service" Android *"TucsonLocationService"* che si occupa di gestire la geo-localizzazione mediante tecnologia specifica ovvero mediante API Android.

Il servizio è così avviato.

3.4.2 - Migrazione e modifiche

Per poter realizzare i passi sopra descritti si è proceduto riabilitando ed importando nel progetto di TuCSoN Android:

it.unibo.tucson.android.geolocation

e le relative classi:

- *AgentGeolocationService.java*
- *AgentLocationService.java*
- *CreateGeolocationServiceAgent.java*
- *DestroyGeolocationServiceAgent.java*
- *GeolocationFragment.java*
- *TucsonGeolocationService.java*
- *TucsonLocationService.java*

Nonostante le reazioni Respect siano implementate all'interno del codice è stato importato anche il file *geoService_spec.rsp* in modo da non generare eccezioni dovute alla sua mancanza.
Una volta aggiunto il caricamento del Fragment relativo alla geo-localizzazione all'interno della *MainActivity.java* sono state riabilitate le stringhe all'interno del file *strings.xml*.
Giunti a questo punto la parte grafica relativa alla geo-localizzazione è stata riabilitata.
Avviando i servizi venivano mostrati due malfunzionamenti importanti.

Il primo si rifaceva al fatto che non veniva visualizzata la via corrispondente alle coordinate locali.
Inizialmente si presupponeva potesse essere un problema di chiave fornita da Google per accedere ai servizi di geo-localizzazione obsoleta.

In realtà non è necessaria nessuna chiave, il problema era unicamente riconducibile al simulatore. Provando l'applicazione su dispositivo fisico tutto va a buon fine.

Il secondo risiedeva nel fatto che quando il servizio veniva avviato, veniva mostrata l'ultima posizione agganciata dal GPS tramite il metodo *"locationManager.getLastKnownLocation(provider)"* di *TucsonLocationService.java*. Questo metodo permette quindi di velocizzare la visualizzazione della posizione, attendendo un po' di tempo viene invece mostrata la posizione reale.
E' chiaro che la prima posizione visualizzata potrebbe essere errata soprattutto se questa viene richiesta in due luoghi diversi.

esempio: Avviamento dell'App Tucson nel luogo A con visualizzazione indirizzo A. Terminazione dell'app e spostamento a luogo B. Avviamento dell'app di Tucson ma l'indirizzo visualizzato è, inizialmente, indirizzo A. Dopo un po di tempo, quando il dispositivo aggancia il satellite, si riceve l'aggiornamento mostrando l'indirizzo B.

Per ovviare all'inconveniente si è scelto di far attendere l'utente mediante un messaggio di log ma di garantire sempre la posizione corretta.
(Avviando il servizio all'esterno la ricerca è ovviamente molto più breve)

```
@Override
public int onStartCommand(final Intent intent, final int flags,
        final int startId) {
    first = true;
    this.serviceId = intent.getStringExtra("serviceId");
    this.postLog("Received start id " + startId + ": " + intent);
    this.postLog("Starting location service");
    // Getting LocationManager object from System Service LOCATION_SERVICE
    this.locationManager = (LocationManager) this
            .getSystemService(Context.LOCATION_SERVICE);
    // Creating a criteria object to retrieve provider
    final Criteria criteria = new Criteria();
    // Getting the name of the best provider
   String provider = this.locationManager.getBestProvider(criteria,
            true);
   this.postLog("Location service searching position. Wait...the process is faster outdoor");

    this.locationManager.requestLocationUpdates(provider,
            TucsonLocationService.MIN_TIME_INTERVAL,
            TucsonLocationService.MIN_DIST_INTERVAL, this);
    return Service.START_STICKY;
}
```

Infine è stato modificato il codice con cui ricavare l'indirizzo IP dalla rete dato che poteva essere interrogata una risorsa di rete non presente causando eccezioni. Ora il sistema tenta di ricavare l'IPv4 se presente altrimenti l'IPv6.

Il risultato alla pressione di "Create Service":

3.4.3 - Test

I test riguardanti la geo-localizzazione sono stati gradualmente inseriti in

it.unibo.tucson.android.tests.TestAgent.java

apportando tutte le modifiche necessarie all'adattamento e risolvendo le dipendenze inconsistenti.
Questi comprendono:

- **CURRENT_PLACE**

  Tenta di recuperare la posizione fisica assoluta del dispositivo ospitante il centro di tuple tramite il predicato *current_place*.
  (COMPLETATO)

- **START_PLACE**

  Tenta di recuperare la posizione fisica assoluta del nodo dove la catena di eventi è stata originata tramite il predicato *start_place*.
  (COMPLETATO)

- **EVENT_PLACE**

  Testa il predicato event_place.
  (COMPLETATO)

- **AT**

  Testa tramite la guardia *"at"* se l'agente sta eseguendo nella stessa posizione in cui il nodo è in esecuzione.
  Da segnalare l'aggiunta della chiamata al metodo *toTerm()* in fase di trasmissione delle tuple per ottenere ed inviare il valore delle coordinate
  (COMPLETATO)

- **NEAR**

  Testa tramite la guardia *"near"* se l'agente sta eseguendo vicino alla posizione in cui il nodo è in esecuzione.
  Da segnalare l'aggiunta della chiamata al metodo *toTerm()* in fase di trasmissione delle tuple per ottenere ed inviare il valore delle coordinate
  (COMPLETATO)

- **SPATIAL_LOG**

  Testa il log relativo alla geo-localizzazione.

  (COMPLETATO)

- **MOTION_LOG**

  Simula un semplice log di movimento che include il tempo di start / arrivo e la posizione. Sono state modificate alcune definizioni di reazioni e le relative chiamate.

  (COMPLETATO)

Da non dimenticare la migrazione del codice utile a creare ed agganciare un servizio di geo-localizzazione ad un agente di test in:

it.unibo.tucson.android.tests.TestAgent

```
// Attaching geolocation service to the agent
TucsonTupleCentreId tcId;
this.acc = this.getContext();
try {
    tcId = new TucsonTupleCentreId(this.myName()
            + "_geolocation_service", "localhost",
            String.valueOf(this.port));
    ((ACCProxyAgentSide) this.acc)
    .attachGeolocationService(
            "it.unibo.tucson.android.geolocation.AgentGeolocationService",
            tcId);
} catch (final TucsonInvalidTupleCentreIdException e) {
    Utils.postError("", "creating geolocation service -> "
            + "TucsonInvalidTupleCentreIdException", messageHandler);
}
```

Questo utilizza, per creare servizi di geo-localizzazione, le classi *AgentGeolocationService.java* e *AgentLocationService.java* che sono state opportunamente modificate come *TucsonGeolocationService.java* e *TucsonLocationservice.java* già descritte.

Un ultima nota, oltre al completamento dei test, riguarda la possibilità di migliorare le notifiche di aggiornamento della posizione che probabilmente causano rallentamenti al sistema.
Realizzando chiamate asincrone e disaccoppiando il thread principale da queste non si dovrebbero più riscontrare problemi di performance.

Terminata la migrazione della geo-localizzazione il lavoro è proseguito completando la conversione del sistema al modello event-driven.

CAPITOLO 4 - AGGIORNAMENTO TuCSoN: MODELLO EVENT-DRIVEN

Il lavoro di allineamento dell'App e di migrazione della geo-localizzazione ha introdotto un altro aspetto di importanza rilevante. Originariamente TuCSoN era un sistema basato su una comunicazione a scambio di messaggi i quali contenevano informazioni discriminanti l'operazione da invocare e l'agente mittente. L'operazione vera e propria veniva creata nel momento in cui il messaggio giungeva dall'*ACCProxyAgentSide* all'*ACCProxyNodeSide* mentre si parlava di "evento" solo quando il flusso delle operazioni raggiungeva la chiamata al metodo *"doOperation()"* dell'*AbstractTupleCentreVMContext*.
In seguito questo approccio è stato modificato. Per questo motivo e per poter permettere alla macchina virtuale di accedere alle informazioni riguardanti la posizione non è stato scelto di inserire quest'ultima direttamente nel TucsonMsgRequest ma di modificare in modo più invasivo il flusso, rendendo però la piattaforma completamente event-driven. L'inefficienza nella serializzazione dell'evento, dato che si lavora con protocollo TCP, è stata aggirata creando tale evento sia *AgentSide* sia *NodeSide* trasmettendo solo le informazioni utili.
Qui di seguito verranno quindi riportate le principali caratteristiche del modello event-driven, il primo lavoro di conversione e quindi lo stato attuale per arrivare alle modifiche dell'aggiornamento oggetto di tesi e relativi test.

4.1 - Il modello event-driven

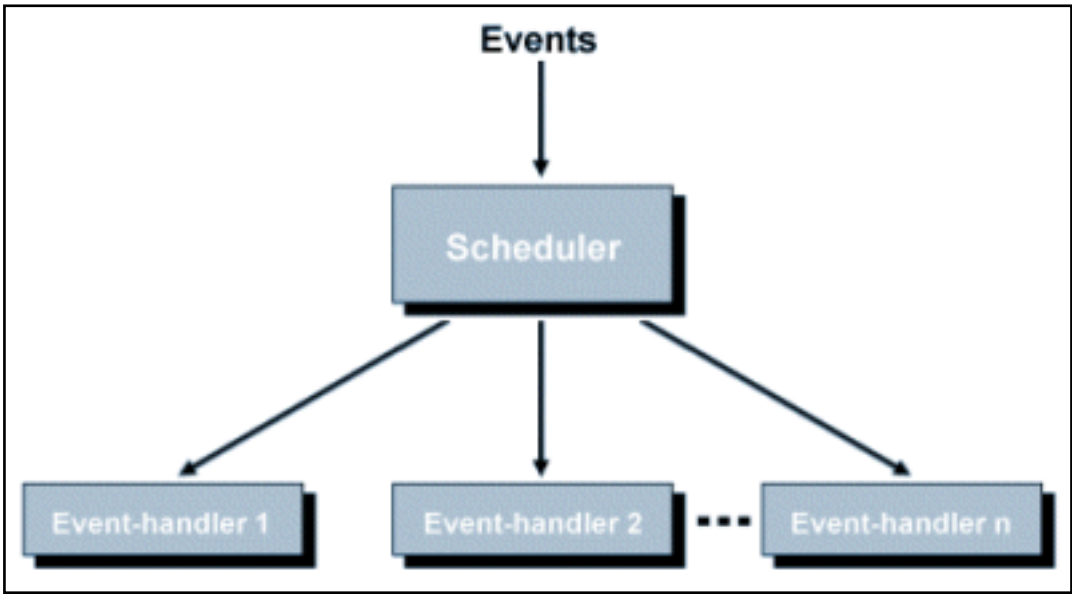
La programmazione a eventi rientra nei paradigmi di programmazione informatici [4]. Mentre in un programma tradizionale l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore, nei programmi scritti utilizzando la tecnica a eventi il flusso del programma è determinato dal verificarsi di eventi esterni.

La tecnica consiste nel non aspettare che un'istruzione impartisca al programma il comando di elaborare una certa informazione ma il sistema è predisposto per eseguire all'infinito un loop di istruzioni all'interno del quale si verifica continuamente la comparsa dell'evento contenente le informazioni da elaborare (potrebbe trattarsi della creazione di un file in una certa cartella o della pressione di un tasto del mouse o della tastiera). Solo in questo caso si lancia l'esecuzione della parte di programma relativa alla gestione dell'evento in questione. Programmare "a eventi"

significa, quindi, ridefinire in modo personalizzato le azioni con cui il sistema risponde al verificarsi di un certo evento.

Gli eventi esterni a cui il programma deve reagire possono essere rilevati mediante polling all'interno di un loop di programma, oppure in risposta ad un interrupt. Di norma le applicazioni usano una combinazione di entrambe queste due tecniche.

I programmi che utilizzano la programmazione a eventi sono quindi composti tipicamente da diversi brevi sotto-programmi, chiamati gestori o event handlers, eseguiti in risposta agli eventi esterni e da un dispatcher, che effettua la chiamata, spesso utilizzando una coda che contiene l'elenco degli eventi già verificatisi, ma non ancora "processati". Capita anche che i gestori degli eventi possano, al loro interno, innescare ("trigger") altri eventi, producendo una cascata di eventi.



[7]

Tale paradigma incoraggia l'utilizzo di tecniche di programmazione flessibili ed asincrone, e si basa sul principio di imporre al programmatore meno vincoli possibile ("modeless"). I programmi dotati di interfaccia grafica ad esempio sono tipicamente realizzati secondo l'approccio event-driven. Anche i sistemi operativi sono un classico esempio di programmi event-driven su almeno due livelli. Al livello più basso gli handler gestiscono gli eventi innescati dall'hardware, con il processore principale che agisce da dispatcher. Ad un livello superiore i sistemi operativi fungono da dispatcher per i processi attivi, passando dati ed, eventualmente, altri interrupt ad altri processi, che possono, a loro volta, gestire ad un livello più alto gli stessi eventi.

L'esempio riportato sotto, in *pseudocodice*, fornisce un esempio di lettura dati da un socket eseguita con tecnica event-driven [7]:

```
function read_next_data(fd)
   data = read_async( fd )
   if len(data) = 0
       => Niente da leggere, registrati per la prossima lettura
       event_polling_register( fd, read_next_data )
       => Torna indietro per fare qualcos'altro
   else
       => Dati disponibili, ricevuta quantità dati pari a: len(data)
       add_data_to_buffer( buffer, data )
   endif
```

4.2 - TuCSoN: Lo stato attuale

Per poter realizzare il modello sopra descritto, TuCSoN prevede due nuove entità rappresentate dalle classi *InputEventMsg* e *OutputEventMsg*. Queste rappresentano, rispettivamente, un evento generato per una richiesta effettuata da un agente e per una risposta da parte della macchina virtuale. Entrambe le entità rappresentano una forma serializzabile di evento ed incapsulano le informazioni, incluse quella sullo spazio, necessarie per la ricostruzione dell'evento vero e proprio.

La struttura:

```
public InputEventMsg(final String s, final long oid, final int opt,
        final LogicTuple lt, final String trg, final long t,
        final Position p) {
    this.source = s;
    this.opId = oid;
    this.opType = opt;
    this.tuple = lt;
    this.target = trg;
    this.reactingTC = trg;
    this.time = t;
    this.place = p;
}
```

InputEventMsg.java

```
public OutputEventMsg(final long i, final int t, final boolean a,
        final boolean s, final boolean ok) {
    this.opId = i;
    this.opType = t;
    this.allowed = a;
    this.success = s;
    this.reqTuple = null;
    this.resTuple = null;
    this.resultSuccess = ok;
}
```

OutputEventMsg.java

Ovviamente sono state modificate anche le classi *TucsonMsgRequest.java* e *TucsonMsgReply.java* al fine di incapsulare le due classi sopra citate.
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.
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.
A questo scopo era 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 il motore Prolog da utilizzare, il tipo di operazione richiesta e la tupla (o il template) specificato.
Nella nuova versione di Tucson il metodo makeOperation viene spostato su una nuova classe:
alice.tucson.service.OperationHandler
in cui non viene più dichiarato il motore prolog.

4.3 - TuCSoN: Aggiornamento

Ora che è stata presentata la situazione attuale non resta che iniettare tali componenti sulla nuova versione di TuCSoN adattando e correggendo eventuali dipendenze irrisolte.

MESSAGGI

E' facile intuire che dovendo agire sulla comunicazione le prime entità modificate sono state i messaggi.

Una volta aggiunti nel package *alice.tucson.service* i nuovi

- *InputEventMsg*
- *OutputEventMsg*

si è proceduto nella modifica di *TucsonMsgReply.java* e *TucsonMsgRequest.java* (package: alice.tucson.network) per permettergli di incapsulare le due entità sopra citate contenenti l'evento in trasmissione. A fronte di questo sono state aggiornate anche tutte le chiamate ai metodi risultate ora inconsistenti.

Modificati i messaggi si è passati alla modifica dei mittenti e riceventi di questi ovvero gli ACC, lato agente e lato nodo.

ACC

L'intervento sull'*ACCProxyAgentSide* ha riguardato la creazione di un InputEventMsg tramite le relative informazioni e l'incapsulamento di questo all'interno di un TucsonMsgRequest. L'operazione non ha causato problemi evidenti.

```
final InputEventMsg ev = new InputEventMsg(this.aid.toString(),
        op.getId(), op.getType(), op.getLogicTupleArgument(), null,
        System.currentTimeMillis(), this.getPosition());
exit = new TucsonMsgRequest(ev);
try {
    info.sendMsgRequest(exit);
} catch (final DialogException e) {
    e.printStackTrace();
}
```

L'intervento sull'*ACCProxyNodeSide* ha riguardato la gestione di un InputEventMsg in entrata, l'invocazione del metodo *makeOperation* utile a richiamare l'operazione opportuna definita nella classe *RespectOperation,* la creazione di un nuovo evento di input mediante le informazioni ricevute e la creazione di un *TucsonMsgReply* contenente un *OutputEventMsg*. Sono state infine aggiornate tutte le chiamate a metodi inconsistenti e la gestione delle eccezioni aggiornando anche *InterTupleCentreACCProxy.java*

TUPLECENTRECONTAINER

Una volta arrivato all'ACCProxyNodeSide l'evento viene creato e passato al TupleCentreContainer.
Questo è stato modificato a fronte della nuova gestione dei messaggi.

RESPECT

L'azione sulle api respect ha portato alla modifica, nel package *alice.respect.api* delle interfacce:

- I*RespectTC.java*
- *IOrdinaryAsynchInterface.java*
- *IOrdinarySynchInterface.java*
- *ISpecificationAsynchInterface.java*
- *ISpecificationSynchInterface.java*

e relative implementazioni:

- *RespectTC.java*
- *OrdinaryAsynchInterface.java*
- *OrdinarySynchInterface.java*
- *SpecificationAsynchInterface.java*
- *SpecificationSynchInterface.java*

Tutti i metodi contenuti all'interno di tali classi non ricevono più parametri di input come l'id agente o direttamente la tupla ma prevedono direttamente l'inputEvent come parametro di ingresso.

CONTESTI

Il timed context viene aggiornato per supportare gli eventi: *alice.respect.api.ITimedContext* e quindi *alice.respect.core.TimedContext*.
La gestione degli eventi è stata aggiunta anche a
"*alice.tucson.introspection.InspectorContextSkel.java"* senza modifiche rilevanti.

SPAWNACTIVITY

La gestione degli eventi è stata aggiunta anche nelle classi riguardanti la computazione parallela "*alice.tucson.api.AbstractSpawnActivity.java"* ridefinendo come sopra la chiamata al metodo *"make"* e la creazione di un InputEvent così come all'interno di "*alice.tucson.service.Spawn2PLibrary.java*" e "*alice.tucson.service.ObservationService.java*".

TUCSONNODESERVICE

Le modifiche sono proseguite adattando il *TucsonNodeService.java* (package *alice.tucson.service*) ed il *NodeManagementAgent.java* modificando anche le chiamate al metodo *enablePersistency()* aggiunto alla classe *TupleCentreContainer.java*.
Semplici modifiche alle chiamate di metodi utili all'ottenere dati relativi agli eventi sono state apportate anche a *OperationHandler.java* appartenente allo stesso package.

SITUATEDNESS

La gestione della situatedness verrà affrontata successivamente, al momento è stato solamente eliminato il vecchio TucsonMsgRequest adattandolo agli eventi in
alice.respect.situatedness.AbstractTransducer.java

Una volta importati tutti i componenti, adattati e risolti tutti gli errori di compilazione si è proceduto all'esecuzione dei test.

4.4 - Test

Il sistema TuCSoN prevede diversi test "embeddati" (*alice.tucson.examples*) in grado di verificarne le funzionalità basilari e specifiche.
Avendo effettuato modifiche invasive è necessario eseguire tutti i test controllandone l'esito positivo.
L'ordine in cui questi sono stati eseguiti prevede:

1. *HelloWorld*
2. *PrologHelloWorld*
3. *HelloWorldJTuple*
4. *DiningPhilos*
5. *TimedDiningPhilos*
6. *AsynchAPI*
7. *SpawnedWorkers*
8. *PrologSpawnedWorker*
9. *DistributedDiningPhilos*
10. *rbac*

Inoltre, nella versione aggiornata di TuCSoN, esistono due test aggiuntivi:

11. *Persistency*
12. *Situatedness*

HelloWorld

Test basilare. Porta a termine una "*out*" ed una "*read*".
(COMPLETATO)

HelloWorldJTuple

(COMPLETATO)

PrologHelloWorld

Analogo al precedente ma utilizza un agente TuCSoN prolog
(COMPLETATO)

DiningPhilos

Classico test di coordinazione che prevede N filosofi i quali possono “pensare” o “mangiare”.

Non è stato necessario apportare modifiche radicali.

(COMPLETATO)

TimedDiningPhilosophers

Classico test di coordinazione che prevede N filosofi i quali possono “pensare” o “mangiare”. In questo caso viene scambiato con il centro di tuple anche il tempo massimo in cui un filosofo può “mangiare” ovvero occupare il “lock”.

Non è stato necessario apportare modifiche radicali.

(COMPLETATO)

asynchAPI

Le primitive per accedere al centro di tuple locale sono testate in maniera asincrona introducendo un elemento di non determinismo.

(COMPLETATO)

SpawnedWorkers

Implementazione dell’architettura Master/Worker che utilizza il nodo TuCSoN come uno spazio di memoria condiviso. Il Master delega al Worker la computazione di alcuni fattoriali. L’unica modifica apportata riguarda un controllo in *alice.respect.core.RespectVMContext.java* in cui era necessario verificare che *“targetTC”* fosse istanza di *“TucsonTupleCentreId”*.

(COMPLETATO)

PrologSpawnedWorkers

L’unico problema era lo stesso del test sopra

(COMPLETATO)

DistributedDiningPhilosophers

Classico test di coordinazione che prevede N filosofi i quali possono “pensare” o “mangiare” eseguito su più nodi.

L’unico problema era lo stesso dei test sopra

(COMPLETATO)

Rbac:

Classico test di sicurezza adottato per la verifica delle credenziali degli agenti che interagiscono con il nodo TuCSoN.

(FALLITO)

Come si può notare non sono emersi particolari problemi fino al test chiamato *"rbac"*.

Tale test dovrebbe occuparsi di lanciare tre agenti:

l'agente administrator che si autentica per cambiare alcune proprietà RBAC, un agente autorizzato a cui è permesso eseguire alcuni ruoli predefiniti ed un agente non autorizzato al quale è consentito unicamente il ruolo di default senza che sia necessario il login.

Quest'ultimo agente dovrebbe eseguire con successo una *"rd"*, poi fallire una *"out"*, e infine fallire tentando di assumere il ruolo *"roleWrite"*.

Il risultato del test invece mostrava il fallimento immediato della prima "rd". Una prima analisi effettuata con il supporto del debug rivelava come l'UnauthorisedAgent, pur dovendo agire secondo il ruolo specificato, cercava di ottenere l'acc in:

EnhancedACC acc = negACC.playRoleWithPermissions(permissions);

ma faceva scattare un eccezione dato che la lista di "policies", inizializzata all'avvio del test, risultava vuota.

Portando avanti altre indagini si scopriva che lo stesso test funzionava su alcuni pc mentre falliva su altri anche a parità di sistema operativo.

Il confronto fra due log delle operazioni, uno appartenente al test non funzionante ed uno al test funzionante, mostrava una singola differenza.

Ad un certo punto della catena di invocazioni/reazioni (comprese tra *"Installing RBAC…"* e *"RBAC installed…" dell'AdminAgent*), vi è una *"inp(set_basic_agent_class)"* che scatena una sola reazione ReSpecT (contenuta nel file *boot_spec.rsp* in *alice.tucson.service.config*).

Questa reazione, come penultima istruzione, esegue una *"out(change_roles_class)"* che dovrebbe scatenare una delle due reazioni seguenti. Nel caso di test non superato questa fase non andava a buon fine perché la guardia "internal" falliva di conseguenza non venivano eseguite tutte le istruzioni relative all'impostazione dei ruoli causando malfunzionamenti nei test.

Al fine di scoprirne il motivo si poteva:

- inserire delle stampe nella reaction ReSpect *"inp(set_basic_agent_class)"*, con una tecnica multi-paradigm per osservare meglio il valore assunto dalle variabili usate fino a quel momento.

Tale tecnica prevede l'utilizzo delle seguenti linee di codice:

class('java.lang.System') . out <- get(Out),
Out <- println(Name)

per poter avere una sorta di log all'interno delle reazioni Respect.

- inserire delle stampe nelle guardie *"from_tc"*, *"to_tc"*, *"endo"* e *"intra"* (guardie richiamate da *"internal"*) nella classe *Respect2PLibrary.java* in *alice.respect.api*.

Essendo la seconda tecnica più veloce (mantiene intatta la possibilità di usare il debugger di eclipse e quindi di seguire passo passo il flusso di esecuzione) la ricerca è proseguita verificando le guardie.

Il risultato mostrava:
- from_tc_0 -> TRUE (ok)
- to_tc_0 -> TRUE (ok)
- endo_0 -> !exo_0 -> TRUE (error)
- intra_0 -> FALSE (error)

Due guardie, incaricate di verificare la corrispondenza dell'IP mittente e ricevente, fallivano. Oltre a questo si riscontrava un'altra anomalia: disabilitando il wifi sulla macchina fisica il test andava a buon fine.
Ciò che scatenava l'errore era da ricollegarsi al metodo in cui *l'ACCProxyAgent* recuperava l'indirizzo fisico della macchina da inviare.

AdminAgent per creare l'*AdminACCProxyAgentSide* chiama il seguente metodo:

```
AdminACC adminACC = TucsonMetaACC.getAdminContext(
        this.getTucsonAgentId(), "localhost", 20504, "admin", "psw");
```

ma nel costruttore di *AdminACCProxyAgentSide* il relativo *tid* viene impostato recuperando l'Ip della macchina mediante:
localhost = InetAddress.getLocalHost();
Questo metodo scatenava il problema.

Il suo funzionamento infatti consisteva nel recuperare l'ip di una delle interfacce di rete presenti sul sistema, ma poteva capitare che, pur se il sistema era eseguito in locale, l'IP restituito non fosse *localhost* (indirizzo della macchina) ma:

- Se la macchina era connessa al wifi veniva restituito l'indirizzo di rete attuale
- Se la macchina non era connessa veniva restituito l'indirizzo della prima interfaccia disponibile, nel caso specifico "*vboxnet0*".

A questo punto il messaggio veniva spedito dall'ACCProxyAgentSide al centro di tuple, incapsulando come IP sorgente uno dei due citati sopra (a seconda del caso).

Alla ricezione del messaggio da parte del centro di tuple, quando veniva eseguita la reazione, la guardia "*internal*" chiamava fra le altre *exo_0* e *intra_0*. Entrambe controllavano che l'IP dell'agente chiamante e l'Ip del centro di tuple coincidessero.
Ovviamente questo non accadeva dato che il centro di tuple doveva essere eseguito su *localhost* mentre il chiamante sugli IP citati prima quindi si spiegava il fallimento della guardia.
Perché allora a WIFI disattivato gli IP dovevano essere diversi ma la guardia non falliva?
In sintesi, nel flusso di operazioni sul nodo che controllano la coincidenza o meno degli IP chiamanti e destinatari ci sono vari metodi fra i quali *Respect2PLibrary.checkIP(source)* il quale permette di far andare a buon fine la suddetta guardia non solo se gli IP coincidono ma anche se l' IP del chiamante corrisponde alla prima interfaccia di rete disponibile. La prima interfaccia di rete disponibile era "*vboxnet0*" che effettivamente coincideva al chiamante in caso non fosse presente la connessione wifi mandando tutto quindi a buon fine.
La soluzione è consistita quindi nel modificare il recupero dell'Ip da parte dell'AdminAgent.
Al momento il sistema funziona in modo che se *AdminProxyAgentSide* ha "localhost" o "127.0.0.1" come parametri di input nel costruttore (sistema che gira in locale) questi vengono presi così come sono nella creazione del *TucsonTupleCentreId* evitando di chiamare "*InetAddress.getLocalHost();*" che viene invece chiamato in caso di "sistema distribuito" dato che risulta necessario avere gli indirizzi IP effettivi.

Questa soluzione potrebbe però creare ancora confusione:

1. Nel caso il metodo non si comporti sempre come nelle macchine test utilizzate, ovvero prendendo l'IP dell'interfaccia di rete attiva (es: wifi, quando connesso).

2. Se due interfacce di rete sono attive contemporaneamente. In questo caso ad esempio viene probabilmente presa la prima in ordine di priorità la quale potrebbe non essere quella giusta.

Potrebbe quindi essere meglio evitare tali chiamate ed ottenere i dati necessari direttamente da input come avviene nel caso “localhost". Il quesito rimane in sospeso e verrà ripreso in futuro.

```
if (!tmpNode.equals("localhost")) {
    if (!tmpNode.equals("127.0.0.1")) {
        InetAddress localhost;
        try {
            localhost = InetAddress.getLocalHost();
            final String localNodeAddress = localhost.getHostAddress();
            tmpNode = localNodeAddress;
        } catch (final UnknownHostException e) {
            return new TucsonTupleCentreId(ACCProxyAgentSide.TC_ORG, "'"
                    + tmpNode + "'", "" + tmpPort);
        }
    }
}
```

Il test in ogni caso termina con successo.

Sistemato rbac, è stata testata la persistenza:

alice.tucson.example.persistence

che risulta andare a buon fine causando un eccezione solo in fase di chiusura del test.

La migrazione dei concetti event-driven alla nuova release TuCSoN non ha comportato modifiche sostanziali al lavoro precedentemente effettuato ma diverso è il discorso riguardante la parte di “*situatedness*” la quale non era ancora stata sviluppata nella vecchia versione.

Prima di descriverne il testing però è necessario introdurne i concetti teorici.

CAPITOLO 5 - AGGIORNAMENTO TuCSoN: SITUATEDNESS

Fino a questo punto è stato descritto il lavoro di migrazione dei concetti di geo-localizzazione e conversione al modello event-driven della versione più aggiornata di TuCSoN. Tale versione però conteneva già una parte implementante la proprietà di "situatedness" ovvero la capacità di un sistema di essere immerso in un ambiente reale ed averne la percezione, oltre ad un modo con cui interagirvi. Proprio quest'ultimo punto comporta la necessità di fornire agli agenti la possibilità di cogliere eventi da un ambiente in maniera reattiva.
Procedendo nell'aggiornamento quindi viene toccata anche questa funzionalità ed avanzando per gradi verrà presentata la situazione iniziale, le modifiche effettuate ed i test necessari.

5.1 - Descrizione

[8] Nel corso degli anni il linguaggio di coordinazione ReSpecT ha subito diversi aggiornamenti. Un estensione importante è riconosciuta nell'implementazione dei concetti di tempo, spazio e ambiente. Prima di considerare ReSpecT come un sistema immerso in un ambiente però, era necessario garantire che i singoli componenti potessero dialogare in termini di spazio (nuovi predicati visti, gestione dei luoghi in cui si verificano gli eventi) e di tempo (quando si sono verificati gli eventi, cogliere il tempo attuale, innescare reazioni dopo un certo tempo).
Avendo già descritto precedentemente i concetti di spazio al fine della geo-localizzazione ed essendo la gestione del tempo di secondario interesse per quel che riguarda questa tesi, verrà riportato solo un breve riassunto del primo concetto per poi introdurre l'architettura utile a modellare l'ambiente.

LO SPAZIO IN TuCSoN E ReSpecT

Dovendo includere la nozione di *situatedness*, il sistema dovrà necessariamente interagire con delle risorse ambientali (virtuali o concrete) dotate di precisa posizione.
Come detto il concetto di locazione è distinto in spazio virtuale e spazio fisico, viene differenziato cioè, a titolo esemplificativo, il nodo di rete dove un TuCSoN node sta operando dall'ubicazione fisica della macchina che ospita il sistema.

Tale distinzione permette di identificare i diversi nodi in una rete ma anche, introducendo il tema “situatedness", di identificare il luogo fisico di ubicazione di un sensore o attuatore che interagisce col sistema avendone informazioni dettagliate riguardo la sua posizione.
E’ stato già mostrato che, introdotti tali concetti all’interno di un sistema TuCSoN, è necessario definire un ulteriore estensione di ReSpecT per permettere ai singoli componenti del sistema di dialogare riguardo lo spazio sia fisico che virtuale di un altro componente. Sono stati quindi definiti nuovi eventi ammissibili, in particolare *from(Place)* e *to(Place)*, indicando con *Place* l’ubicazione fisica, e *node(Node)* per gestire la locazione virtuale. Assieme a questi eventi, vengono definiti nuovi predicati di osservazione e nuove guardie che dovranno fornire controlli sia sulla locazione virtuale che su quella fisica:

- **at(@Place)** verificata se l’evento si manifesta nella locazione fisica definita da Place;
- **near(@Place, @Radius)** verificata se l’evento si manifesta in prossimità della locazione fisica definita da Place entro un raggio definito da Radius;
- **on(@Node)** verificata se l’evento si manifesta nella locazione virtuale definita da Node.

Definito lo spazio e tralasciato momentaneamente il tempo viene quindi affrontato l’ambiente.

L’AMBIENTE IN TuCSoN E ReSpecT

Come già analizzato precedentemente, tentando di inserire un sistema TuCSoN in un ambiente, si palesa la necessità di fornire metodologie di comunicazione tra i componenti del sistema e i componenti dell’ambiente. Questo significa fare in modo che un tuple centre possa cogliere eventi ambientali e fare da intermediario tra i vari componenti e le risorse ambientali. Bisogna quindi estendere nuovamente il linguaggio ReSpecT per la gestione di queste nuove tipologie di eventi e interazioni.
Il metodo è analogo al concetto di tempo e spazio: si iniettano nuovi predicati di osservazione e nuove guardie per poter gestire le nuove tipologie di eventi. In particolare, si definisce la proprietà ambientale

env(Key, Value)

identificata da un nome *“key”* e da un valore *“value”*. Questa proprietà viene osservata tramite i predicati come *current_env(?Key, ?Value)*, *event_env(?Key, ?Value)* e *start_env(?Key, ?Value)*.

A questi si aggiungono due nuove guardie relative alla direzione degli eventi:

- ***from_env*** verificata quando l'evento proviene da una risorsa ambientale;
- ***to_env*** verificata quando l'evento è destinato ad una risorsa ambientale.

A questo punto si definisce un metodo per poter leggere dei valori dai sensori e quindi ottenere delle proprietà:

getEnv(Key, Value)

Il principio di utilizzo prevede la richiesta di una *getEnv* da parte di un tuple centre ad una risorsa la quale, una volta ricevuta la richiesta, ritornerà a sua volta il valore del parametro key.
Definito l'evento, diviene possibile programmare il tuple centre con reazioni alle notifiche ambientali del tipo:

reaction(getEnv(key,Value), from env, out(data(Value)))

In questo modo il sistema è reattivo agli eventi ambientali sfruttando i principi di ReSpecT.
Per quanto riguarda l'utilizzo di *getEnv(Key,Value)* da parte di un agente, per motivi di correttezza concettuale è stato scelto di renderlo possibile a discrezione del tuple centre. Quest'ultimo infatti può permettere o meno la richiesta di una getEnv agli agenti che comunicano con esso.

L'altro metodo definito è un metodo utile a comunicare un valore ad un attuatore, forzandone quindi un parametro. Per poter comunicare dunque con gli attuatori presenti nell'ambiente, viene definito l'evento

setEnv(Key,Value)

Anche in questo caso, agli agenti può essere concessa o meno la possibilità di richiedere una *setEnv* che nel caso verrà eseguita dal tuple centre.
Essendo un nuovo evento, può anch'esso diventare oggetto di reazioni del tipo:

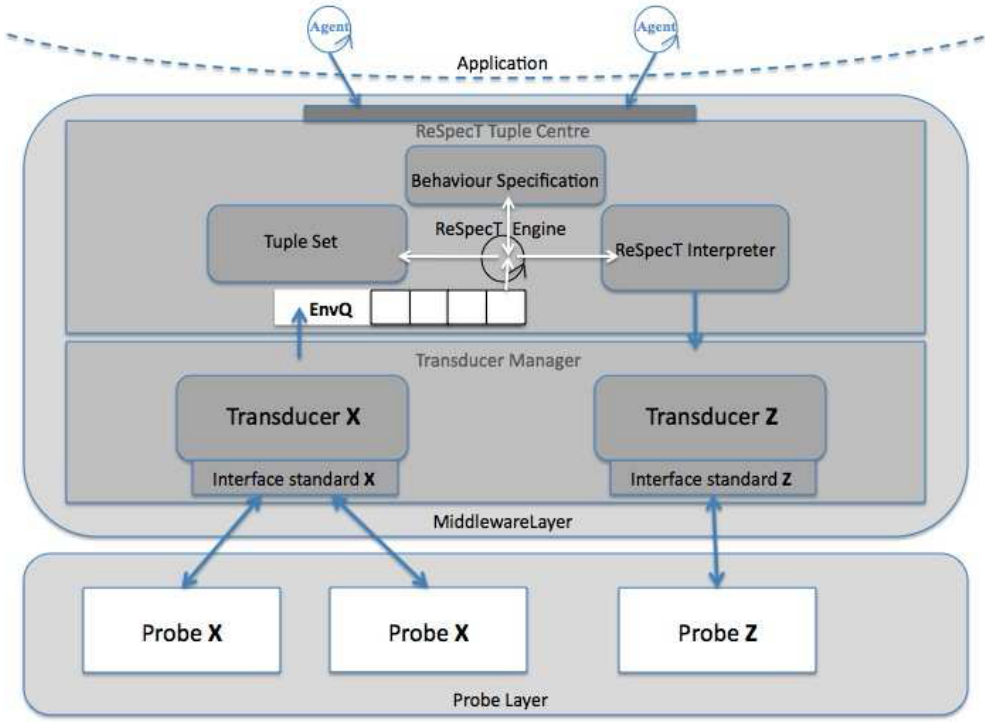
reaction(setEnv(key,Value), to env, out(key set to(Value)))

L'ARCHITETTURA

L'architettura implementante concetti di situatedness è stata pensata su tre livelli:

- livello **probe** come strato più basso
- livello **transducer** come livello intermedio
- livello **applicativo**

Mentre il livello applicativo viene definito a seconda dell'applicazione, i due nuovi livelli sono stati progettati per poter astrarre dalle tecnologie specifiche partecipando alla costruzione dell'artefatto che astrarrà l'ambiente agli agenti applicativi.



[8]

LIVELLO PROBE

Il livello probe (sonda) è lo strato più basso della struttura, rappresenta nello specifico il sensore/trasduttore o meglio è in comunicazione diretta con la risorsa concreta. Il suo scopo primario consiste nel comunicare con la tecnologia specifica in modo tale da riuscire ad ottenere ed inviare informazioni sia da questa sia al transducer. All'interno del livello probe sono presenti componenti (le probes) che rappresentano una sorta di risorsa software in stretto legame con la controparte hardware. Generalmente quindi le singole probe si occupano della mediazione mentre il livello probe, che le contiene, implementa il supporto necessario che spesso corrisponde alle API specifiche per quella tecnologia. La presenza del livello probe fornisce un'astrazione da queste tecnologie.

Normalmente l'associazione tra probe e transducer è di tipo 1:1, tuttavia è possibile associarne diversi ad uno stesso transducer, in particolare quando questi sono simili nel comportamento o nelle informazioni trattate. Queste associazioni sono dinamiche ovvero è possibile inserirle o eliminarle all'interno del sistema. La comunicazione fra queste entità è di tipo peer-to-peer dato che i due componenti si scambiano continuamente informazioni come pari, senza aggiungere alcuna interpretazione.

Una probe possiede un identificatore univoco (ProbeId) all'interno dello stesso transducer, questo significa che possono coesistere probe con lo stesso nome all'interno di un sistema ma non possono essere gestiti da un unico transducer. L'identificatore viene scomposto in identificatore di sensore (SensorId) e identificatore di attuatore (ActuatorId) proprio perché la comunicazione può avvenire sia con risorse sensoriali sia con attuatori. Le probe sono componenti la cui realizzazione è di competenza prevalentemente del programmatore, essendo dipendenti dalla tecnologia utilizzata.

All'interno del sistema TuCSoN viene quindi presentata solamente una prima interfaccia di utilizzo che riassume i metodi principali necessari per una corretta interazione tra probe e transducer, consultabile tramite *ISimpleProbe.java*.

LIVELLO TRANSDUCER

L'entità Transducer è utile a tradurre notifiche provenienti dalle varie probe in linguaggio ReSpecT.

Lo scopo primario è fornire un layer intermedio che nasconda i dettagli tecnici della comunicazione tra sensore/attuatore e il tuple centre, mettendo a disposizione una notifica degli eventi uniforme e generale. Un Transducer dovrà occuparsi quindi di tradurre le notifiche ricevute da una risorsa ambientale in linguaggio ReSpecT-like in grado di essere interpretata dal tuple centre. Specularmente deve essere in grado anche di ricevere richieste da parte di quest'ultimo traducendole in comandi familiari alla risorsa di destinazione. Tutto ciò deve essere progettato garantendo un certo livello di astrazione, concetto in conflitto con la necessità del Transducer di essere specifico per natura. Idealmente infatti ogni tipo di risorsa vuole il suo specifico Transducer, e quindi la diretta conseguenza consiste nel fornire una classe astratta di base (*AbstractTransducer.java*) da specificare poi in base all'applicazione del caso.

Vengono riportati alcuni metodi considerati degni di nota:

public void notifyEnvEvent (String key , int value)

Metodo usato da una risorsa per notificare una proprietà al sistema, definendo i dati necessari alla completezza dell'informazione.

public boolean notifyOutput (InternalEvent ev)

Metodo usato dal Tuple Centre per comunicare una richiesta di operazione verso una risorsa. In particolare si distinguono solo due tipi di richieste, getEnv e setEnv, e nel caso fosse uno di questi, viene richiamato il metodo apposito, che verrà definito in base all'applicazione.

public abstract boolean getEnv(String key)

Definisce il comportamento che deve assumere il Transducer a seguito di una richiesta getEnv. Questo metodo viene definito astratto in modo da lasciare al programmatore la definizione specifica del suo corpo;

public abstract boolean setEnv(String key , int value)

Definisce il comportamento che deve assumere il Transducer a seguito di una richiesta setEnv. Questo metodo viene definito astratto in modo da lasciare al programmatore la definizione specifica del suo corpo.

Avendo fatto l'overview di concetti ed architettura spaziale ed ambientale si procede con il caso concreto.

5.2 - Modifiche e test

Al fine di capire se le modifiche effettuate avessero compromesso anche la parte di situatedness si è proceduto eseguendo il test specifico che non è andato a buon fine.

Il test in questione infatti, appartenente a *alice.tucson.examples.situatedness*, viene avviato mediante *Thermostat.java* il quale acquisisce la temperatura da un sensore (ActualSensor) ed agisce su un attuatore (ActualActuator) per mantenere la temperatura fra 18 e 22 gradi. Il termostato è implementato come agente TuCSoN mentre sensori ed attuatori sono i "probes" sopra descritti che si interfacciano al sistema mediante i trasduttori. Tutti i componenti hanno il proprio centro di tuple programmato attraverso *sensorSpec.rsp* e *actuatorSpec.rsp*.
E' stato necessario ricostruire il flusso delle chiamate utili a leggere la temperatura in modo da capirne il funzionamento ed il fallimento. Ne viene riportato un estratto.

Thermostat -> sensorTc -> SensorTrasducer -> ActualSensor -> tempTc -> ActualSensor -> SensorTrasducer -> sensorTc -> Thermostat

In pratica Thermostat richiede la temperatura al centro di tuple (sensorTc) tramite una "in":

```
template = LogicTuple.parse("sense(temp(_))");
op = acc.in(sensorTc, template, null);
```

La "in" scatena una reazione definita in *sensorSpec.rsp:*

```
reaction(
    in(sense(temp(T))),
    (operation, invocation),
    (
        sensor@localhost:20504 ? getEnv(temp, T)
    )
).
```

la quale richiama *getEnv(…)* sul S*ensorTranducer* tramite *notifyOutput()* dichiarata nell'*AbstractTransducer.*
La chiamata *getEnv(…)* a sua volta permette di richiamare la *readValue()* sul sensore vero e proprio associato (*ActualSensor*) il quale si occuperà, tramite una "rd", di ottenere il valore attuale di temperatura sul centro di tuple *tempTc*.

Una volta acquisito il valore di temperatura, *ActualSensor* richiama *notifyEnvEvent(temp)* sul trasduttore il quale richiama getEnv(temp):

```
reaction(
    getEnv(temp, T),
    (from_env, completion),
    (
        out(sense(temp(T)))
    )
).
```

sul centro di tuple *sensorTc*. La reazione respect associata è una "out" che inietta la temperatura appena ricavata nel suddetto centro di tuple, quindi Thermostat, che all'inizio aveva chiesto con una "in" la temperatura, può a questo punto leggerla e sbloccarsi.

Il problema riscontrato consisteva nel fatto che quando *SensorTrasducer* (tramite *notifyEnvEvent*()) chiamava la reazione *getEnv* sul *sensorTc* per fare la *out* della temperatura letta la guardia *from_env_0* falliva.

Tutto questo era dovuto ad un errore di implementazione causato dal non corretto aggiornamento di *"ACCProxyNode.java"* che non poteva gestire operazioni di tipo getEnv() o setEnv().

Il codice aggiornato e corretto prevede quindi:

```
} else if (msgType == TucsonOperation.getEnvCode()
        || msgType == TucsonOperation.setEnvCode()) {
    this.node.resolveCore(tid.getName());
    this.node.addTCAgent(this.agentId, tid);
    ITupleCentreOperation op = null;
    synchronized (this.requests) {
        try {
            if (this.tcId == null) {
                op = TupleCentreContainer.doEnvironmentalOperation(
                        msgType, this.agentId, tid, msg.getInputEventMsg().getTuple(),
                        this);
            } else {
                op = TupleCentreContainer.doEnvironmentalOperation(
                        msgType, this.tcId, tid, msg.getInputEventMsg().getTuple(),
                        this);
            }
        } catch (final TucsonOperationNotPossibleException e) {
            System.err.println("[ACCProxyNodeSide]: " + e);
            break;
        } catch (final OperationTimeOutException e) {
            System.err.println("[ACCProxyNodeSide]: " + e);
            break;
        } catch (final UnreachableNodeException e) {
            System.err.println("[ACCProxyNodeSide]: " + e);
            break;
        }
    }
    this.requests.put(Long.valueOf(msg.getInputEventMsg().getOpId()), msg);
    this.opVsReq.put(Long.valueOf(op.getId()),
            Long.valueOf(msg.getInputEventMsg().getOpId()));
}
```

Tale integrazione di codice richiede anche l'aggiornamento della classe *TupleCentreContainer.java*
(package: alice.tucson.service) con i metodi *doEnvironmentalOperation(…)* non importati in fase di aggiornamento.
Al termine delle modifiche il test viene completato con successo.

CONCLUSIONI

Giunti alle battute finali è possibile affermare che la versione di TuCSoN su Android è allineata all'ultima release disponibile la quale vede ora integrate anche le funzioni di geo-localizzazione e situatedness aggiornata.

Ricapitolando, il complessivo lavoro consistito in:

- Allineamento app Android da TuCSoN v1.11.0.0209 all'ultima release disponibile TuCSoN v1.12.0.0301
- Refactoring ed integrazione della geolocalizzazione platform-independent (lato TuCSoN)
- Refactoring ed integrazione della geolocalizzazione platform-dependent (lato Android)
- Refactoring ed integrazione della proprietà di situatedness event-driven
- Testing di ogni fase

è stato interamente completato e gli obiettivi sono stati raggiunti.

Da segnalare che le prove effettuate in fase di debug sono state portate a termine su due piattaforme: un dispositivo virtuale (emulatore) Nexus 5 configurato con Android 5.0 (Lollipop) ed un dispositivo fisico, Nexus 6, configurato con Android 6.0 (Marshmallow) non trovando problematiche di compatibilità se non un incremento di performance.

I test sono tutti funzionanti mentre per quello che riguarda l'applicazione mobile è consigliabile proseguire apportando alcune migliorie. A titolo esemplificativo è necessario rivedere la gestione delle chiamate server, utili ad ottenere un indirizzo a partire da coordinate geografiche, implementando una soluzione asincrona.

BIBLIOGRAFIA E SITOGRAFIA

[*1] Coordinazione space-aware per dispositivi mobili in TuCSoN - Michele Bombardi*

[2] Statistiche Android - https://developer.android.com/about/dashboards/index.html

[3] API Android - developer.android.com

[4] it.wikipedia.org

[5] Esempi geo-localizzazione Android - www.html.it

[6] Strumenti geo-localizzazione Android - www.vogella.com

[7] Event Driven Model - technologyuk.net

[8] Coordinazione situata: integrazione di Arduino in un middleware basato su tuple - Steven Maraldi

[9] Space-aware Coordination in ReSpecT - S.Mariani, A.Omicini

[10] The TuCSoN Coordination Model & Technology. A Guide - A.Omicini, S.Mariani

[11] Situatedness in TuCSoN - http://apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/situatednesspdf.pdf

[12] Situated Tuple Centres in ReSpecT - M.Casadei, A.Omicini

RINGRAZIAMENTI

Ora che il lavoro è terminato, augurandomi che possa essere un valore per coloro i quali svilupperanno e useranno TuCSoN, desidero esprimere alcuni ringraziamenti.

Il primo ringraziamento lo faccio a Dio che mi ha accompagnato in questi anni accogliendo quotidianamente tutte le mie insicurezze indicandomi la strada ogni volta che ho tentato di mettere qualsiasi scelta in mano sua.

Il pensiero poi non può che andare alla mia famiglia, quindi ai miei genitori, Claudio e Francesca, che mi hanno dato la possibilità e la forza di laurearmi al corso triennale continuando a sostenermi in tutto il percorso universitario. Ci sono stati giorni in cui non sono stato molto "carino" e come sempre hanno capito e sopportato. Non sempre sono bravo a ringraziarli. Gli devo tutto.

Non posso che continuare con Alessandra e Carlotta, le mie sorelle ma anche Matteo e Leonardo, il mio primo nipotino nato proprio in questi giorni di tesi, così come la nonna e la zia Paola che hanno gioito con me ad ogni passettino compiuto.

Grazie a Martina, per essere entrata nel mio cuore ed aver scelto di passare con me questi anni standomi vicino anche quando le preoccupazioni non mi facevano essere al meglio, con il grande desiderio che questa Laurea possa essere nostra, contribuendo ad aiutarci a costruire un futuro insieme.

Non posso non citare i miei amici, ai quali ho tolto tantissimo tempo per poter portare avanti studio/lavoro. Basc, per aver desiderato più di me il completamento di questo percorso di studi, ma ancor più Luca, Zebe, Bene, Cassa e Marta con i quali condivido i ricordi indelebili di via Pietralata. Vivere insieme esperienze come questa unisce nel profondo e quei giorni saranno i giorni di cui parleremo fra dieci o venti anni.

Una citazione particolare va a Fabrizio e Luca, due persone che non chiamerò mai colleghi ma amici, continuando con Simo&Rik e Matteo con i quali ho condiviso tante fatiche. L'aspetto umano sarà sempre il ricordo migliore che mi lascerà l'università.

Voglio ringraziare il mio relatore, Andrea Omicini, così come il correlatore, Stefano Mariani, per la grandissima disponibilità dimostratami e l'impeccabile presenza lungo questo lavoro. Credo che non avrei potuto fare scelta migliore rivolgendomi a loro.

Voglio ringraziare anche tutti i professori che ho avuto, nella mia formazione c'è qualcosa di tutti e non posso dimenticarmene. Scrivendo queste righe sembra incredibile ripensare a tutti i momenti passati a Cesena.

Mi avvio al termine con persone a cui sono davvero riconoscente. I miei primi datori di lavoro, anche se è veramente riduttivo chiamarli così, Davide Casadei Valentini e Davide Petrini. Loro, insieme a tutta la Kreosoft, mi hanno si permesso di lavorare e studiare parallelamente ma soprattutto hanno reso estremamente positivo il mio ingresso nel mondo del lavoro facendomi sentire prima di tutto parte di un gruppo e poi di una azienda.

Un grazie ora è per la Passepartout e per Simone Casadei Valentini che hanno deciso di credere in me e verso i quali mi sento davvero grato perché mi stanno dando la possibilità di continuare a crescere.

Infine, con le lacrime agli occhi, voglio concludere davvero con una persona per cui la mia Laurea sarebbe stata un orgoglio infinito e che sono sicuro lassù, starà dicendo agli angeli: "Ve lo dicevo io che ce la faceva di nuovo!!". Il pensiero vola al mio caro nonno.

E ora..benvenuto futuro