

**ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA**

---

Scuola di Ingegneria e Architettura Campus di Cesena  
Corso di Laurea Magistrale in Ingegneria Informatica

**SPATIAL COMPUTING PER DISPOSITIVI  
MOBILE ED EMBEDDED**

Elaborato in:  
Linguaggi e Modelli Computazionali LM

*Tesi di Laurea di:*  
**MARCO ALESSI**

*Relatore:*  
**Prof. MIRKO VIROLI**

---

**ANNO ACCADEMICO 2013–2014  
SESSIONE III**







# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Scenario . . . . .	1
1.2	Obiettivi . . . . .	2
<b>2</b>	<b>Spatial computing</b>	<b>5</b>
2.1	Il medium amorfо . . . . .	5
2.2	Proto . . . . .	7
2.3	Scenari applicativi . . . . .	8
<b>3</b>	<b>Magic Carpet</b>	<b>11</b>
3.1	Il DISI incontra il MAMbo . . . . .	11
3.2	NFC . . . . .	13
3.3	Le comunicazioni opportunistiche . . . . .	14
3.4	Architettura . . . . .	16
3.5	Applicazioni realizzate . . . . .	19
3.5.1	Channel . . . . .	20
3.5.2	Partition . . . . .	23
3.5.3	Eightpuzzle . . . . .	25
3.5.4	Tic Tac Toe . . . . .	27
3.5.5	Phuzzle . . . . .	28
3.5.6	Flagfinder . . . . .	31
3.6	Il simulatore Java . . . . .	33
<b>4</b>	<b>Le tecnologie alla base</b>	<b>35</b>
4.1	Bluetooth . . . . .	36
4.2	Bluetooth Low Energy . . . . .	37
4.2.1	Beacon . . . . .	39
4.3	Raspberry Pi . . . . .	40
<b>5</b>	<b>Comunicazioni opportunistiche</b>	<b>43</b>
5.1	Bluetooth standard . . . . .	43
5.2	Bluetooth Low Energy . . . . .	44

5.2.1	Bluetooth e BLE su Raspberry Pi . . . . .	44
5.3	Sistema Android-Raspberry . . . . .	45
5.3.1	LauncherActivity . . . . .	45
5.3.2	BaseActivity . . . . .	46
5.3.3	PartitionActivity . . . . .	47
5.3.4	PartitionData . . . . .	47
5.4	WiFi ad-hoc . . . . .	48
<b>6</b>	<b>Integrazione nel framework</b>	<b>51</b>
6.1	Architettura complessiva . . . . .	51
6.1.1	Localizzazione di un nodo . . . . .	51
6.1.2	Localizzazione dei nodi vicini . . . . .	52
6.1.3	Comunicazione tra i nodi vicini . . . . .	53
6.2	Progetto . . . . .	53
6.2.1	GeoNode . . . . .	54
6.2.2	GeoTopology . . . . .	60
6.2.3	GeoEmitter . . . . .	62
<b>7</b>	<b>Test</b>	<b>69</b>
7.1	Modalità WiFi ad-hoc . . . . .	69
7.2	Modalità BLE . . . . .	71
<b>8</b>	<b>Conclusioni e sviluppi futuri</b>	<b>75</b>

# Capitolo 1

## Introduzione

### 1.1 Scenario

Al giorno d'oggi la diffusione di dispositivi mobili sempre più avanzati e l'evoluzione delle tecnologie *wireless* ha cambiato radicalmente la comunicazione e la vita delle persone. Qualsiasi utente in possesso di uno *smartphone* o un *tablet* è ora in grado di ricevere informazioni e il numero di servizi sta aumentando in maniera esponenziale.

Questo ha permesso agli sviluppatori software di investigare nel campo del cosiddetto *pervasive computing*: ad esempio l'app di *Starbucks* rileva la presenza della propria caffetteria preferita nelle vicinanze e propone all'utente sullo schermo la tessera fedeltà per pagare direttamente con lo smartphone.

Il passaggio dal desktop allo smartphone come strumento principale per la connettività sta portando ad esempio lo shopping puramente online all'*everywhere commerce*: gli acquisti si possono fare dunque offline con l'aiuto di informazioni o coupon reperiti all'istante via mobile, oppure online mentre si è davanti a una vetrina.

Secondo un nuovo rapporto di ricerca della società di analisi *Berg Insight*, il valore totale del mercato pubblicitario mobile basato sulla localizzazione (*LBA*) crescerà da 1,2 miliardi di euro nel 2013 a 10,7 miliardi di euro nel 2018. Questo quindi corrisponderà al 38,6% di tutta la pubblicità e marketing mobile. Il marketing e la pubblicità di prossimità rappresenteranno, quindi, circa il 7% della pubblicità digitale, o il 2% del totale della spesa pubblicitaria globale per tutti i media.

Inoltre lo sviluppo delle tecnologie wireless, in particolare alla definizione di paradigmi di comunicazione innovativi che supportano scenari a connettività intermittente, dove ogni dispositivo può apparire,

scomparire e riconfigurarsi dinamicamente, ha permesso la creazione di reti auto-organizzanti in qualsiasi luogo e momento, in grado di stabilire contatti, comunicazioni e scambio di informazioni tra utenti in movimento. Non è più fantascienza che qualsiasi oggetto di uso quotidiano possa essere integrato con un chip per collegarlo ad un'ampia rete di altri dispositivi, con lo scopo di creare un ambiente in cui la connettività è incorporata in modo tale da essere discreta e sempre disponibile, a bassi costi e a lunga durata. Un esempio pratico di questo concetto riguarda il settore della domotica: gli elettrodomestici di casa, come lavatrice, lavastoviglie, frigorifero e forno potrebbero interagire tra loro per scambiarsi informazioni al fine di gestire meglio i consumi energetici.

Il punto di partenza di questa tesi è il progetto *Magic Carpet*, nato nell’ambito della tesi dell’Ing. Davide Ensini con l’idea di mostrare alcuni semplici modelli di auto-organizzazione, permettendo di agire sugli elementi del sistema e di osservare le reazioni dovute allo spostamento o alla modifica del comportamento di alcuni nodi. I *tablet* e gli *smartphone* sono i nodi e il “tappeto magico” permette la localizzazione dei dispositivi. Ogni nodo del sistema non è direttamente a conoscenza di tutti gli altri presenti, ma interagisce solo con i propri vicini in modo da costruire una mappa che tenga traccia di tutti i nodi presenti e le relative distanze da sè stesso, allo scopo di auto-organizzarsi per svolgere un determinato task applicativo. Con questa idea sono state realizzate varie *app*, con l’enfasi in alcuni casi sugli aspetti scientifici e in altri su quelli ludici.

Un altro aspetto osservato da vicino è quello delle tecnologie per il posizionamento, con enfasi sulla *proximity*. Il “tappeto magico”, una rete di *tag NFC*, diventa solo una delle alternative per la localizzazione dei nodi, grazie anche al lavoro dell’Ing. Andrea Fortibuoni, che con la sua tesi permette l’acquisizione delle tecnologie *GPS* e *Bluetooth Low Energy*, aprendo le porte ad applicazioni su scala geografica ampia.

## 1.2 Obiettivi

Un tema particolarmente caldo nella ricerca è quello delle comunicazioni opportunistiche, scartate inizialmente in questo progetto anche a causa delle limitazioni in tal senso della piattaforma Android. Per via della grande attenzione alla sicurezza dell’utente di fatto si limita molto la libertà nell’uso delle varie interfacce a bordo dei dispositivi, richiedendo, nel migliore dei casi, invasive conferme ad ogni connessione,

come avviene per Bluetooth o per Wi-Fi Direct. Tuttavia la direzione intrapresa dall'elettronica consumer in generale sembra essere quella di una sempre maggiore apertura e interoperabilità.

Le reti opportunistiche rappresentano un'evoluzione delle reti ad-hoc, in cui non esiste un percorso stabile tra ogni coppia di nodi, sorgente e destinatario di un messaggio, e si sfrutta la mobilità dei nodi e degli utenti per creare nuove opportunità di comunicazione. Le reti opportunistiche supportano partizioni, lunghe disconnessioni e instabilità topologica, in generale dovuta principalmente alla mobilità degli utenti. Il cammino per raggiungere un nodo e consegnare un messaggio viene costruito dinamicamente, selezionando tra i nodi vicini il miglior intermediario per avvicinarsi alla destinazione finale. Quando non esiste nessuna opportunità di inoltro il messaggio viene immagazzinato, per poterlo poi inviare quando si presenteranno nuove opportunità.

Una soluzione potrebbe venire dal supporto al *Bluetooth Low Energy*, le cui specifiche sembrano offrire un'ottima opportunità per le comunicazioni di prossimità. Inoltre *BLE* potrebbe essere utile anche per determinare la distanza tra due dispositivi, semplicemente mediante la misura della potenza del segnale ricevuto.

In questa tesi si vuole focalizzare proprio l'attenzione sulle possibili tecnologie per realizzare comunicazioni realmente opportunistiche. Si analizzerà inoltre il mondo dei dispositivi embedded, in particolare il *Raspberry Pi*, la nota piattaforma per l'esplorazione del mondo dei computer e dei linguaggi di programmazione. Vengono quindi considerate in particolare la tecnologia *Bluetooth* standard e *Bluetooth Low Energy* su *Android* e *Raspberry Pi* e se ne evidenzieranno i difetti e i possibili vantaggi, anche integrando tra loro questi due mondi.

Viene infine realizzata un'integrazione delle varie tecnologie nel framework del “tappeto magico” e se ne mostreranno le potenzialità e i limiti.



# Capitolo 2

## Spatial computing

### 2.1 Il medium amorfo

Lo *Spatial Computing* [12] è un campo emergente della ricerca in cui si ha una stretta correlazione tra la computazione e la disposizione dei calcolatori nello spazio. I dispositivi che effettuano computazioni di questo tipo vengono detti *spatial computers*, ovvero gruppi di calcolatori distribuiti in uno spazio fisico, per i quali:

- la distanza tra i dispositivi ha una forte incidenza sulle comunicazioni;
- gli “obiettivi funzionali” del sistema sono generalmente definiti in termini della sua struttura spaziale.

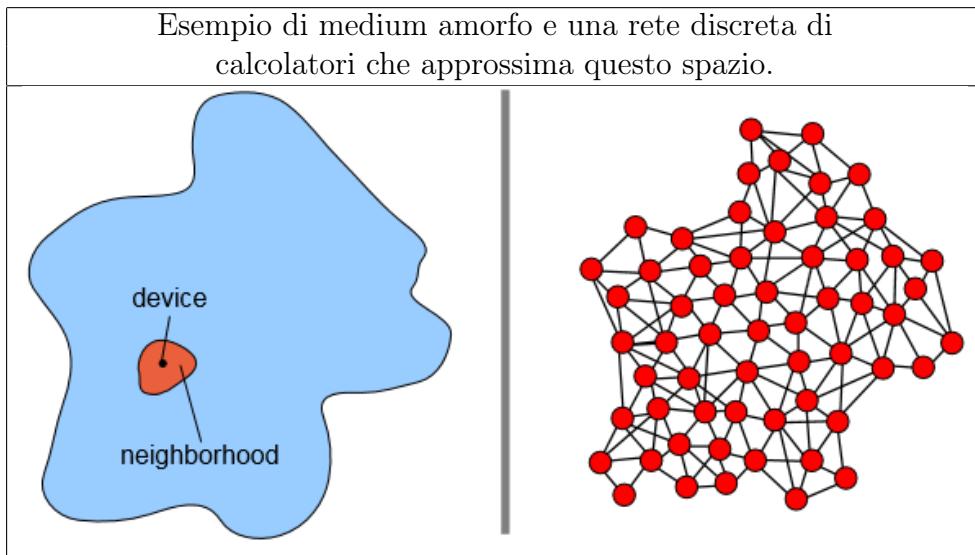
Le applicazioni realizzabili in questo campo coprono molti ambiti diversi tra cui il monitoraggio ambientale, il *pervasive-computing*, le reti di sensori, le reti ad-hoc e lo *smart-environment*. Già dalla fine degli anni ’90 un gruppo di ricercatori del *MIT* [9] aveva immaginato di miscelare circuiti, micro-sensori ed attuatori ai materiali da costruzione, ottenendo ponti in grado di fornire informazioni su carico di vento ed integrità strutturale, o di programmare cellule in biologia come vettori per sostanze farmaceutiche o per produrre manufatti nell’ordine di grandezza dei nanometri. Vent’anni fa i limiti tecnologici frenavano la realizzazione di sistemi di questa tipologia, ma questo non ha impedito che si effettuassero numerose sperimentazioni sui modelli di coordinazione applicabili nel campo dello *spatial computing*, osservando le proprietà emergenti di tali sistemi e sviluppando algoritmi, pattern e metodologie, fino ad arrivare ad un vero e proprio cambio di paradigma: dal momento che per molti sistemi è più efficace porre il

“focus” non tanto sui dispositivi che li compongono, ma piuttosto sullo spazio in cui essi sono distribuiti, non è più conveniente programmare pensando al singolo nodo ma ora si programma lo spazio.

L’idea di base è quella di modellare il sistema come uno spazio continuo [13] piuttosto che come una rete, per facilitare l’organizzazione della computazione distribuita in tali sistemi, agendo quindi su regioni dello spazio geometrico anziché sui singoli dispositivi con conseguenti vantaggi in termini di scalabilità, robustezza e adattabilità. In questo modo non solo si semplifica lo sviluppo ma si possono costruire algoritmi distribuiti che affrontano in modo naturale molti problemi altrimenti considerati difficili.

Se i dispositivi comunicano direttamente tra loro su distanze brevi, allora la struttura globale della rete di comunicazione forma un’approssimazione discreta della struttura dello spazio di interesse, che possiamo definire come un “medium amorfico”. Ogni calcolatore ha un vicinato e l’informazione si propaga nel medium ad una velocità massima  $c$ , per cui ogni calcolatore conosce l’ultimo stato di tutti gli altri device del suo vicinato (*neighborhood*). Tutti gli infiniti device sono programmati allo stesso modo, ma dato che a seconda della posizione i loro sensori daranno input diversi e saranno diversi gli stati dei vicini, le esecuzioni divergeranno.

Un *amorphous medium* può essere considerato come una *Riemannian manifold* (o varietà riemanniana), cioè una generalizzazione del concetto di curva e di superficie differenziabile in dimensioni arbitrarie, su cui sono definite le nozioni di distanza, lunghezza, geodetica, area (o volume) e curvatura. Questa astrazione può ovviamente essere solo approssimata nel mondo reale, ed è proprio quello che si fa adottando questa metafora: si approssima uno spazio costituito da infiniti nodi con una rete discreta di calcolatori. Ogni device rappresenta così una piccola regione dello spazio intorno a sé e i messaggi inviati tra dispositivi adiacenti costituiscono il flusso informativo scambiato nel vicinato.



I vantaggi principali derivanti dall’uso della metafora spaziale si identificano in scalabilità, robustezza e adattabilità.

Se un dispositivo per una qualche ragione dovesse interrompere il suo normale funzionamento, i suoi vicini si prenderebbero carico in modo naturale dello spazio vuoto, semplicemente allargando le maglie dell’approssimazione discreta del medium amorfico. Allo stesso modo è possibile aggiungere nuovi nodi, rendendo più fine la risoluzione dell’approssimazione.

## 2.2 Proto

*Proto* [22] è un linguaggio di programmazione versatile puramente funzionale con una sintassi *LISP-like* che utilizza la metafora del medium amorfico discretizzato per vedere ogni calcolatore spaziale come un’approssimazione di un manifold spazio-temporale con un dispositivo in ogni punto. Le informazioni scorrono attraverso questo manifold ad una velocità limitata e ogni dispositivo ha accesso al recente passato degli altri dispositivi all’interno di un certo raggio di vicinanza.

Le primitive Proto sono operazioni matematiche su *field*, ovvero funzioni che associano ogni punto nel volume spazio temporale ad un valore. Esse sono suddivisibili in 4 categorie:

- computazioni ordinarie puntuali;
- operazioni di vicinato che implicano comunicazione;

- operazioni di *feedback* che stabiliscono variabili di stato;
- operazioni di restrizione che modulano la computazione cambiando il suo dominio.

Proto funziona compilando un programma di alto livello ed eseguendolo in ciascun nodo della rete, localmente si parla di una *Proto Virtual Machine* eseguibile su piattaforme diverse e anche su un simulatore. In concreto i programmi interagiscono con il loro ambiente attraverso sensori e attuatori che misurano e manipolano lo spazio occupato dai dispositivi.

I linguaggi di formazione di pattern usano un’ampia varietà di rappresentazioni di alto livello ma sono comunque limitati per quanto riguarda la portata: pattern di tipo diverso sono difficili da combinare e non possono essere composti in maniera pulita. Proto non offre al programmatore astrazioni di così alto livello ma le sue primitive hanno a che fare solo con vicinati locali nel volume spazio-temporale. Nonostante questo, essendo un linguaggio con semantica funzionale definita in termini di operazioni di aggregazione, alcune funzioni della libreria standard colmano questo gap offrendo operatori a livello aggregato.

Così facendo il programmatore non agisce più a livello di singoli dispositivi (livello locale), ma può direttamente descrivere il comportamento globale delle varie regioni dello spazio. Tali programmi saranno poi trasformati automaticamente in azioni locali che verranno eseguite dai device della rete reale, al fine di raggiungere un’approssimazione del comportamento aggregato desiderato.

In questo modo è possibile colmare quel gap che esiste tra la computazione a livello di singoli dispositivi e la possibilità di controllare il loro comportamento aggregato, e che tuttora costituisce una sfida nel mondo della programmazione.

### 2.3 Scenari applicativi

Un primo possibile scenario applicativo è il *crowd steering*, la guida della folla in situazioni densamente popolate allo scopo di ottimizzarne il deflusso. Gli utenti del sistema hanno come obiettivo il raggiungimento di un punto di interesse posizionato all’interno dello spazio in questione, percorrendo il tragitto più veloce e sfruttando informazioni provenienti dai propri device mobili oppure dai display pubblici collocati nell’ambiente. Il desiderio è quindi quello di ottenere un sistema che, in base a interazioni locali, sia capace di evitare affollamenti

adattandosi dinamicamente ad ogni situazione emergente e imprevista. L’idea di base per risolvere il problema è quella di generare un gradiente computazionale a partire dal punto di interesse mantenendo così in ogni nodo informazioni sulla distanza dalla sorgente. A questa si deve poi aggiungere un fattore legato al livello di affollamento in modo da aumentare la distanza stimata nei punti in cui si sta creando un afflusso considerevole deviando gli utenti verso percorsi più lunghi ma meno affollati.

In [18] si prende in considerazione un grande museo e una varietà di turisti che si muovono al suo interno, assumendo che ciascuno di essi sia dotato di un dispositivo provvisto di funzionalità wireless e con un qualche software a livello utente. Si presume che all’interno del museo siano presenti una serie di dispositivi che, oltre ad offrire connettività wireless ai turisti, possono essere sfruttati sia allo scopo di monitorare e controllare la zona, sia anche per fornire ai visitatori informazioni che li aiutino a raggiungere i loro obiettivi: ad esempio orientarsi all’interno del museo, trovare specifiche opere d’arte, coordinare le loro attività con quelle di altri gruppi di turisti in modo da evitare affollamenti oppure facendo in modo di riunirsi in zone adeguate.

Un caso simile viene affrontato in [17], dove si considera il modello di evacuazione delle folle, mentre in [11] il linguaggio Proto viene utilizzato, con alcuni adattamenti, per la guida di un gruppo di robot, orientati all’esplorazione di aree o al trasporto.

Nell’articolo [15] viene infine presentato un sistema multi-agente emergente di auto-organizzazione progettato per il controllo in tempo reale decentralizzato di reti di distribuzione idrica. Tale sistema è in grado di raggiungere una capacità decisionale paragonabile alle decisioni prese da operatori umani anche esperti, con una maggiore flessibilità, robustezza, scalabilità, adattività e autonomia.



# Capitolo 3

## Magic Carpet

### 3.1 Il DISI incontra il MAMbo

Il progetto *Magic Carpet* [14] nasce con lo scopo di essere presentato in occasione dell'evento del 14-15 marzo 2014, nel quale il Dipartimento di Informatica - Scienze e Ingegneria si è presentato al Museo d'Arte Moderna di Bologna. Per l'occasione vengono presentati prototipi e manufatti digitali per i quali si richiedono:

- fruibilità da parte di un pubblico non tecnico;
- *WOW factor*: i manufatti e le loro installazioni non devono avere necessariamente un carattere artistico ma hanno il principale obiettivo di colpire e attrarre il pubblico;
- innovatività delle tecnologie impiegate;
- fattibilità dell'installazione in un ambito museale;
- interattività con l'audience.

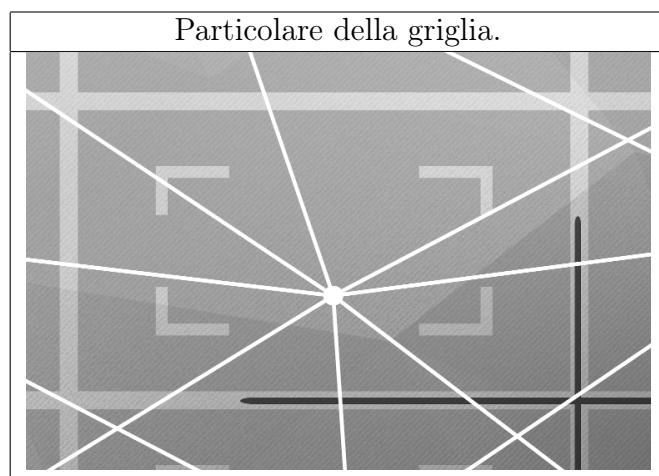
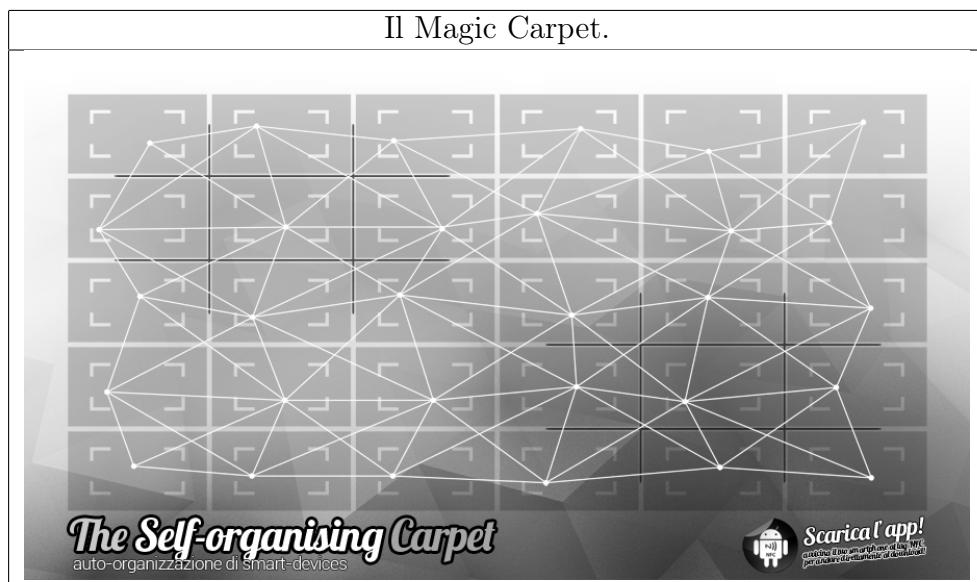
La scelta fatta è stata quella di realizzare l'astrazione di *spatial computing* su di un tavolo, grazie ad un tappeto in cui sono stati disposti dei tag *NFC* (*Near Field Communication*) in modo da formare una griglia.

Utilizzando poi dei *tablet* e *smartphone* con sistema operativo *Android* vengono mostrati alcuni semplici pattern basati su questi concetti di vicinato, ma anche giochi per avere sia aspetti scientifici che ludici.

Appoggiando un device sul tappeto in corrispondenza di un certo tag NFC questo diventa parte attiva del sistema e interagisce con i propri vicini allo scopo di auto-organizzarsi per svolgere un determinato task applicativo.

Per soddisfare il requisito di appetibilità del manufatto, il tappeto viene dotato di una veste in cui sono raffigurati la griglia ed una stilizzazione dei collegamenti tra i nodi. Inoltre tale veste nasconde i tag, scelta che permette di creare curiosità nascondendo ogni indizio riguardo alla capacità del sistema di associare ad ogni device un vicinato.

Nelle figure seguenti viene mostrato il tappeto utilizzato ed alcuni particolari.



Un tag NFC posto sotto il logo consente ai device abilitati e con connessione ad internet di ottenere l'applicativo e partecipare alla dimostrazione.



### 3.2 NFC

Il *Near Field Communication* (o NFC) è una tecnologia che fornisce connettività wireless a corto raggio (2-4 cm), rilasciata nel 2004 dalle aziende Nokia, Philips e Sony. Deriva dalla tecnologia wireless *RFID* (*Radio Frequency Identification*), ma con la differenza di consentire una comunicazione bidirezionale: quando due apparecchi NFC vengono posti a contatto (o accostati entro un raggio di 4 cm), viene creata una rete peer-to-peer tra i due in modo da poter inviare e ricevere informazioni. In questo modo non si ha più la separazione fra “trasponder” (o tag) e “reader” tipica dell’RFID, e neanche la suddivisione fra tag passivi (che devono essere “letti” da un reader) e attivi (che comunicano direttamente fra loro).



Ogni tag NFC ha un proprio *ID* univoco, e sul mercato sono presenti diverse categorie di tag che si distinguono per:

- capacità complessiva: memoria totale disponibile per scrivere dati sul tag;
- numero massimo di caratteri che può contenere un link URL memorizzato sul tag;
- numero massimo di caratteri di cui si può disporre per scrivere un messaggio testuale sul tag;
- crittografia;
- compatibilità.

Un'altra caratteristica importante è la possibilità di poter modificare in qualsiasi momento cosa è scritto sul tag NFC, a meno di non rendere il tag *read-only* (operazione irreversibile). In genere questa funzionalità è fornita solo nei tag di tipo Ultralight. Il costo d'acquisto dei tag NFC varia quindi in base alle sue caratteristiche e oscilla attorno ai 2 euro.

### 3.3 Le comunicazioni opportunistiche

Quando si lavora in ambito *pervasive* si ha a che fare con dinamiche difficilmente o solo parzialmente prevedibili. I dispositivi connessi possono muoversi e venire a trovarsi in prossimità di altri dispositivi o di sensori, con la possibilità di generare comportamenti positivi supportati da un opportuno scambio di informazioni. Tale scambio richiede un'interazione che può avvenire in varie forme.

Nel quotidiano facciamo da anni esperienza di dispositivi che cooperano mediante accoppiamento *Bluetooth*, come gli auricolari per cellulari, o mediante *service discovery* sulla stessa *Local Area Network*, ad esempio nel caso di supporti di memoria condivisi. Nel primo caso la comunicazione è diretta, mentre nel secondo è mediata dall'infrastruttura di rete. In entrambi i casi l'unica operazione richiesta all'utente è di attivare, sui device da connettere, la ricerca dell'altro, senza fornire informazioni aggiuntive. La tendenza per i nuovi *device* interconnessi è l'eliminazione anche di questo passaggio.

La situazione è in rapida evoluzione e lo sforzo che si sta compiendo è mirato a rendere sempre più efficienti, sicure e immediate

queste comunicazioni, anche in ambienti che non siano stati predisposti, vale a dire senza infrastrutture come *access point* o reti cellulari. Questo sforzo si traduce nella sperimentazione di nuovi protocolli e nell’evoluzione di quelli esistenti, con spinte che arrivano da scenari applicativi diversi, come le *body network* in ambito *health care*, la *home automation*, *smart city* e *smart grid* [21, 10]. Una pietra miliare nella diffusione di queste tecnologie è l’integrazione di un supporto alla costruzione di reti *mesh* in iOS.

Quello appena descritto è il quadro di un ambito vastissimo e in pieno fermento che prende il nome di *Internet of Things*, e che ci mette davanti ad un requisito implicito comune ad ogni “abitante” di questo scenario: una connettività che sia ampia, cioè in grado di interfacciarsi con il maggior numero di possibili *peer*, e che sia semplice, ovvero che non richieda infrastrutture o configurazioni particolari.

Il middleware inizialmente realizzato per il *Magic Carpet* presenta le seguenti caratteristiche:

- Ogni nodo viene implementato su uno smart device Android;
- Ogni nodo deve poter reperire informazioni sul proprio vicinato, e deve essere abilitato alla comunicazione bidirezionale esclusivamente con i vicini;
- La computazione è organizzata in cicli in cui ad una fase di ascolto dei vicini e dei sensori ambientali segue una fase di valutazione del proprio stato e di distribuzione dello stesso nel vicinato;
- I nodi leggono tag NFC per conoscere la propria posizione.

Successivamente all’esperienza di “Il DISI incontra il MAMbo” il progetto non si è arrestato ed è stato importante il contributo sperimentale dato dalla tesi dell’Ing. Andrea Fortibuoni [16]. Tale lavoro esamina le tecnologie di posizionamento GPS e Bluetooth Low Energy (BLE), basata sui cosiddetti *Beacon*.

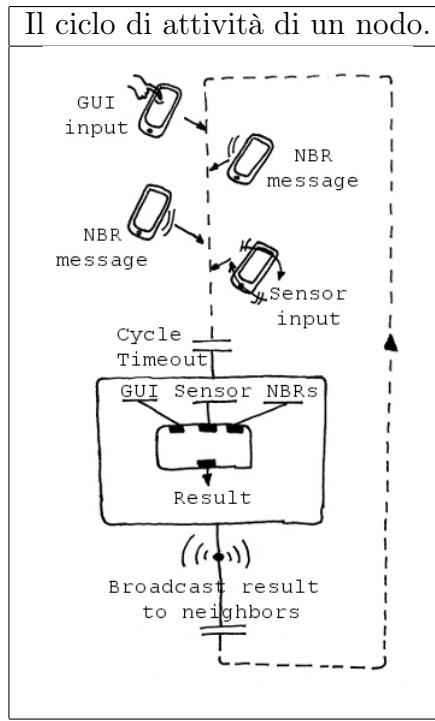
Queste tecnologie sono molto meno precise rispetto a quanto si possa ottenere con NFC, ma questo va visto in una nuova prospettiva: trascorso l’evento, decade il vincolo di lavorare su di un semplice tavolo. Variando la dimensione dei vicinati si compie un cambio di scala tale per cui un errore di qualche metro risulta accettabile, fino ad arrivare ad ambiti outdoor.

Per realizzare l’astrazione tipica dello Spatial Computing, in cui ogni nodo del sistema ottiene dinamicamente le informazioni sui vicini presenti nel suo intorno, si è sfruttata una parte *Server* per gestire

la comunicazione di tali informazioni agli *smart device*, simulando in questo modo le interazioni locali. Nell'ambito di questa tesi verrà studiato il supporto a comunicazioni opportunistiche grazie all'utilizzo di varie tecnologie.

### 3.4 Architettura

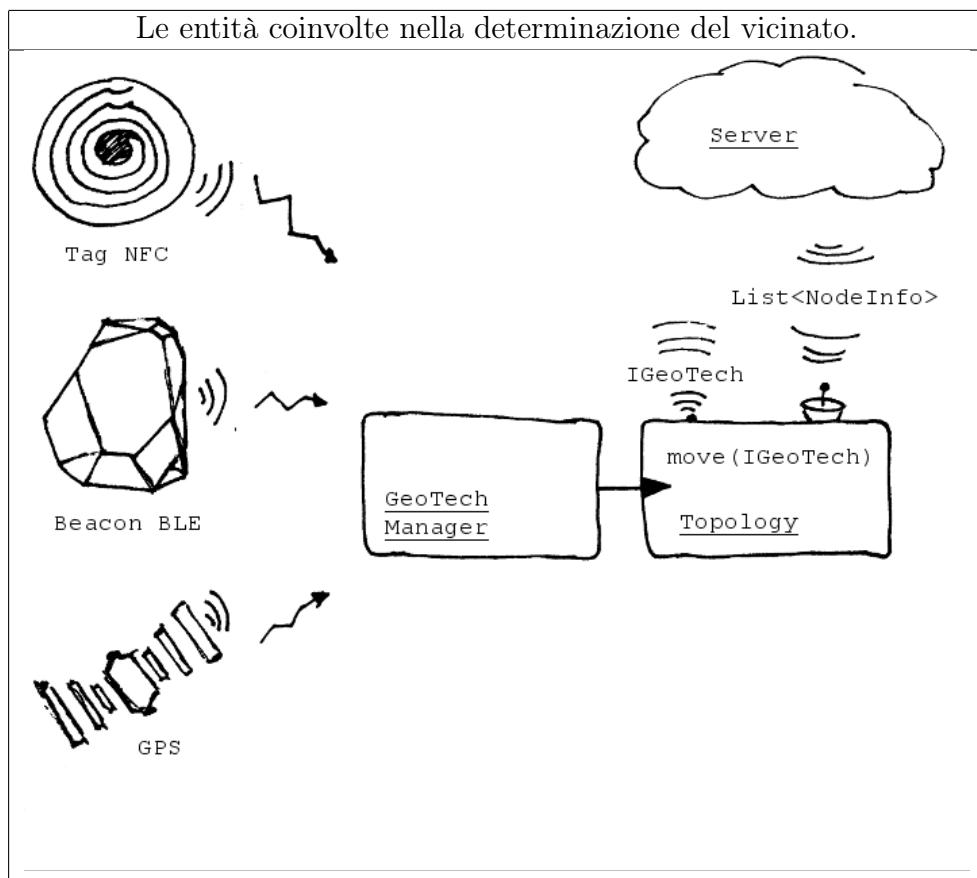
Il paradigma adottato prevede il susseguirsi di cicli computazionali caratterizzati da una fase di attesa ed una fase attiva. Durante la fase di attesa vengono ricevute informazioni dai nodi vicini e da eventuali sensori ambientali, mentre durante la fase attiva l'applicativo produce un risultato elaborando le informazioni ricevute in precedenza per poterlo a sua volta diffondere ai vicini.



Per quanto riguarda la costruzione della mappa di vicinato i nodi si registrano ed inviano informazioni sulla propria posizione, il server le elabora e le confronta con quelle ottenute dagli altri nodi per fornire ad ognuno le informazioni aggiornate sul proprio vicinato, che comprendono la lista dei vicini, le loro distanze e il loro indirizzo.

La piattaforma Android consente di specificare una reazione al ricevimento di nuove informazioni da ciascuno dei sensori utilizzati, in modo simile. Per quanto riguarda NFC, un intent viene generato ogni

volta che si legge con successo un tag, ma non quando, allontanando il device, se ne perde traccia. Per ottenere questa informazione si instaura la connessione con il tag e se ne verifica periodicamente la permanenza, interpretando la perdita della connessione come un distaccamento. Per i beacon BLE, dopo aver abilitato la scansione, viene fornita periodicamente una lista. Il supporto per GPS funziona in modo analogo. In figura vengono rappresentate le entità coinvolte:



Il *GeoTechManager* ha inoltre il ruolo di filtro per frenare la diffusione di aggiornamenti non significativi, e in ogni caso di evitare che tali aggiornamenti siano troppo frequenti. Un aggiornamento è considerato non significativo se è generato da una fonte meno precisa di un'altra attualmente disponibile. Ad esempio, quando un device si trova su un tag NFC, la fonte più precisa a disposizione, qualunque informazione legata ai beacon percepiti o al segnale GPS risulta irrilevante, perché superata da una più affidabile. Le informazioni non propagate vengono comunque immagazzinate, per poter essere utilizzate immediatamente in caso di perdita del segnale dalla fonte di posizionamento

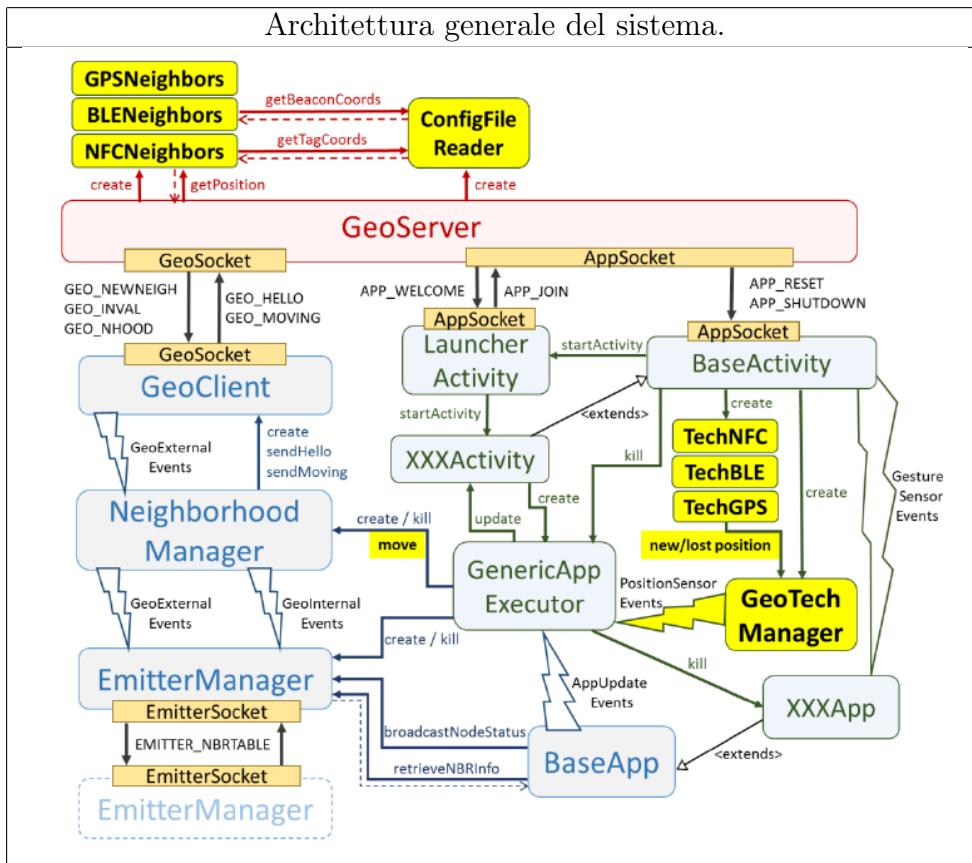
attualmente in uso. Quindi all'allontanamento dal tag NFC segue immediatamente il recupero della migliore informazione disponibile tra quelle di classe inferiore, e la conseguente propagazione.

Ogni nodo riceve dal server degli aggiornamenti ogni volta che nel suo vicinato avviene una modifica significativa, che può riguardare l'ingresso o uscita di nodi dal vicinato stesso, ma anche semplicemente il variare di una distanza.

Il server ha diverse responsabilità e si articola in vari componenti, ne fanno parte le tabelle di riferimento che associano ad ogni tag NFC e ad ogni Beacon una posizione nel sistema di coordinate terrestri.

Quando da un Client si riceve un messaggio, si controlla la validità dei suoi campi, partendo da quello più significativo, cioè con l'indicazione più precisa sulla posizione. Se è disponibile un tag NFC, si assumono per il device le coordinate associate al tag, altrimenti, se il device vede uno o più Beacon, se ne considerano gli *RSSI* (*Received Signal Strength Indication*) per controllare se ce ne sia uno particolarmente prossimo. Se così è, al device si associa la posizione di tale beacon, mentre in caso contrario si elabora una media delle coordinate di tutti i beacon rilevati, ottenendo un'approssimazione che tiene conto di tutte le percezioni. Nel caso in cui non siano percepiti né tag né beacon si ricade su GPS, che di per sé fornisce le coordinate del nodo.

Una volta nota, tale posizione viene confrontata con quella degli altri nodi presenti per costruire adeguatamente i vicinati, associando i nodi la cui distanza reciproca ricade al di sotto di una certa soglia.



### 3.5 Applicazioni realizzate

Si pone il focus in particolare sulle applicazioni realizzate per la piattaforma in *Java*. Un contributo importante è stato dato dalla tesi dell'Ing. Davide Ensini che ha mostrato il possibile utilizzo di un *Domain Specific Language (DSL)*, ottenuto mediante l'integrazione con un interprete per il linguaggio *Protelis* che, consentendo di modellare in modo compatto le interazioni e i concetti dello spatial computing, abbatte le barriere concettuali e rende lo sviluppo più agile ed efficace.

Ad oggi purtroppo non è possibile sfruttare questi risultati sugli smart device, poiché l'interprete *Protelis* è implementato con costrutti introdotti nella versione 8 di *Java*, non supportata da *Android*. Quindi al momento lo sviluppo del framework verso queste nuove implementazioni è stato frenato, mentre si è preferito porre l'attenzione sulla parte comunicativa.

### 3.5.1 Channel

La prima applicazione realizzata è stata *Channel*, nella quale si sfruttano i dispositivi che si trovano tra una sorgente e una destinazione per visualizzare il canale più breve che li unisce.

Per realizzare questo pattern viene originato un gradiente da ciascuno degli estremi, e successivamente, da uno dei due, viene lanciato un gradcast recante come informazione la distanza dall'altro. In ogni punto dello spazio sono così disponibili due gradienti, indici della distanza da ciascuno degli estremi, ed un gradcast che porta come informazione la distanza tra i due. Sono parte del canale le regioni in cui la somma dei due gradienti è uguale alla distanza tra sorgente e destinazione. Disponendo anche delle coordinate dei vicini, oltre che delle loro distanze, si fa comparire sul display di ogni dispositivo una freccia che punta al prossimo nodo lungo il channel.

Si prende questa prima applicazione come standard per mostrare le fasi di implementazione, a partire dal middleware realizzato.

- *ChannelData*: viene definita una classe Java per individuare lo stato del nodo, e quindi quello che verrà inviato ai nodi vicini. I metodi pubblici sono i seguenti, associati ai relativi campi:

```
public double getDistSrc();
public double getDistDst();
public double getDistApart();
public boolean getChannel();
public int getArrowDir();
```

- *ChannelSensor*: viene definita una classe Java per gestire l'input dai sensori, quindi sintetizza tutti i parametri ambientali disponibili. In questo caso l'utente tramite un *double tap* può diventare sorgente o destinazione:

```
public boolean isSrc(); // vero se il device si
trova in regione src
public boolean isDst(); // vero se il device si
trova in regione dst
```

- *ChannelApp*: viene definita una classe Java in modo da estendere la classe astratta *BaseApp* per implementare la logica dell'applicazione che porta a produrre un nuovo stato per il nodo. Si riporta di seguito parte dell'implementazione della funzione *generateLocalNBR*, che elabora il nuovo stato del nodo:

Listing 3.1: Calcolo della distanza dalla sorgente

```
// Set the distance from the source
if (env.isSrc()) {
    dstSrc = 0;
} else {
    for (int i = 0; i < nbrs.size(); i++) {
        ChannelData h = nbrs.get(i);
        if (h == null)
            continue;
        distance = nodes.get(i).getDistance();
        if (h.getDistSrc() != INVALID
            && (dstSrc == INVALID || h.
                getDistSrc() + distance < dstSrc
            )) {
            dstSrc = h.getDistSrc() + distance;
        }
    }
}
```

Listing 3.2: Calcolo dell'appartenenza o meno al channel

```
// Set the total distance (source + destination
)
if (env.isDst()) {
    dstApart = dstSrc;
} else {
    double minDstFromDst = INVALID;
    double minDstFromDstSVal = INVALID;
    for (int i = 0; i < nbrs.size(); i++) {
        ChannelData h = nbrs.get(i);
        if (h == null || h.getDistSrc() < dstSrc)
            continue;
        if ((minDstFromDst == INVALID || (h.
            getDistApart() != INVALID
            && h.getDistDst() != INVALID && (h.
                getDistDst() < minDstFromDst ||
                (h
                    .getDistDst() == minDstFromDst && h
                    .getDistApart() >
                    minDstFromDstSVal)))) {
            minDstFromDst = h.getDistDst();
            minDstFromDstSVal = h.getDistApart();
        }
    }
    dstApart = minDstFromDstSVal;
}

// Check if a device is in a channel or not,
then set the proper arrow
```

```
// direction.
boolean channel = dstApart != INVALID && dstSrc
    + dstDst <= dstApart + tolerance;
```

- *ChannelActivity*: consiste in un Activity Android ed estende la classe astratta  *BaseActivity*, in modo da specificare l'aspetto grafico. Si riporta l'implementazione del metodo *update*, che aggiorna la grafica in seguito all'elaborazione di un nuovo stato del nodo:

Listing 3.3: Funzione update

```
public void update(ChannelData state) {
    TextView tw = (TextView) ((Activity) context
        )
        .findViewById(R.id.logView);
    tw.setText(""
        + Math.round(state.getDistSrc() * Math
            .pow(10, 3))
        / Math.pow(10, 3) + " "
        + Math.round(state.getDistDst() * Math
            .pow(10, 3))
        / Math.pow(10, 3) + " "
        + Math.round(state.getDistApart() *
            Math.pow(10, 3))
        / Math.pow(10, 3) + " " + state.
            getChannel() + " "
        + state.getArrowDir());

    currentDegree = 0f;
    if (state.getDistSrc() == 0) {
        arrow.setImageResource(R.drawable.source)
            ;
    } else if (state.getDistDst() == 0) {
        arrow.setImageResource(R.drawable.
            destination);
    } else if (state.getArrowDir() != INVALID) {
        arrow.setImageResource(R.drawable.arrow2)
            ;
        currentDegree = state.getArrowDir();
    } else {
        arrow.setImageDrawable(null);
    }
    // Set the proper direction of the arrow
    RotateAnimation ra = new RotateAnimation(
        currentDegree,
        currentDegree, Animation.
        RELATIVE_TO_SELF, 0.5f,
```

```

        Animation.RELATIVE_TO_SELF, 0.5f);
    ra.setDuration(0);
    ra.setFillAfter(true);
    arrow.startAnimation(ra);
}

```

Viene mostrata in figura l'applicazione in funzione sul Magic Carpet:



### 3.5.2 Partition

*Partition*, diversamente da *Channel*, sfrutta l'idea di *gradcast*, riducendo sostanzialmente la complessità dell'implementazione alla diffusione del gradiente. Date diverse sorgenti differenziate per colore, i dispositivi che non lo sono assumono il colore della sorgente più vicina.

Viene qui mostrato solamente il cuore della logica applicativa.

Listing 3.4: Calcolo della distanza dalla sorgente e del colore

```

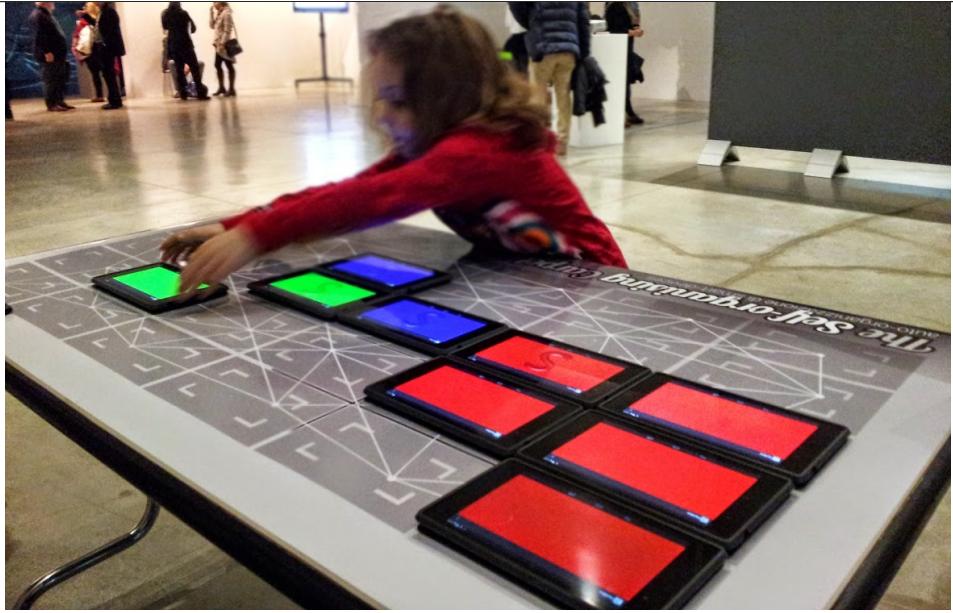
double dstSrc = INVALID;
int tempColor = 0;
// Set the distance from source, with proper color
if (env.isSrc()) {
    dstSrc = 0;
    tempColor = color;
}

```

```
    } else {
        for (int i = 0; i < nbrs.size(); i++) {
            PartitionData h = nbrs.get(i);
            if (h == null)
                continue;
            double distance = nodes.get(i).getDistance();
            if (h.getDistSrc() != INVALID
                && (dstSrc == INVALID || h.getDistSrc()
                    + distance < dstSrc)) {
                dstSrc = h.getDistSrc() + distance;
                tempColor = h.getColor();
            }
        }
    }
}
```

L'implementazione per Android consente la scelta tra 3 colori (rosso, verde o blu) per i device individuati come sorgente. In figura viene mostrata l'app in esecuzione:

App Partition.



### 3.5.3 Eightpuzzle

*Eightpuzzle* è una versione ridotta di *15-puzzle*, noto come gioco del quindici, e consiste nell'ordinare i numeri dall'1 all'8 in una griglia effettuando solo traslazioni che portino ad occupare la casella vuota con una delle tessere adiacenti.



La riduzione di scala è dovuta principalmente al numero di device disponibili, inoltre sono stati creati 3 livelli di difficoltà, utili per la dimostrazione, con la versione *easy* per poter arrivare velocemente alla vittoria con poche mosse.

Questa app, come quelle seguenti del Magic Carpet, è realizzata con coordinate che non rispettano le distanze reali, ma indicano solo la posizione del device in una griglia intera.

Il vincolo di muovere solo sulla casella libera, e solo una tessera adiacente, è forzato nella versione reale del gioco dall'incastro tra i tasselli. Nella versione per Magic Carpet, un messaggio avvisa l'utente dell'imminente errore quando si solleva dal tavolo un device che non andrebbe mosso. Se tale messaggio viene ignorato le regole vengono violate, per cui viene annullata l'intera partita.

Qui di seguito si visualizza la parte della logica applicativa in cui vengono gestite le mosse durante il gioco.

Listing 3.5: Gestione del gioco

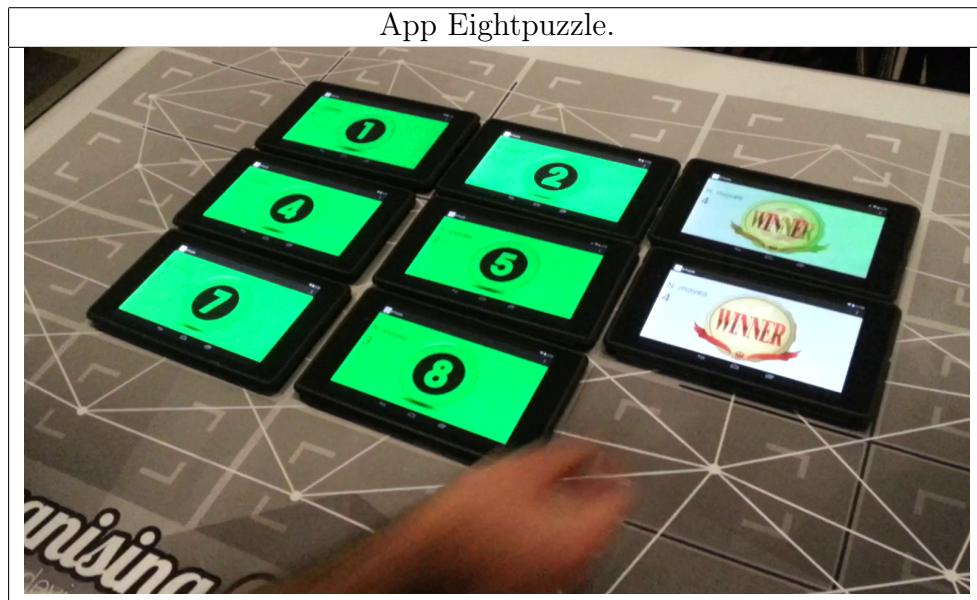
```
// gestione gioco
for (int i = 0; i < grid.size(); i++) {
    if (grid.get(i).getX() == myPos.getX() && grid.
        get(i).getY() == myPos.getY()) {
        pos = i;
    }
}
```

```

    }
    if (pos == INVALID) {
        moveError = true;
    } else if (numbers[pos] == 9) { // gestione mossa
        lecita
        IntGridCoordinates newPos = grid.get(pos);
        IntGridCoordinates lastPos = grid.get(position);
        if (Math.pow(newPos.getX() - lastPos.getX(), 2)
            + Math.pow(newPos.getY() - lastPos.getY(), 2)
            < 1.5) {
            numbers[pos] = number;
            numbers[position] = 9;
            nMoves++;
            position = pos;
        } else {
            gameState = 0;
        }
        if (moveError) {
            moveError = false;
        }
    } else if (pos == position) {
        for (int i = 0; i < nbrs.size(); i++) {
            EightPuzzleData h = nbrs.get(i);
            if (h == null || gameId != h.getGameId())
                continue;
            if (h.getGameState() < 1) {
                gameState = 0;
            } else if (h.getNMoves() > nMoves && h.
                getNumbers() != null) {
                numbers = h.getNumbers();
                nMoves = h.getNMoves();
            }
            count++;
        }
        if (moveError) {
            moveError = false;
        }
        if (count > lastCount) {
            lastCount = count;
        } else if (checkGameError(position, count)) {
            gameState = 0;
        }
    }
}

```

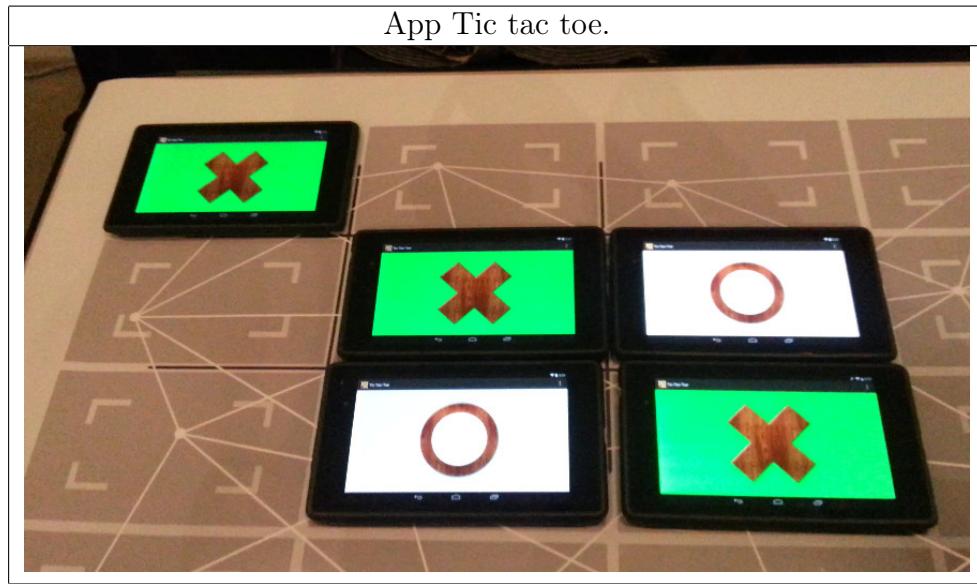
Il raggiungimento della configurazione desiderata viene riconosciuto mostrando un messaggio di congratulazioni su tutti i device.



### 3.5.4 Tic Tac Toe

Meglio noto con il nome di *tris* si presta molto bene alla dimostrazione, per la possibilità di coinvolgere coppie di visitatori in sfide appassionanti. Sono state realizzate due versioni, di cui una a schema libero, che introduce un apprezzabile aspetto di novità in un gioco che di per sé è banale.

L'implementazione è in grado di mantenere attive contemporaneamente due partite sullo stesso tavolo. In figura viene mostrata la vittoria del giocatore contrassegnato dal simbolo *X*, con in evidenza la serie vincente.



### 3.5.5 Phuzzle

*Phuzzle* unisce i termini *Photo* e *Puzzle*, e consiste in un'applicazione in cui i dispositivi cooperano per unire i propri display realizzandone uno più grande, e rendere in tal modo un'unica immagine. L'aggregazione avviene secondo vari criteri, quello prevalente è il tentativo di allargare la superficie aggregata fino ad un massimo di nove device (immagine 3x3), eventualmente tollerando buchi o distorsioni dell'immagine.

Listing 3.6: Funzione generateLocalNBR

```

if (place && myPos != null) {
    for (int i = 0; i < nbrs.size(); i++) {
        PhuzzleData h = nbrs.get(i);
        if (h == null || h.getPosition() == null)
            continue;
        if (nodes.get(i) != null) {
            nbrPos = nodes.get(i).getPosition().
                getCoordinates();
        }
        if (nbrPos != null) {
            // Check if near an existing device in
            // bottom-left position
            if (h.getPosition().getX() == 0 && h.
                getPosition().getY() == 0) {
                if (myPos.getX() >= nbrPos.getX() &&
                    myPos.getY() >= nbrPos.getY()) {
                    if (h.getHorizontalSize() > (myPos.
                        getX() - nbrPos.getX()) && h.

```

```
        getVerticalSize() > (myPos.getY()
            - nbrPos.getY())) {
        h_size = h.getHorizontalSize();
        v_size = h.getVerticalSize();
        position = new IntGridCoordinates
            (myPos.getX() - nbrPos.getX(),
             myPos.getY() - nbrPos.getY())
            ;
        break;
    }
} else if (nbrPos.getX() < myPos.getX()
    && nbrPos.getY() > myPos.getY()) {
    if (max_y == INVALID || nbrPos.getY()
        () < max_y) {
        max_y = nbrPos.getY();
    }
}
}
}
}
if (position == null) {
    // Create the proper grid of the picture
    h_size = 1;
    v_size = 1;
    position = new IntGridCoordinates(0, 0);
    for (int i = 0; i < nbrs.size(); i++) {
        PhuzzleData h = nbrs.get(i);
        if (h == null)
            continue;
        if (nodes.get(i) != null) {
            nbrPos = nodes.get(i).getPosition().
                getCoordinates();
        }
        if (nbrPos != null) {
            if (nbrPos.getX() >= myPos.getX() &&
                nbrPos.getY() >= myPos.getY() &&
                max_y == INVALID || nbrPos.getY() <
                max_y) {
                if ((nbrPos.getX() - myPos.getX() +
                    1) > h_size) {
                    h_size = nbrPos.getX() - myPos.
                        getX() + 1;
                }
                if ((nbrPos.getY() - myPos.getY() +
                    1) > v_size) {
                    v_size = nbrPos.getY() - myPos.
                        getY() + 1;
                }
            }
        }
    }
}
```

```
    }  
}  
}  
}
```

Per questa applicazione la versione Android del server viene estesa abilitando lo scatto e la distribuzione, supportata da un server HTTP, di una fotografia.

Phuzzle è particolarmente interessante perché mostra una possibile applicazione reale del sistema, che riguarda la riconfigurazione automatica in situazioni dinamiche di elementi fisici di un sistema. In questo caso si tratta di schermi, ma si può immaginare un'estensione a macchine di vario tipo.

Nelle figure seguenti si utilizza Phuzzle con un'immagine di presentazione utilizzata in occasione della dimostrazione.

Vari schermi di dimensioni diverse coesistono sul Magic Carpet.



Si forma l'immagine 3x3, anche se non sono presenti tutti i tasselli.

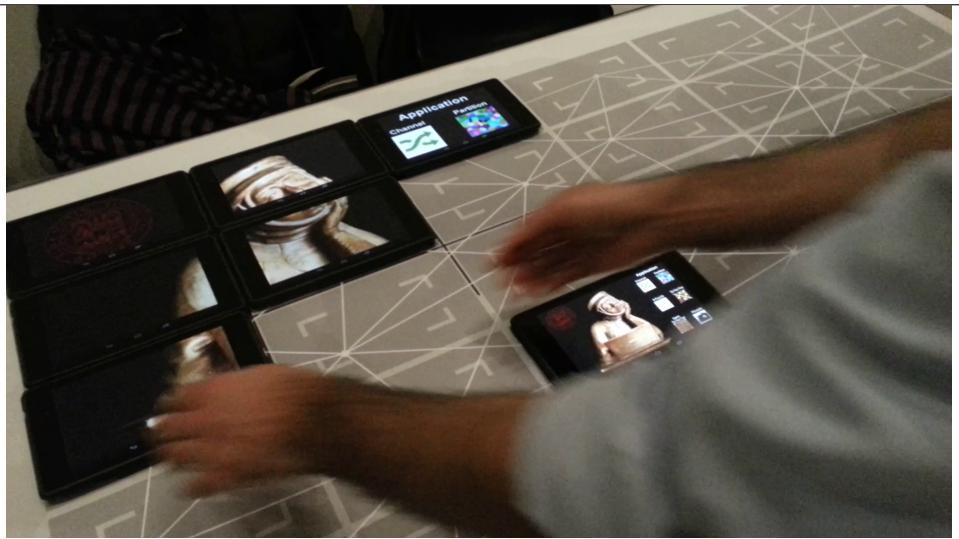
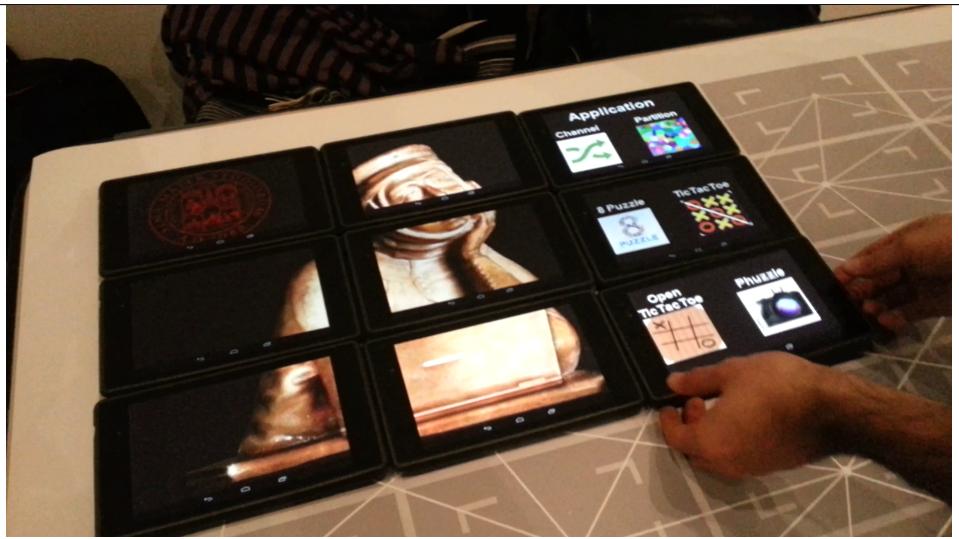


Immagine completamente formata.



### 3.5.6 Flagfinder

*Flagfinder* si distingue dalle altre app sin qui viste per essere l'unica non orientata al solo Magic Carpet. Progettata da Andrea Fortibuoni, essa costituisce anche il primo caso di studio in cui sono stati utilizzati, oltre ai tag NFC, i sensori Bluetooth, insieme ai Beacon, ed il segnale GPS. L'app guida l'utente alla ricerca di un determinato punto di interesse costituito da un device destinazione.

Nella dimostrazione realizzata la bandiera è collocata sopra un tag NFC del Magic Carpet, posto in un laboratorio della sede di Ingegneria e Scienze Informatiche di via Sacchi. L'utente parte dall'esterno e viene guidato, mediante GPS, verso l'edificio. Una volta entrato sono vari Beacon, posizionati ad-hoc, a guidare verso l'aula in cui si trova il tappeto. Una volta che il device *Finder* è posto su di un tag adiacente alla destinazione, la bandiera si considera trovata.

Esterno di via Sacchi, l'app guida l'utente verso l'edificio.



L'utente ha raggiunto il tappeto.  
Il Finder continua a puntare verso la bandiera.

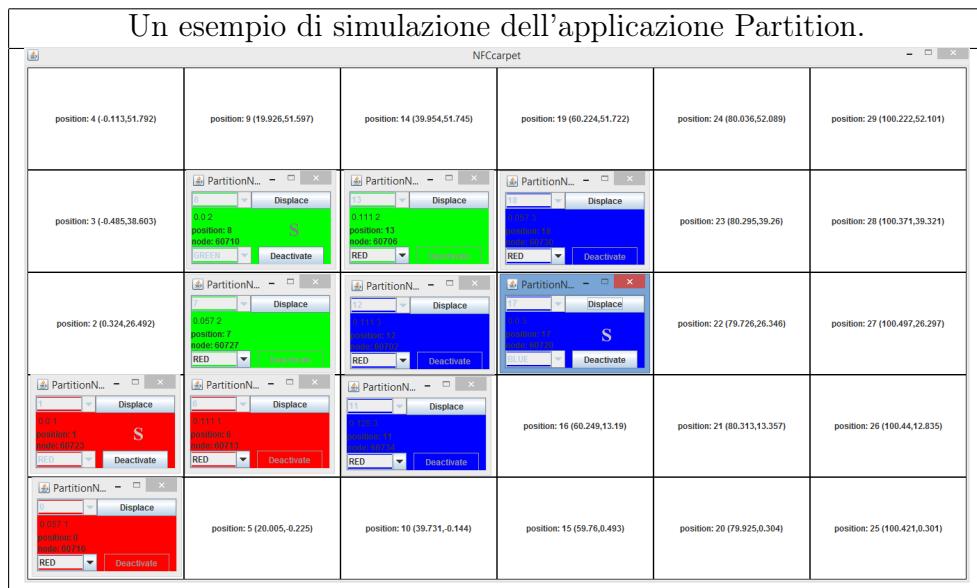


### 3.6 Il simulatore Java

Si è deciso di realizzare un simulatore Java interamente basato sulle librerie javax.swing per poter avere uno strumento tale da poter sperimentare e verificare il funzionamento del sistema e delle app sviluppate in modo molto agile, senza dover utilizzare i device Android. Le componenti di questo simulatore sono:

- una classe *NFCcarpet* che consente di realizzare un tappeto virtuale con tag NFC, visualizzando tutte le posizioni in una griglia che si adatta dinamicamente alle dimensioni dello schermo;
- una classe *BaseNode* che permette di simulare un generico nodo del sistema, rappresentandolo come una finestra grafica delle stesse dimensioni di una posizione del tappeto simulato, e che racchiude dentro di sé tutte le funzionalità comuni ai nodi delle varie applicazioni specifiche, ovvero la possibilità di connettersi all'indirizzo ip e porta del Server e di simulare, grazie a componenti grafici come *JButton* e *JComboBox*, un posizionamento sopra ad un certo tag NFC del tappeto, una rimozione da una certa posizione e il *double tap* sul dispositivo;
- le classi rappresentanti i nodi simulati delle specifiche applicazioni, dentro le quali si va a implementare la rispettiva logica di aggiornamento grafico a fronte degli aggiornamenti dei dati forniti dalla *BaseApp*;
- una classe *NodeLauncher* che si occupa di lanciare i nodi simulati relativi a una specifica applicazione.

Nella figura seguente viene mostrato un esempio di simulazione dell'applicazione Partition, con 10 nodi posizionati sul tappeto, di cui 3 sorgenti di colore diverso fra loro.



# Capitolo 4

## Le tecnologie alla base

Il problema della localizzazione è un tema molto caldo, grazie anche all’impulso dovuto alla progressiva realizzazione dello scenario denominato *Internet of Things*. In particolare la localizzazione indoor necessita di una tecnologia più precisa rispetto a quella basata sul *GPS* (*Global Positioning System*), per guidare le persone verso determinati punti di interesse con un margine d’errore di pochi metri.

Sono molte le tecnologie che in questo ambito sono state sviluppate nel corso degli anni distinguendosi per le differenti caratteristiche e peculiarità:

- tecniche di *Dead Reckoning*: stimano la posizione corrente sulla base del calcolo degli spostamenti da un punto iniziale noto, sfruttando direzione e velocità dell’utente acquisite tramite sensori inerziali (come accelerometro e bussola) in uno specifico intervallo di tempo;
- rete cellulare: non fornisce un’informazione sufficientemente accurata per consentire la navigazione in ambienti chiusi e circoscritti;
- sound-based: impiegano gli ultrasuoni per misurare la distanza tra trasmettitore e ricevitore;
- infrarossi: l’utente indossa un *badge* che emette segnali infrarossi e che vengono ricevuti dai sensori presenti nell’ambiente circostante;
- onde radio (come Wi-Fi e Bluetooth): localizzano l’utente misurando l’intensità del segnale ricevuto (*Received Signal Strength Indicator*) dai nodi presenti in un’infrastruttura;

- *RFID* o *NFC*: tecniche che consentono il rilevamento di prossimità grazie a particolari tag elettro-magnetici;
- sistemi ottici: si compiono operazioni di analisi su immagini acquisite dal device dell’utente e inviate periodicamente a un web server centrale, le quali saranno poi confrontate con quelle presenti nel database in modo da restituire la posizione corrente;
- codici *QR* (*Quick Response*): sono codici a barre bidimensionali che la maggior parte dei telefoni di ultima generazione è in grado di decodificare.

La tesi dell’ Ing. Andrea Fortibuoni [16] ha puntato l’attenzione su 3 tecnologie principali: NFC, Bluetooth Low Energy (o BLE) e GPS. In questo modo si è cercato di unire le potenzialità che queste offrono, ricoprendo tutti i difetti che hanno se le si utilizza singolarmente per la localizzazione: NFC per avere una precisione al millimetro ma limitata a un corto raggio (2-3 cm), BLE per scenari indoor a medio raggio (50-60 metri) con precisione variabile nell’intorno di pochi metri, e GPS per scenari outdoor su ampio raggio.

In questa tesi si focalizza l’attenzione sul Bluetooth in generale e sulle reti WiFi ad-hoc. Verranno poi mostrati i possibili utilizzi per realizzare comunicazioni realmente opportunistiche. Si introduce anche il mondo dei dispositivi embedded, in particolare il Raspberry Pi.

## 4.1 Bluetooth

Bluetooth [6] é uno standard tecnico industriale di trasmissione dati per reti personali senza fili (*WPAN: Wireless Personal Area Network*). Fornisce un metodo standard, economico e sicuro per scambiare informazioni tra dispositivi diversi attraverso una frequenza radio sicura a corto raggio.

Bluetooth (a volte abbreviato in BT) cerca i dispositivi coperti dal segnale radio entro un raggio di qualche decina di metri mettendoli in comunicazione tra loro. Questi dispositivi possono essere ad esempio palmari, telefoni cellulari, personal computer, portatili, stampan- ti, fotocamere digitali, console per videogiochi purché provvisti delle specifiche hardware e software richieste dallo standard stesso.

La specifica Bluetooth é stata sviluppata da Ericsson e in seguito formalizzata dalla *Bluetooth Special Interest Group* (SIG). La SIG, la cui costituzione é stata formalmente annunciata il 20 maggio 1999, é

un'associazione formata da Sony Ericsson, IBM, Intel, Toshiba, Nokia e altre società che si sono aggiunte come associate o come membri aggiunti.

Il nome è ispirato a Harald Blatand (Harold Bluetooth in inglese), re Aroldo I di Danimarca, abile diplomatico che unì gli scandinavi introducendo nella regione il cristianesimo. Gli inventori della tecnologia devono aver ritenuto che fosse un nome adatto per un protocollo capace di mettere in comunicazione dispositivi diversi (così come il re unì i popoli della penisola scandinava con la religione). Il logo della tecnologia unisce infatti le rune nordiche Hagall e Berkanaan, analoghe alle moderne H e B.

Il protocollo Bluetooth lavora nelle frequenze libere di 2,45 GHz. Per ridurre le interferenze il protocollo divide la banda in 79 canali e provvede a commutare tra i vari canali 1600 volte al secondo (frequency hopping).

Nel corso degli anni si è sempre più raffinato nelle varie specifiche rilasciate, sia dal punto di vista della potenza e della velocità di trasmissione, sia nell'affidabilità, robustezza e consumo energetico: dalla versione 1.0 del 1999 si sono susseguite la 2.0 nel 2004, la 2.1 nel 2007 e la 3.0 nel 2009, fino ad arrivare alla 4.0 rilasciata nel luglio del 2010. L'ultima versione dello standard è la 4.1, un'evoluzione incrementale dello standard 4.0.

Rispetto alla tecnologia NFC che permette un raggio d'azione di pochi centimetri, quella Bluetooth consente un raggio d'azione più ampio (50-70 metri effettivi), a discapito di una peggiore precisione (nell'ordine di un paio di metri).

## 4.2 Bluetooth Low Energy

Nel 2010 Bluetooth ha fatto il suo più grande salto, la specifica 4.0 Low Energy [7]. Come suggerisce il nome, il focus di questa nuova versione è quello di poter funzionare su dispositivi con consumi bassissimi. Per identificare i nuovi prodotti LE viene creato un nuovo logo, *Bluetooth Smart*.

Bluetooth Smart ha un protocollo interamente nuovo per un collegamento più rapido ed è un'alternativa al classico Bluetooth delle versioni dalla 1.0 alla 3.0, che può funzionare su dispositivi alimentati da una batteria a bottone, come gli *iBeacon*. Questa tecnologia sta avendo una larga diffusione anche nel mondo dei *wearable*, a partire dalle varie *smart-band* fino agli *smartwatch*.

Questa tecnologia è stata progettata adottando le specifiche dello standard Nokia denominato Wibree e rilasciato già nel 2006, il quale consentiva di operare alla stessa frequenza della banda caratteristica del Bluetooth Basic Rate (ovvero 2,4 GHz ISM) ma con una bit-rate inferiore (un massimo di 1 Mbps a fronte dei 3 Mbps dello standard classico). Il vantaggio principale introdotto da questa tecnologia risiedeva nei consumi energetici, ridotti fino a 100 volte meno rispetto allo standard classico (0,01 mW), permettendo quindi a dispositivi dotati anche solo di semplici batterie coin-cell al litio di durare per mesi o addirittura anni.

Come già detto precedentemente il Bluetooth Smart fu poi incorporato nella versione 4.0 del Core-Specifications e divenne standard nel 2010. I primi dispositivi commerciali (computer e smartphone) dotati della versione 4.0 furono immessi sul mercato solo dalla seconda metà del 2011, primi fra tutti Apple con il suo iPhone 4S. Solo di recente gli altri sistemi operativi, come Android (dalla versione 4.3), Windows Phone (dalla versione 8), e BlackBerry (dalla versione 10) hanno introdotto nativamente sui loro dispositivi il supporto Low Energy.

L'architettura consente di avere quattro ruoli definiti nel *Generic Access Profile* per BLE:

- Broadcaster: è un ruolo ottimizzato per applicazioni di sola trasmissione, e usa l'advertising per l'invio dei dati;
- Observer: è un ruolo ottimizzato per applicazioni di sola ricezione, è complementare al Broadcaster dal quale riceve i dati;
- Peripheral: è un ruolo ottimizzato per dispositivi che supportano una sola connessione e sono più semplici di dispositivi Central, funziona in modalità slave;
- Central: è un ruolo ottimizzato per dispositivi che supportano connessioni multiple, ed è colui che inizia le connessioni con i dispositivi Peripheral, funziona in modalità master.

Bluetooth Smart usa *GATT* come base per i suoi profili. Esso è costruito sopra al protocollo *ATT* (*Attribute Protocol*) ed è obbligatorio per BLE ma può essere usato anche sulla versione standard di Bluetooth.

La lista dei dispositivi che sfruttano la tecnologia Smart e Smart Ready è in continuo aumento, tanto che il Bluetooth SIG prevede una copertura Low Energy sul 90% degli smartphone in circolazione entro l'anno 2018.

### 4.2.1 Beacon

I campi di applicazione del BLE sono in continuo aumento e molte aziende hanno esaminato la possibilità di utilizzarlo per scopi commerciali, visualizzando ad esempio informazioni (come offerte, promozioni, avvisi ad hoc) sugli smartphone dei clienti quando essi entrano nel raggio di un trasmettitore BLE fisso.

Col Bluetooth Smart si sfrutta il modello di advertising caratteristico del ruolo di Broadcaster, che permette ad un dispositivo di inviare ripetutamente informazioni racchiuse in piccoli pacchetti e senza preoccuparsi di stabilire una connessione: è così che nasce l'idea di “beacon”.

*iBeacon* è un prodotto lanciato da Apple nel 2013 che consiste in un dispositivo Bluetooth LE con la funzionalità di “faro”. Questo prodotto in accoppiamento ad iOS 7 permetteva di notificare all'utente la presenza del dispositivo nelle vicinanze.

Questa tecnologia ha molti vantaggi nell'uso indoor, in scenari di proximity market o di domotica. Il dispositivo è formato da un chip BLE, una CPU ARM, e una batteria a bottone.



Le informazioni trasmesse nei pacchetti di advertising contengono quattro informazioni rilevanti:

- *UUID*: un codice univoco a 128 bit utile per identificare i beacon appartenenti ad una determinata applicazione;

- *Major*: un valore a 16 bit usato per distinguere all'interno di un'applicazione a quale sottoinsieme appartengono;
- *Minor*: un valore anch'esso a 16 bit per un ulteriore distinzione dei beacon all'interno del sottoinsieme associato.
- *TX Power*: un dato di 8 bit che rappresenta la potenza di trasmissione (misurata con la metrica *RSSI*), utile per capire a che distanza ci si trova dal beacon.

La precisione del segnale BLE dipende da vari fattori, fra cui le attenuazioni dovute a ostacoli fisici come ad esempio mura e soprattutto elementi contenenti acqua, come il corpo umano.

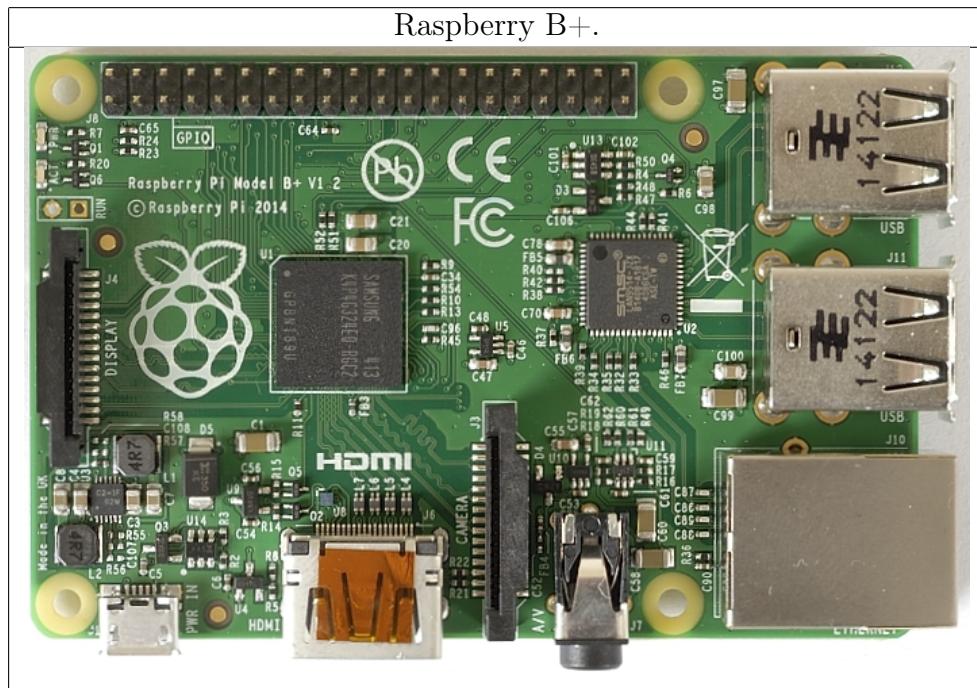
### 4.3 Raspberry Pi

Negli ultimi anni c'è stata un'esplosione dei dispositivi embedded. Questi dispositivi si possono adattare bene al concetto di *Internet of Things*. Tra i più importanti e famosi abbiamo *Arduino* e il più recente *Raspberry Pi* [5].

Il *Raspberry Pi* [8] è un single-board computer (un calcolatore implementato su una sola scheda elettronica) sviluppato nel Regno Unito dalla *Raspberry Pi Foundation*. L'idea di base è la realizzazione di un dispositivo economico, concepito per stimolare l'insegnamento di base dell'informatica e della programmazione nelle scuole.

Il suo lancio al pubblico è avvenuto alla fine del mese di febbraio 2012 e, finora, ne sono state prodotte quattro versioni con prezzi da 20 a 35 dollari statunitensi. La vendita è stata affidata ai due distributori mondiali Farnell e RS Components, specializzati nel campo elettronico con diverse filiali in tutto il mondo, e già nei primi 8 mesi ne sono state vendute circa 500.000 unità.

La versione qui utilizzata è quella *B+*, un aggiornamento tecnico del precedente *Raspberry B*, basato sul chip *Broadcom BCM2835* e munito di 512 MB di RAM, uscita HDMI, ethernet 10/100. Possiede un lettore microSD, un'uscita video AV integrata nel jack da 3.5 mm, 4 porte USB e un chip audio con un miglior rapporto segnale/rumore.



Questo calcolatore è stato progettato per funzionare sui sistemi *Linux* e *RiscOS*. Ci sono diverse distribuzioni linux pronte all'utilizzo, una di queste è *Raspbian*, distribuzione consigliata dalla stessa fondazione, basata su *Debian 7 Wheezy* che include un l'ambiente desktop *LXDE* e tanti altri software pronti all'uso. Il linguaggio principale per lo sviluppo sulla piattaforma è Python, ma c'è un ottimo supporto anche per C, C++ e Java.

Molte applicazioni sono state realizzate sfruttando una delle più importanti funzionalità fornite dal Rasberry Pi, la presenza dei GPIO. I *General Purpose Input/Output* sono pin digitali senza una specifica funzionalità, ma che possono essere programmati a seconda delle esigenze. Questa possibilità ha portato alla creazione di svariate schede di espansione per Raspberry.



# Capitolo 5

## Comunicazioni opportunistiche

In questo capitolo vengono illustrate le prove effettuate con le tecnologie descritte nel precedente capitolo per realizzare comunicazioni opportunistiche. Le applicazioni realizzate consistono nell'invio e ricezione di semplici stringhe in modo da valutare le performance e i limiti di queste tecnologie.

### 5.1 Bluetooth standard

Un primo tentativo è stato fatto basando le comunicazioni su Bluetooth standard, con l'utilizzo di una libreria Android [20] che permette di far comunicare diversi dispositivi mobili tramite Bluetooth [1] e che è in grado di gestire le richieste di connessione, la dinamicità del sistema e la comunicazione in stile peer-to-peer (*P2P*).

Tuttavia i risultati ottenuti non sono stati buoni: la fase di discovery ha la durata di 12 secondi e, pensando ad un'integrazione nel framework, già questo di per sé potrebbe rivelarsi un ostacolo insormontabile. Inoltre dispositivi diversi impiegano tempi diversi per individuarsi e si è notato come alcuni siano molto lenti a vedere dispositivi con stessi servizi rispetto ad altri. Probabilmente questo può essere dovuto al fatto che alcune case produttrici risparmiano sull'utilizzo dei moduli Bluetooth da usare per i dispositivi mobili, puntando a valorizzare di più altri aspetti.

Risultati migliori per quanto riguarda il discovery sono stati ottenuti non selezionando solamente i dispositivi che effettivamente stanno eseguendo l'applicazione desiderata, in modo da non confrontare gli UUID: questo potrebbe portare a problemi di gestione dei device e anche di sicurezza. In generale comunque il meccanismo di invio broadcast di una certa informazione ai dispositivi rilevati come vicini si

traduce per il bluetooth in continue connessioni e disconnessioni, ad ogni ciclo computazionale. Questo risulta problematico e contro il normale utilizzo del protocollo, per cui si passa ad analizzare la modalità BLE.

## 5.2 Bluetooth Low Energy

Le nuove modalità operative della specifica Bluetooth Smart sembrano offrire un'ottima opportunità per le comunicazioni di prossimità, per cui si potrebbe utilizzare il servizio di advertise per poter notificare ai vicini la propria presenza ed in contemporanea lo *scanning* di dispositivi BLE per ricevere queste informazioni.

Android da questo punto di vista supporta dalla versione 4.3 il ruolo di *Central* [2] e fornisce API per poter fare discovery, richiedere servizi e leggere o scrivere caratteristiche. Con la versione 5.0 Android ha introdotto anche il ruolo di *Peripheral* ma purtroppo, al momento della realizzazione di questa tesi, molti device non hanno questo supporto driver, tra cui i Nexus 7 in dotazione. Solamente le nuove versioni dei Nexus 6 e 9 avrebbero questo supporto, ma non è stato possibile effettuare prove a riguardo.

A questo punto, visto il supporto non completo su Android, si è deciso di provare a realizzare il servizio tramite i Raspberry Pi descritti nel capitolo precedente, visto che Android supporta comunque bene il ruolo di *Central*, come già dimostrato nel framework con il riconoscimento dei Beacon.

### 5.2.1 Bluetooth e BLE su Raspberry Pi

Sui Raspberry Pi B+ in dotazione è stato installato Raspbian, il sistema operativo Linux Debian nella sua forma adattata e ridotta per Raspberry, e lo stack BlueZ.

Dopo un'attenta analisi delle possibilità fornite da Linux all'interno di BlueZ per quanto concerne il Bluetooth Low Energy e la sua implementazione in Java, si è ritenuto sconveniente utilizzarlo in quanto questo richiede una modifica dello stack BlueZ di cui non sono chiaramente documentati i meccanismi interni necessari all'introduzione di un profilo *GATT* personalizzato.

Per questo motivo la ricerca di un supporto migliore ha portato all'utilizzo di un modulo *node.js*, *bleno* [4], per realizzare in maniera specifica il ruolo *Peripheral* del BLE. Allo stesso modo esiste un modulo, *noble*, per realizzare il ruolo *Central*, ma visto il supporto già

presente su Android è stato deciso di non utilizzarlo, anche perché sono noti dei conflitti nell'utilizzo dei due moduli in contemporanea.

## 5.3 Sistema Android-Raspberry

Grazie alla precedente analisi è stato realizzato un prototipo basato sull'integrazione in ogni nodo di un sistema Android-Raspberry in modo da mantenere la business logic sul device e utilizzando il Raspberry Pi come servizio per fare advertise.

Per prima cosa è necessario definire il protocollo di comunicazione tra questi due dispositivi, e sono possibili varie modalità, quella qui utilizzata è basata su una connessione *SSH* instaurata dal dispositivo Android.

Ogni Raspberry Pi viene dotato di un dongle WiFi per la connessione di rete e un dongle Bluetooth 4.0 per effettuare l'advertise.

Bleno permette di fare advertise sia in stile Beacon, sia specificando una stringa *name*, di lunghezza massima 26 byte, e un determinato UUID del servizio. Ogni servizio può essere a sua volta composto da caratteristiche, caratterizzate a loro volta da descrittori.

Per ricevere le informazioni di un determinato servizio è comunque necessario che il dispositivo Android esegua una connessione, per cui, in questo caso, ci si limita ad utilizzare *name* per trasferire l'informazione.

L'applicazione seguente vuole mostrare un possibile utilizzo del BLE per realizzare comunicazioni opportunistiche, partendo dalla app Partition, già realizzata per il Magic Carpet.

### 5.3.1 LauncherActivity

Questa Activity si occupa di effettuare la connessione SSH verso il Raspberry Pi, sfruttando la libreria *jsch-0.1.51.jar*, prima di lanciare l'app vera e propria. Si mostra il particolare del metodo *executeRemoteCommand* che lancia il servizio di advertise del Raspberry.

Listing 5.1: Funzione executeRemoteCommand

```
private static String executeRemoteCommand(String
    username, String password, String hostname) throws
    Exception {
    jsch = new JSch();
    session = jsch.getSession(username, hostname);
    session.setPassword(password);
```

```

// Avoid asking for key confirmation
session.setConfig("StrictHostKeyChecking", "no")
;

session.connect();

// SSH Channel
channelssh = (ChannelExec) session.openChannel("exec");
baos = new ByteArrayOutputStream();
channelssh.setOutputStream(baos);

// Execute command
channelssh.setCommand("sudo node /home/pi/tesi/tesi.js");
channelssh.setOutputStream(null);
out = channelssh.getOutputStream();

channelssh.connect();

return baos.toString();
}

```

### 5.3.2 BaseActivity

Questa Activity effettua lo scanning dei dispositivi BLE, in modo da ricevere le informazioni dai nodi vicini. Ogni app dovrà specificare il formato dei propri dati nella stringa e non dovrà comunque superare i 26 byte massimi. Nel prototipo viene effettuato lo scan per 2,5 secondi, con una pausa di 0,5 secondi tra uno scan e il successivo. In questo modo si ottiene un buon compromesso in termini di performance e affidabilità del sistema.

Viene mostrata la callback che si occupa di gestire i nodi rilevati e il metodo relativo.

Listing 5.2: Callback dello scan BLE

```

// Device scan callback.
private BluetoothAdapter.LeScanCallback
mLeScanCallback = new BluetoothAdapter.
LeScanCallback() {
@Override
public void onLeScan(final BluetoothDevice
device, final int rssi, byte[] scanRecord) {
runOnUiThread(new Runnable() {
@Override

```

```

        public void run() {
            addDevice(device, rssi);
        }
    );
}
};

private void addDevice(BluetoothDevice device, int
rssi) {
    if (!mLeDevices.contains(device)){
        mLeDevices.add(device);
        mLeRssi.add(rssi);
    } else {
        mLeRssi.set(mLeDevices.indexOf(device),
rssi);
    }
}
}

```

### 5.3.3 PartitionActivity

Questa Activity contiene la logica applicativa e la gestione dell’interfaccia grafica. In questo prototipo basato esclusivamente sul BLE le distanze vengono misurate grazie ad una stima della potenza del segnale ricevuto (RSSI). Questa stima è difficilmente realizzabile in modo realistico per qualsiasi situazione, esistono articoli di ricerca e tesi su questo studio per capire la stabilità del segnale RSSI e poterlo utilizzare per le distanze.

Ad ogni ciclo computazionale viene calcolato un valore aggiornato del dato di tipo PartitionData descritto in seguito, che verrà inviato per l’advertise al Raspberry.

### 5.3.4 PartitionData

Questa classe Java contiene le strutture dati necessarie per l’app Partition. Si ricorda che utilizzando il solo BLE si riescono a realizzare solamente applicazioni con pochi dati, in quanto per l’advertise semplice si ha il limite di 26 byte per la stringa. In questo caso è necessaria una stringa per identificare l’applicazione, una stringa per identificare il device (per esempio con l’indirizzo ip), un double per identificare la distanza e un intero per identificare il colore.

Listing 5.3: Dettagli della classe PartitionData

```

public String toString(){
    String data = "part;" + host + ";" + distSrc + ";" + color
                ;
    return data;
}

public boolean parseData(String data){
    String[] parts = data.split(";");
    if (!parts[0].equals("part") || parts.length!=4)
    {
        return false;
    }
    host = parts[1];
    distSrc = Double.parseDouble(parts[2]);
    color = Integer.parseInt(parts[3]);
    return true;
}

```

## 5.4 WiFi ad-hoc

L'ultima modalità testata è stata quella del WiFi ad-hoc, grazie alla collaborazione di Mattia Baldani. Una wireless ad hoc network (*WANET*) è un tipo di rete wireless decentralizzata. La rete si dice ad-hoc poichè non si basa su infrastrutture preesistenti, come i router in una rete collegata o punti d'accesso nelle reti wireless, ma ogni nodo partecipa al routing inoltrando dati agli altri nodi. Oltre al classico routing, le reti ad-hoc possono essere utilizzate per il *flooding* dei dati inoltrati.

La natura decentralizzata delle reti wireless ad-hoc le rende adatte per una varietà di applicazioni dove non si può contare su un nodo centrale e può migliorare la scalabilità rispetto alle reti wireless gestite, ma teoricamente e praticamente i limiti all'intera capacità di tale rete devono essere identificati.

Una minima configurazione e un veloce rilascio rendono le reti ad-hoc adatte per situazioni di emergenza come disastri naturali o conflitti militari. La presenza di un protocollo di routing dinamico e adattativo rendono una rete ad-hoc capace di essere formata velocemente.

Per realizzare questa modalità è stato necessario eseguire una procedura per accedere ai privilegi di root nei dispositivi. Inoltre, grazie a due file di configurazione, è possibile switchare dalla modalità WiFi classica a quella ad-hoc, ad esempio grazie ad un semplice pulsante.

Listing 5.4: Implementazione del metodo startWifiAdhoc

```

private void startWifiAdhoc(String myIpAddr, String
    netmask, String essid, int frequency){

    wifiManager.setWifiEnabled(false);

    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}

    String exeDir = "/data/data/" + getPackageName()
        + "/";

    // Execute as root the commands to enable adhoc
    // wifi
    List<String> execResults = Shell.SU.run(new
        String[]{}
        exeDir + "lib_ifconfig_.so wlan0 up",
        exeDir + "lib_ifconfig_.so wlan0 " + myIpAddr
            + " netmask " + netmask,
        exeDir + "lib_iw_.so dev wlan0 set type ibss"
            ,
        exeDir + "lib_iw_.so dev wlan0 ibss join " +
            essid + " " + frequency + " " + BSSID,
        "ip rule add table main"
    );
}

WifiManager wifi = (WifiManager)
    getSystemService(Context.WIFI_SERVICE);
multicastLock = wifi.createMulticastLock("opportunistictnet");
multicastLock.acquire();
}

```

Una *WLAN IBSS* è una rete wireless che rende possibile collegare in modo indipendente più postazioni wireless tra loro senza nessun dispositivo centrale che funga da tramite.

Per quanto riguarda la comunicazione si utilizzano pacchetti UDP, sfruttando il meccanismo dell'invio broadcast.

Nelle reti di calcolatori, il termine *broadcast* indica una modalità di instradamento per la quale un pacchetto dati inviato ad un indirizzo particolare (detto appunto di broadcast) verrà consegnato a tutti i computer collegati alla rete (ad esempio, tutti quelli su un segmento di rete ethernet, o tutti quelli di una sottorete IP).

Per via dei costi gli indirizzamenti broadcast non sono consentiti a livello globale (Internet) per ovvi motivi di congestione e sicurezza, ma risultano vincolati nell'ambito di una rete locale.

Il Broadcast può essere eseguito su diversi livelli del modello OSI. A livello *datalink* l'invio del pacchetto o richiesta viene eseguito sfruttando il MAC Address (indirizzo Fisico) impostandolo al suo valore più alto (FF:FF:FF:FF:FF:FF). A livello *Network* l'invio del pacchetto o richiesta viene eseguita sfruttando l'IP (indirizzo logico) impostandolo sempre al suo valore più alto facendo attenzione alla relativa subnet di appartenenza.

A differenza del TCP, l'UDP (User Datagram Protocol) è un protocollo di tipo *connectionless*, inoltre non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi, ed è perciò generalmente considerato di minore affidabilità. In compenso è molto rapido (non c'è latenza per riordino e ritrasmissione) ed efficiente per le applicazioni "leggere" o *time-sensitive*. Ad esempio, è usato spesso per la trasmissione di informazioni audio-video real-time.

I test effettuati con due Nexus 7 hanno prodotti ottimi risultati e i ritardi in questo caso sono irrilevanti rispetto al BLE, dove comunque, oltre al supporto non completo su Android, si hanno dei tempi più lunghi dovuti allo scanning. In questo caso un nodo invia il suo dato in broadcast ed ogni altro nodo interessato si preoccuperà di recuperarlo, rimanendo in ascolto sulla porta scelta. Purtroppo con più di due Nexus 7 questo sistema ha mostrato dei problemi, probabilmente dovuti ad una carenza del driver *Qualcomm* che non gestisce bene il *merge IBSS*. I Nexus 5 invece con il chip *Broadcom* supportano un *BSSID* fisso e quindi funzionano meglio, con nessun problema rilevato in presenza di 3 device.

Questi risultati sono stati in qualche modo prevedibili in quanto questo tipo di installazione è frequente quando i client sono pochi, ad esempio per permettere a due o tre computer di condividere file o connessioni Internet.

Il sistema *IBSS* è economico, ma non è adatto ad una rete numerosa e concentrata, a causa della sovrapposizione dei segnali e del conseguente calo di affidabilità. Per cui, pensando ad un utilizzo con una moltitudine di nodi, questo non può bastare e si dovranno pensare a modalità molto più scalabili.

# Capitolo 6

## Integrazione nel framework

### 6.1 Architettura complessiva

Il middleware realizzato in questa tesi permette al sistema di localizzare i propri vicini in maniera autonoma, per cui non è più necessaria l'interazione con un Server, come descritto nel capitolo 3, per il concetto di vicinato.

L'architettura generale rimane quella descritta per il Magic Carpet, qui si focalizza l'attenzione sulla parte relativa alla localizzazione e alla comunicazione.

#### 6.1.1 Localizzazione di un nodo

Ogni nodo viene localizzato grazie ad un mix di tecnologie. Vengono quindi definite varie entità, specializzate nel recupero delle informazioni provenienti dai sensori, e un'entità di gestione di più alto livello.

- *TechNFC*: si occupa di rilevare e segnalare se il nodo è posizionato sopra ad un certo tag NFC, identificativo di una determinata posizione sul tappeto;
- *TechBLE*: si occupa di rilevare e segnalare quali beacon BLE vengono percepiti dal nodo;
- *TechGPS*: si occupa di rilevare e segnalare la locazione del nodo in coordinate terrestri, sfruttando il segnale GPS;
- *GeoTechManager*: si occupa di notificare un nuovo cambiamento di posizione di un nodo in base alle tecnologie utilizzate, e anche

di definire gli intervalli di tempo che consentono il passaggio dinamico da una tecnologia ad un'altra sulla base di ciò che viene rilevato dai sensori. Quello che viene notificato consiste in un'entità *GeoTech*, che racchiude al suo interno le informazioni riguardanti l'eventuale tag NFC rilevato, la lista di beacon e le coordinate GPS.

Grazie a questa architettura modulare sarà possibile in futuro aggiungere nuove tecnologie, anche sostituendo quelle attuali, semplicemente configurando, oltre all'entità specializzata per la tecnologia stessa, la nuova entità GeoTech e la gestione di priorità da parte del GeoTechManager. Inoltre è possibile anche agire in maniera dinamica sul sistema di localizzazione, ad esempio tramite input dell'utente si potrebbero abilitare o disabilitare alcune tecnologie.

### 6.1.2 Localizzazione dei nodi vicini

Per realizzare comunicazioni opportunistiche, occorre che i singoli dispositivi vengano dotati di strumenti per scoprire autonomamente i vicini. Vengono quindi definite due modalità:

- *Bluetooth Low Energy*: ogni nodo fa *advertise* delle proprie informazioni, che consistono in un identificativo, indirizzo ip e porta. Ad ogni ciclo computazionale viene effettuata una scansione BLE che, oltre a rilevare gli eventuali beacon presenti, individua indirizzo e porta dei nodi vicini, per cui è possibile inviare a questi nodi la propria localizzazione, grazie all'entità GeoTech definita precedentemente, tramite UDP. In questo modo ogni nodo riceve i GeoTech relativi ai propri nodi vicini ed è in grado di verificarne l'appartenenza al proprio vicinato, in base alla soglia specificata, e di calcolarne le distanze. Date le limitazioni di Android l'advertise verrà realizzato tramite un Raspberry Pi di supporto;
- *WiFi ad-hoc*: ogni nodo invia in broadcast la propria localizzazione GeoTech tramite UDP e riceve i GeoTech relativi ai propri nodi vicini restando in ascolto sulla porta definita dal broadcast. In questo modo la verifica del vicinato è realizzata allo stesso modo della modalità precedente.

Inizialmente l'idea era quella di consentire di avere a disposizione entrambe le modalità in contemporanea. I test effettuati hanno mostrato

alcuni problemi molto probabilmente dovuti alla coesistenza di scanning BLE e broadcasting in WiFi. Questo problema è noto in ricerca ed è dovuto alle frequenze di funzionamento di queste tecnologie, probabilmente per interferenze nel segnale. Fatte queste considerazioni, nel framework le modalità verranno mantenute separate.

### 6.1.3 Comunicazione tra i nodi vicini

La comunicazione tra i nodi vicini per quanto riguarda la ricezione delle informazioni di stato aggiornate relative ad ogni nodo e l'invio ai nodi dello stesso vicinato viene realizzata tramite UDP, allo stesso modo della modalità BLE per quanto riguarda i GeoTech. Quindi ad ogni ciclo computazionale un'applicazione ha a disposizione per la logica applicativa la localizzazione del proprio nodo e le localizzazioni, le distanze e gli stati dei nodi vicini, ottenuti come indicato in questa sezione.

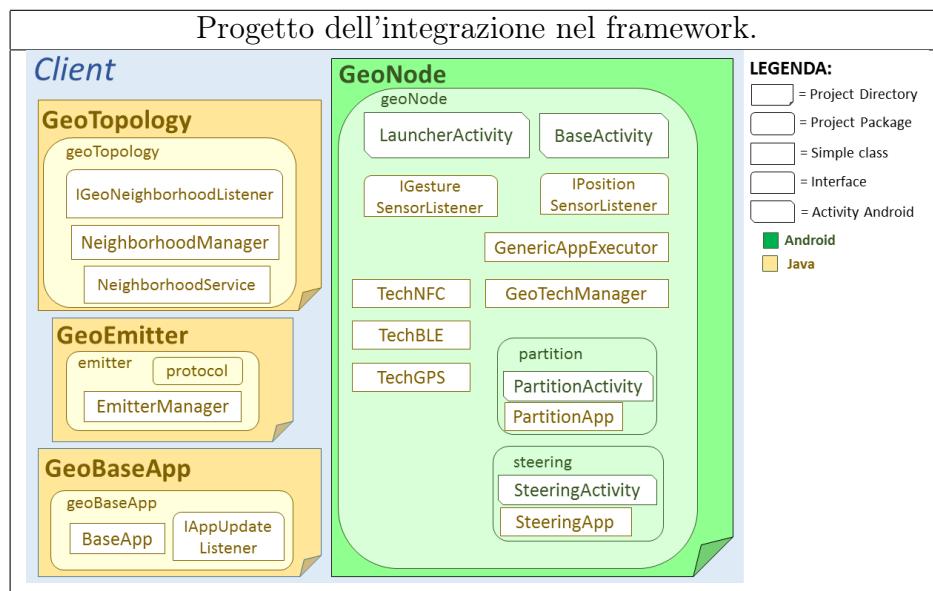
## 6.2 Progetto

Si possono individuare i seguenti sottosistemi:

- *GeoServer*: è l'unico sottosistema della parte Server e si occupa sostanzialmente di specificare quale applicazione mandare in esecuzione tra quelle sviluppate, comunicandolo ai nodi Client che ne fanno richiesta e fornendo loro ogni aggiornamento per quanto riguarda le informazioni relative al proprio vicinato;
- *GeoNode*: si occupa di eseguire sul device l'applicazione scelta dal GeoServer, di specificare tutta la parte computazionale relativa alle varie applicazioni sviluppate, di interagire con le tecnologie NFC, BLE e GPS per poter notificare un nuovo cambiamento di posizione, e di gestire gli eventi generati dall'utente;
- *GeoTopology*: si occupa di comunicare con il GeoServer, inviando i messaggi contenenti le informazioni aggiornate sugli spostamenti dei nodi e ricevendo i vicinati aggiornati;
- *GeoEmitter*: si occupa di ricevere e aggregare tutti i dati aggiornati relativi ad un certo nodo, e di inviare il proprio agli altri nodi dello stesso vicinato;

- *GeoBaseApp*: si occupa di implementare la logica di base di un'applicazione che utilizza il sistema, eseguendo il ciclo computazionale che consente, ad ogni intervallo di tempo, di recuperare dal GeoEmitter le informazioni del nodo Client in considerazione e relativo vicinato, e di notificare gli aggiornamenti di stato relativi ai nodi, segnalando al GeoEmitter di comunicarlo ai nodi interessati.

Date le considerazioni precedenti si dovrà eliminare il sottosistema GeoServer, facendo in modo di integrare la parte riguardante le comunicazioni di prossimità nei sottosistemi di ogni nodo, in particolare GeoNode, GeoTopology e GeoEmitter.



### 6.2.1 GeoNode

Per quanto riguarda il sottosistema *GeoNode* le modifiche principali riguardano le Activity *LauncherActivity* e *BaseActivity* e le classi Java *GenericAppExecutor* e *TechBLE*.

#### Launcher Activity

Questa Activity aveva il compito di stabilire una connessione alla stessa rete WiFi del Server e di comunicare con il GeoServer, creando le relative socket, per richiedere quale applicazione specifica lanciare. Una volta ottenuta la relativa risposta, veniva lanciata l'Activity dell'applicazione vera e propria.

L'Activity viene modificata in modo da prevedere due pulsanti per abilitare le modalità WiFi ad-hoc e integrazione del Raspberry come servizio di advertise BLE, come visto nel capitolo precedente.

Listing 6.1: Attivazione delle modalità

```
Button connectWiFi = (Button) findViewById(R.id.
    connectWiFi);
connectWiFi.setOnClickListener(new View.
    OnClickListener() {
    @Override
    public void onClick(View arg0) {
        new AsyncTask<Integer, Void, Void>(){
            @Override
            protected Void doInBackground(Integer
                ... params) {
                startWifiAdhoc(sp.getString("wifi ip
                    ", "192.168.0.2"), "255.255.255.0
                    ",
                    "mesh", 2437);
                return null;
            }
            .execute(1);
        }
    });
}

Button connectRPi = (Button) findViewById(R.id.
    connectRPi);
connectRPi.setOnClickListener(new View.
    OnClickListener() {
    @Override
    public void onClick(View arg0) {
        new AsyncTask<Integer, Void, Void>(){
            @Override
            protected Void doInBackground(Integer
                ... params) {
                try {
                    executeRemoteCommand("pi", "raspberry
                        ", sp.getString("rpi ip", "192.168.1.134"));
                    thiss.runOnUiThread(new Thread() {
                        public void run() {
                            Toast.makeText(thiss, "
                                Connection OK", Toast.
                                LENGTH_SHORT).show();
                        }
                    });
                    rpi = true;
                } catch (Exception e) {
                    thiss.runOnUiThread(new Thread() {
```

```

        public void run() {
            Toast.makeText(this, "Connection ERROR", Toast.LENGTH_SHORT).show();
        }
    );
    e.printStackTrace();
}
return null;
}.execute(1);
}
);

```

Inoltre è presente la lista delle applicazioni in modo che ogni nodo può decidere autonomamente quale avviare. Un'impostazione aggiuntiva riguarda la selezione del valore di soglia per il vicinato, non più fissato a priori per ogni applicazione ma gestito dinamicamente.

Al lancio di una data applicazione verranno passati tre parametri:

1. *threshold*: valore di soglia per il vicinato;
2. *intCoords*: valore booleano settato a *true* per gestire i giochi realizzati per il Magic Carpet, quindi a coordinate intere rispetto a coordinate terrestri, per le quali il valore viene settato a *false*;
3. *rpi*: valore che indica la modalità di funzionamento scelta, tra integrazione con Raspberry Pi e WiFi ad-hoc.

Listing 6.2: Lancio di un'applicazione

```

private void launchApp(String app, double threshold
, boolean intCoords) {
bleApp.setOutputStream(out);
Intent i = new Intent("it.unibo." + app);
i.putExtra(EXTRA_THRESH, threshold);
i.putExtra(EXTRA_INTCORDS, intCoords);
i.putExtra(EXTRA_RPI, rpi && wifiManager.
    isWifiEnabled());
startActivity(i);
}

```

### Base Activity

Questa Activity costituisce il punto di accesso per il sistema e rimane in primo piano per tutto il funzionamento. L'organizzazione di Android le attribuisce per questo un ruolo privilegiato: solo dall'interno

di questo contesto è possibile agire sui sensori e riceverne le percezioni, e lo stesso vale per la GUI. Si occupa di:

- notificare gli eventi scatenati dall’interazione con l’utente (*double tap* sullo schermo) e dalla localizzazione (ovvero il posizionamento sopra un certo tag NFC, eventuali beacon rilevati e localizzazione GPS);
- inizializzare le applicazioni con le informazioni di base, fornendo loro un metodo astratto che ogni specifica Activity andrà poi a ridefinire per i propri scopi applicativi.

In questa nuova architettura l’Activity in questione si occupa della gestione dei parametri ricevuti dalla LauncherActivity, in modo da configurare correttamente il sistema. Inoltre, quando attiva la modalità con Raspberry Pi, si occupa anche di notificare al GenericAppExecutor la lista aggiornata dei nodi rilevati con il BLE dalla classe TechBLE.

### TechBLE

Questa entità si occupa di rilevare (e segnalare) quali beacon BLE sono percepiti dal nodo. Viene qui integrato lo scanning dei dispositivi BLE, in modo da scoprire i propri vicini. La stringa scelta per l’advertise, rimanendo nei 26 byte a disposizione, è la seguente:

```
nameApp + ":" + id.getId() + ":" + id.getPort()
```

- i : sono utilizzati come separatore per i campi della stringa;
- *nameApp* consiste nel nome dell’applicazione, in modo da filtrare solo i nodi che riguardano l’applicazione scelta;
- *id.getId()* e *id.getPort()* sono rispettivamente l’indirizzo ip e la porta dell’EmitterSocket del nodo.

Si è dovuto procedere utilizzando le API Android per la gestione del protocollo Bluetooth Low Energy, in quanto le API fornite dalla libreria open source di Radius Networks utilizzate in precedenza presentavano dei problemi nel parsing di questi nuovi advertise, diversi da quelli generati dai beacon. In questo modo si riescono a supportare entrambi con una callback in grado di discriminare le due tipologie, come mostrato nell’implementazione seguente.

Listing 6.3: Gestione dell'advertise di beacon e Raspberry Pi

```

// Device scan callback.
private BluetoothAdapter.LeScanCallback
    mLeScanCallback = new BluetoothAdapter.
        LeScanCallback() {
    @Override
    public void onLeScan(final BluetoothDevice
        device, final int rssi, final byte[]
        scanRecord) {
        new Thread(){
            @Override
            public void run() {
                if (device.getName() != null){
                    addDevice(device);
                } else {
                    addBeacon(scanRecord, rssi);
                }
            }
        }.start();
    }
};

private void addDevice(BluetoothDevice device) {
    Node node = parseData(device.getName());
    if (node != null && !nodes.contains(node)){
        nodes.add(node);
    }
}

private Node parseData(String data){
    String[] parts = data.split(":");
    if (!parts[0].equals(caller.getNameApp()) ||
        parts.length!=3){
        return null;
    }
    v("SCAN NODE: " + data);
    return new Node(parts[2], parts[1], Integer.
        parseInt(parts[2]));
}

private void addBeacon(byte[] scanRecord, int rssi)
{
    byte[] value = Arrays.copyOfRange(scanRecord,
        9, 25);
    GeoBeacon beacon = new GeoBeacon(toUuidString
        (value), rssi);
    if (beacon != null && !geoBeacons.contains(
        beacon)){
        geoBeacons.add(beacon);
    }
}

```

```

        v("ADD BEACON: " + beacon.getUuid() + " ,
            rssi: " + beacon.getRssi());
    }

    public String toUuidString(byte[] bytes) {
        return toHexString(bytes, 0, 4) + "-" +
            toHexString(bytes, 4, 2) + "-" + toHexString
            (bytes, 6, 2) + "-" + toHexString(bytes, 8,
            2) + "-" + toHexString(bytes, 10, 6);
    }

    private static String toHexString(byte[] bytes, int
        start, int length) {
        char[] hexChars = new char[length * 2];
        char[] hexArray = "0123456789abcdef".
            toCharArray();
        for (int i = 0; i < length; i++) {
            int v = bytes[start + i] & 0xFF;
            hexChars[i * 2] = hexArray[v >>> 4];
            hexChars[i * 2 + 1] = hexArray[v & 0x0F];
        }
        return new String(hexChars);
    }
}

```

## GenericAppExecutor

Questa classe funge da collegamento tra le specifiche Activity Android di ogni applicazione e i sottosistemi implementati in Java, inclusa la parte computazionale data dalla BaseApp, in modo da disaccoppiare *view* e *control*, separando la parte relativa al controllo da quella relativa alla rappresentazione. Si occupa sostanzialmente di:

- creare l'EmitterManager per una specifica applicazione;
- creare il NeighborhoodManager collegandolo all'EmitterManager appena creato e all'indirizzo ip e porta del GeoServer;
- registrare l'EmitterManager come ascoltatore degli eventi generati dal NeighborhoodManager;
- registrarsi come ascoltatore degli eventi generati dalla specifica applicazione in modo da avvisare di conseguenza la relativa Activity interessata.

Viene qui integrata la connessione SSH verso il Raspberry Pi, in modo da fare advertise della stringa precedentemente definita, una volta

creata l'EmitterSocket grazie all'EmitterManager. Inoltre viene creata una classe Java NeighborhoodService che si occuperà della gestione del vicinato.

### 6.2.2 GeoTopology

Per quanto riguarda il sottosistema *GeoTopology* le modifiche principali riguardano la classe *NeighborhoodManager*, con l'aggiunta della classe *NeighborhoodService* e delle classi precedentemente presenti nel sottosistema *GeoServer* riguardanti la gestione del vicinato.

#### NeighborhoodManager

Questa classe Java si occupava della gestione dei vicini nuovi/persi, comunicati dal GeoClient e notificati a loro volta al sottosistema GeoEmitter, e del posizionamento dei nodi, notificati al GeoClient.

Ora non interessa più la comunicazione con il GeoClient, visto che non sarà più necessario, quindi il NeighborhoodManager avrà il compito di inviare ogni nuova localizzazione del nodo al NeighborhoodService e di notificarla al GeoEmitter che la sfrutterà in caso di modalità WiFi ad-hoc. Se invece è attiva la modalità con Raspberry Pi per fare advertise, al termine di ogni ciclo di scansione il NeighborhoodManager riceverà la lista aggiornata dei nodi vicini e provvederà a notificarla al GeoEmitter.

#### NeighborhoodService

Questa classe Java viene qui introdotta e si occupa di sostituire il Server per la gestione del vicinato.

Utilizza gli strumenti di supporto già presenti nel Server, in particolare:

- *Config*: contiene le coordinate terrestri del tappeto NFC e dei beacon presenti nell'ambiente. Grazie a questo si è in grado di identificare chiaramente le coordinate di un nodo posizionato sopra a un certo tag NFC o nei pressi dei beacon;
- *NFCNeighbors*: si occupa di fornire le coordinate terrestri precise corrispondenti a un certo tag NFC sopra al quale si trova un nodo dato;
- *BLENeighbors*: si occupa di fornire le coordinate terrestri di un nodo dato a fronte di una lista di beacon rilevati;

- *GPSNeighbors*: si occupa di fornire le coordinate terrestri corrispondenti alla posizione di un nodo nello spazio.

Ad ogni aggiornamento di posizione di un nodo, grazie alla funzione di lookup, identifica le coordinate terrestri corrispondenti.

Listing 6.4: Funzione lookup

```
private GenericPosition<? extends ICoordinates>
    lookup(IGeoTech geoTech) {

    if (integerCoordinates) {
        return lookupIntCoords(geoTech.getTag());
    } else {
        if (geoTech.getTag() != -1) {
            return new GenericPosition<ICoordinates>(
                nfcNeigh.getMyPosition(geoTech));
        } else if (geoTech.getBeacons() != null) {
            return new GenericPosition<ICoordinates>(
                (ICoordinates) bleNeigh.
                    getMyPosition(geoTech));
        } else if (geoTech.getGPSCoords() != null) {
            return new GenericPosition<ICoordinates>(
                gpsNeigh.getMyPosition(geoTech));
        } else {
            return null;
        }
    }
}
```

Inoltre questa classe è importante per il GeoEmitter, sia per identificare le coordinate terrestri corrispondenti ai nodi rilevati e le relative distanze da quest'ultimi, sia per verificare l'effettiva vicinanza rispetto a questo nodo, in base alla soglia specificata.

Listing 6.5: Funzione verifyNeighborhood

```
public NodeInfo<C> verifyNeighborhood(Node
    otherNode, IGeoTech geoTech) {
    if (myNodeInfo.getPosition()==null){
        return null;
    }

    GenericPosition<C> pos = (GenericPosition<C>)
        lookup(geoTech);

    if (neighborhoodRelation.isNeighbor((
        GenericPosition<C>) (myNodeInfo.getPosition()
    ), pos)) {
```

```

        return new NodeInfo<C>(otherNode, pos,
            distance(myNodeInfo.getPosition(), pos));
    }

    return null;
}

```

### 6.2.3 GeoEmitter

Per quanto riguarda il sottosistema *GeoEmitter* le modifiche principali riguardano la classe *ConnectionLessEmitterManager*, con l'aggiunta delle classi *Emitter\_node* e *GeoMsg*.

La classe Java *ConnectionLessEmitterManager* è il componente del sistema che funge da interfaccia tra *BaseApp* ed il proprio vicinato, fornendo le identità e le distanze dei vicini, e consentendo l'invio e la ricezione degli stati NBR. Il servizio mantiene una mappa che associa ogni nodo del vicinato al corrispondente ultimo stato noto. In questa classe si sfrutta in particolare il protocollo di comunicazione UDP connectionless. Esiste una versione che utilizza TCP, *ConnectedEmitterManager*, ma per questa integrazione è stato scelto di lasciarla da parte.

Nel dettaglio questa classe si occupa di:

- creare una socket per la comunicazione con gli altri nodi Client, quindi con gli altri *ConnectionLessEmitterManager*);
- lanciare un thread che si occupa di ricevere e immagazzinare i messaggi di tipo *Emitter\_nbrTable* provenienti dagli altri nodi Client;
- lanciare un thread che esegue un ciclo infinito in cui si vanno a recuperare i messaggi immagazzinati in precedenza in modo da aggiornare la tabella del vicinato con le informazioni aggiornate riguardanti i nodi vicini;
- fornire un metodo per poter trasmettere a tutti i nodi del vicinato la nuova tabella con le informazioni aggiornate (messaggi di tipo *Emitter\_nbrTable*).

Il *ConnectionLessEmitterManager* si occupava anche di inserire e rimuovere dalla mappa dei vicinati le informazioni relative ai nodi segnalati dal *NeighborhoodManager* rispettivamente con i metodi *onNewNeighbor* e *onLostNeighbor*, in modo da mantenere una tabella del vicinato sempre aggiornata e consistente.

Questo viene sostituito dalla segnalazione da parte del NeighborhoodManager, con il metodo *onNeighborhoodWiFi*, della nuova localizzazione, grazie al quale in modalità WiFi ad-hoc si propaga ad ogni ciclo computazione un oggetto di tipo *GeoTech* contenente l'eventuale tag NFC, eventuali beacon rilevati e posizione GPS.

Viene creato un nuovo tipo di messaggio *Emitter\_node*, che estende *EmitterMsg*. In questo modo la logica comunicativa dei messaggi di tipo *Emitter\_nbrTable* resta invariata e il thread che si occupa di recuperare i messaggi può discriminare l'azione da eseguire, tra *updateTable* e *updateNodes*.

Il messaggio di tipo *Emitter\_node* inviato in broadcast ha come payload un oggetto di tipo *GeoMsg*, il quale comprende anche la stringa contenente il nome dell'applicazione in esecuzione, per poter selezionare solo i nodi a cui si è interessati. Questi sono i relativi metodi pubblici:

Listing 6.6: Messaggio GeoMsg

```
public String getNameApp(){
    return nameApp;
}

public GeoTech getGeoTech(){
    return geoTech;
}
```

Per l'invio e la ricezione in broadcast si utilizzano due socket diverse dall'EmitterSocket, come visto nel capitolo precedente. La serializzazione dei messaggi avviene nel formato JSON, tramite la libreria Gson [3].

Listing 6.7: Funzione onNeighborhoodWiFi

```
public void onNeighborhoodWiFi(final IGeoTech
    geoTech) {
    if (!rpi){
        new Thread() {
            public void run() {
                updateNBR();
                upToDateSelfInfo = neighborhoodService.
                    getNodeInfo();
                Emitter_node msg = new Emitter_node(
                    whoAmI);
                GeoMsg geoMsg = new GeoMsg(nameApp, (
                    GeoTech) geoTech);
```

```

        msg.setPayload(jsonNode.toJSONString(geoMsg))
        ;
        String encodedMsg = jsonNodeTransport.
        toJSON(msg);
        // Invio in broadcast
        byte[] data = encodedMsg.getBytes()
        ;
        DatagramPacket packet;
        try {
            packet = new DatagramPacket(data,
                data.length, InetAddress.
                getByName(BROADCAST_ADDRESS),
                UDP_PORT);
            sendSock.send(packet);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        v("onNeighborhoodWiFi: broadcast node
            message");
    }
    }.start();
}
}

```

In caso di modalità con integrazione di Raspberry Pi per il BLE questo metodo viene sostituito da *onNeighborhoodBLE*, notificato al termine di ogni ciclo di scansione dal NeighborhoodManager, con la lista aggiornata dei nodi vicini. In questo caso si ha a disposizione la lista dei nodi, con indirizzi ip e porte delle EmitterSocket di ciascun nodo vicino.

Listing 6.8: Funzione onNeighborhoodBLE

```

public void onNeighborhoodBLE(final ArrayList<Node>
    nodes, final IGeoTech geoTech) {
    if (rpi){
        new Thread() {
            public void run() {
                updateNBR();
                GeoMsg geoMsg = new GeoMsg(nameApp, (
                    GeoTech) geoTech);
                String payload = jsonNode.toJSONString(geoMsg
                );
                for (int i=0; i<nodes.size(); i++) {
                    Node n = nodes.get(i);
                    if (n.equals(whoAmI)) {

```

```
    upToDateSelfInfo =
        neighborhoodService.
        getNodeInfo();
    } else {
        Emitter_node msg = new
            Emitter_node(whoAmI);
        msg.setPayload(payload);
        String encodedMsg =
            jsonNodeTransport.toJSON(msg);
        try {
            emitterSocket.sendMessage(new
                Message<IEmitterProtocol.
                Type>(Type.EMITTER_NODE,
                encodedMsg), new
                InetSocketAddress(n.getIp()
                    , n.getPort()));
        } catch (IOException e) {
            e.printStackTrace();
        }
        v("onNeighborhoodBLE: node
            message sent to " + n.getId());
        ;
    }
}
}.start();
}
}
```

La differenza sostanziale nella computazione del vicinato è quella che non si ha una centralizzazione, quindi la mappa viene costruita in questa classe grazie ai messaggi di tipo *Emitter\_node* ricevuti ad ogni ciclo computazionale.

I messaggi ricevuti vengono filtrati in modo da aggiungere come vicini solo i nodi all'interno della soglia definita per il vicinato.

Listing 6.9: Funzione updateNodes

```
public void updateNodes(Emitter_node command) {
    v("about to decode " + command.getPayload());
    GeoMsg geoMsg = jsonNode.fromJSON(command.
        getPayload());
    if (geoMsg.getNameApp().equals(nameApp)) {
        NodeInfo<C> identifier = neighborhoodService.
            verifyNeighborhood(command.getId(), geoMsg
                .getGeoTech());
        if (identifier!=null) {
            synchronized (nbr) {
```

```

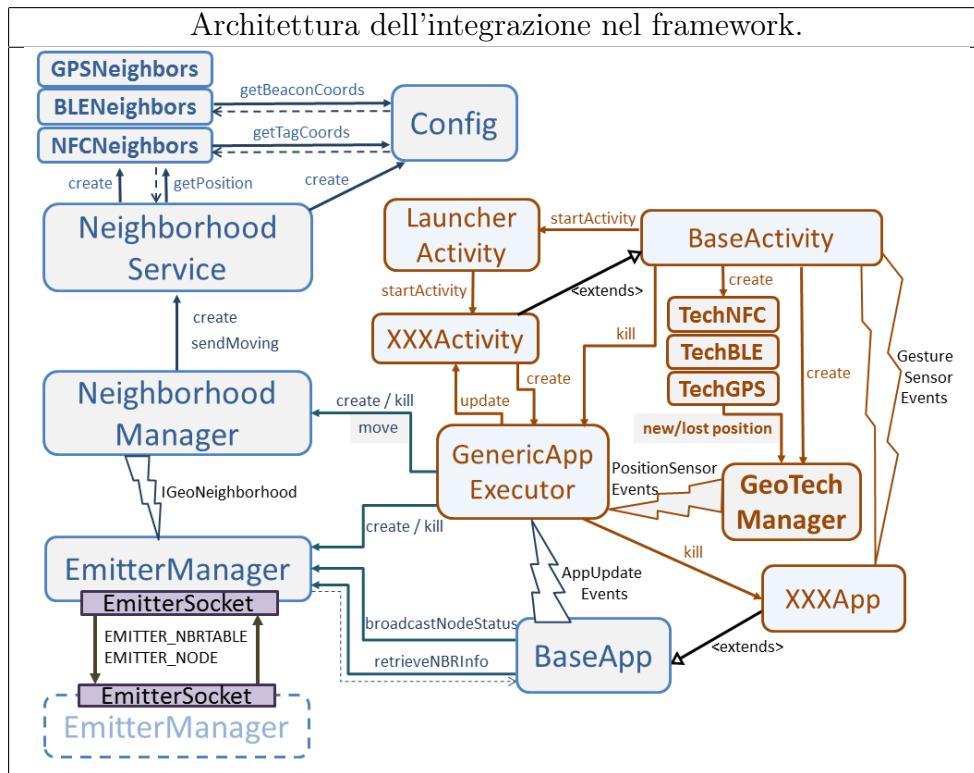
        T data = null;
        for (NodeInfo<C> ni : nbr.keySet()) {
            if (ni.getNode().equals(identifier.
                getNode())) {
                data = nbr.remove(ni);
                break;
            }
        }
        nbr.put(identifier, data);
        tempNodes.remove(identifier.getNode());
    }
}
}
}
}
}
}
}
```

Una citazione particolare va fatta al meccanismo di aggiornamento del vicinato. Si sceglie di non resettarlo continuamente ad ogni ciclo ma vengono rimossi i nodi dai quali non si ricevono più messaggi per un certo numero di cicli computazionali (ad esempio 3). Questo garantisce maggiore stabilità, soprattutto nel caso di scansione BLE.

Listing 6.10: Funzione updateNBR

```

private void updateNBR(){
    if (count==0){
        synchronized (nbr) {
            for (int i=0; i<tempNodes.size(); i++){
                for(Iterator<Map.Entry<NodeInfo<C>, T>>
                    it=nbr.entrySet().iterator();it.
                    hasNext();){
                    Map.Entry<NodeInfo<C>, T> entry = it
                        .next();
                    if (entry.getKey().getNode().equals(
                        tempNodes.get(i))){
                        it.remove();
                    }
                }
            }
            for (NodeInfo<C> ni : nbr.keySet()) {
                tempNodes.add(ni.getNode());
            }
        }
    }
    count = (count + 1) % removeNodeStep;
}
```





# Capitolo 7

## Test

Vengono eseguiti dei test sul sistema realizzato utilizzando i Nexus 7 in dotazione. In particolare si mostrano due applicazioni eseguite in differenti scenari.

### 7.1 Modalità WiFi ad-hoc

Viene qui utilizzata l'applicazione Flagfinder per mostrare un possibile utilizzo in uno scenario outdoor con WiFi ad-hoc. In questo modo il GPS dovrebbe riuscire a garantire una certa precisione e l'assenza di beacon e tag NFC non dovrebbe comportare problemi rilevanti di localizzazione.

L'applicazione viene adattata per un utilizzo più su larga scala, senza dover utilizzare il Magic Carpet. In base alla distanza rilevata dalla destinazione viene modificato il colore di sfondo.

Listing 7.1: Metodo chooseColor

```
private void chooseColor(double distance) {
    if (distance < 5.0) {
        getWindow().getDecorView().setBackgroundColor
            (Color.RED);
    } else if (distance >= 5.0 && distance < 10.0) {
        getWindow().getDecorView().setBackgroundColor
            (Color.rgb(255, 175, 5)); // orange
    } else if (distance >= 10.0 && distance < 15.0)
    {
        getWindow().getDecorView().setBackgroundColor
            (Color.YELLOW);
    } else if (distance >= 15.0 && distance < 20.0)
    {
```

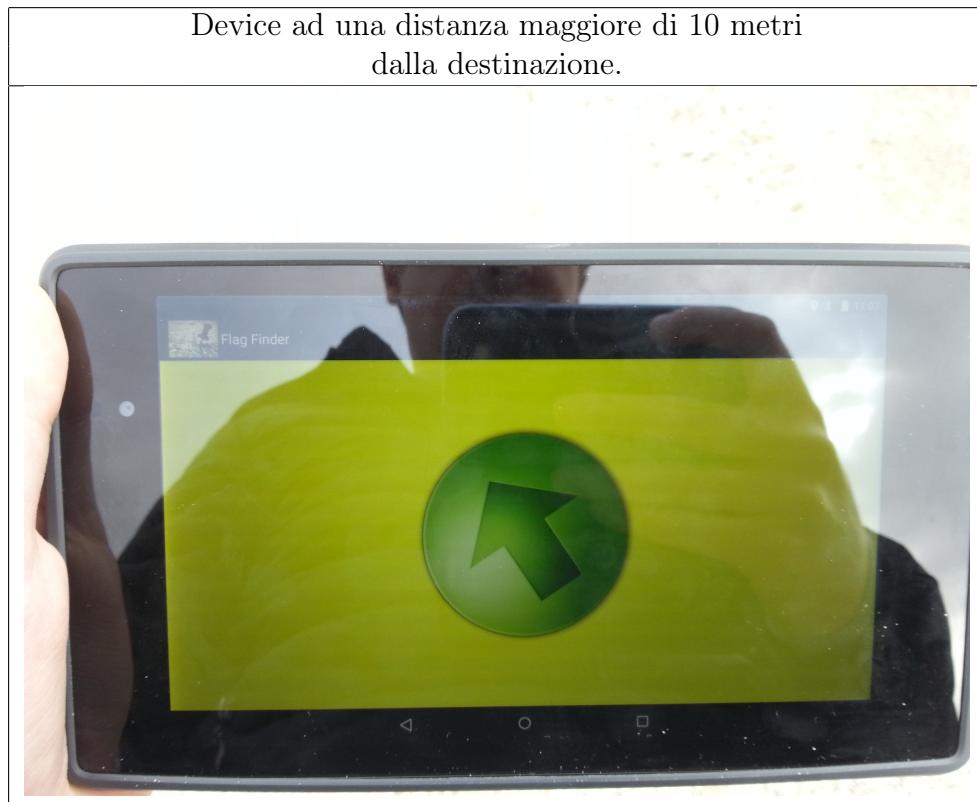
```
        getWindow().getDecorView().setBackgroundColor
            (Color.CYAN);
    } else if (distance >= 20.0) {
        getWindow().getDecorView().setBackgroundColor
            (Color.BLUE);
    }
}
```

Per motivi di tempo sono stati eseguiti pochi test in questa modalità e solamente con i Nexus 7. La localizzazione GPS spesso non è molto precisa e non è possibile sfruttare al meglio il *Network Location Provider* di Android. In futuro si potrà migliorare notevolmente questo aspetto.

Di seguito viene mostrato il device durante alcune fasi di ricerca della destinazione.

Device ad una distanza maggiore di 20 metri dalla destinazione.





## 7.2 Modalità BLE

In questo caso viene sfruttato un concetto simile a quello dell'applicazione Flagfinder. Sui dispositivi non viene visualizzata la freccia basata sui sensori e sulla posizione, visto che potrebbe dare informazioni sbagliate se la configurazione dei beacon non viene fatta ogni volta in modo da valutare eventuali percorsi obbligatori, soprattutto in scenari indoor. Si prevede solamente una semplice TextView che ne visualizza la distanza dalla destinazione.

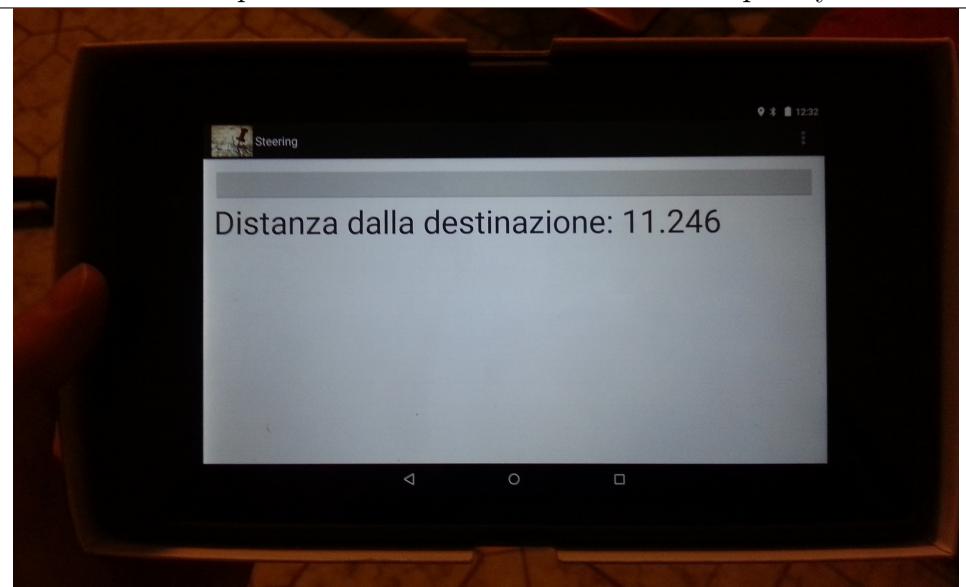
Listing 7.2: Metodo update

```
public void update(SteeringData state) {
    TextView tw = (TextView) ((Activity) context).
        findViewById(R.id.logView);
    myDistance = state.getDistDst();
    if (myDistance == 0) {
        tw.setText("");
        arrow.setImageResource(R.drawable.flag_marker
    );
```

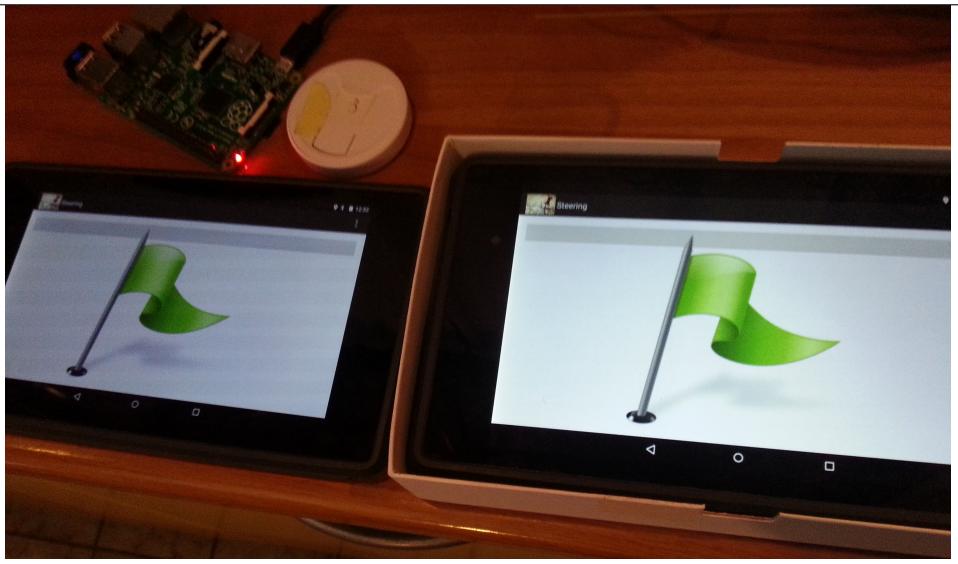
```
        getWindow().getDecorView().setBackgroundColor
            (Color.WHITE);
    } else {
        tw.setText("Distanza dalla destinazione: " +
            Math.round(state.getDistDst() * Math.pow
                (10, 3)) / Math.pow(10, 3));
        arrow.setImageDrawable(null);
    }
}
```

Si utilizza questa applicazione per mostrare un possibile utilizzo in uno scenario indoor con integrazione dei Raspberry Pi per l'advertise BLE. Vengono posizionati due beacon ad una distanza di circa 11 metri tra loro, un dispositivo con Raspberry Pi che funge da destinazione viene posizionato nei pressi di uno dei beacon, mentre un secondo device con Raspberry Pi è in moto e, in base allo spostamento, vede la distanza aggiornata sullo schermo. Quando si troverà nei pressi del dispositivo destinazione, visualizzerà sullo schermo la bandiera che indica l'arrivo.

Device alla ricerca della destinazione. Si mostra il particolare della scatola per realizzare il sistema Android-Raspberry.



Device arrivato a destinazione. Si mostrano le due componenti del dispositivo destinazione e il beacon per il posizionamento.





# Capitolo 8

## Conclusioni e sviluppi futuri

In questa tesi si è partiti dall’idea del framework Magic Carpet e dalla rassegna “il DISI incontra il Mambo” per produrre un sistema che permettesse di mostrare alcuni elementi dell’approccio computazionale *spatial computing*, miscelando elementi scientifici e di intrattenimento con la realizzazione di applicazioni basate esclusivamente su interazioni locali.

L’idea di realizzare comunicazioni di prossimità senza l’ausilio di una centralizzazione ha portato ad investigare sulle possibili tecnologie adatte a questo scopo.

- Il Bluetooth Low Energy si presenta come un ottimo candidato per quanto riguarda il discovery dei nodi vicini e rispetto alla specifica standard ha diversi vantaggi, tra cui la possibilità di fare advertise senza necessità di instaurare connessioni e i consumi energetici ridotti. Purtroppo ci si è dovuti scontrare con un limite tecnologico, in quanto i tablet Nexus 7 in dotazione non hanno al momento il supporto al ruolo di *Peripheral* che si occupa proprio dell’advertise. Questo ha comunque dato la possibilità di esplorare il mondo dei dispositivi embedded, in particolare il Raspberry Pi, utilizzato in questa tesi come supporto per colmare proprio questa mancanza. Comunque in futuro questa potrebbe rivelarsi una buona scelta, soprattutto potendo contare su di un buon supporto da parte di Android.
- Il WiFi ad-hoc presenta anch’esso caratteristiche adatte, in quanto si possono realizzare reti i cui utenti sono solamente i nodi del sistema di interesse e in cui non si ha nessuna centralizzazione. Le comunicazioni realizzate via UDP sono molto veloci e grazie al broadcasting è comunque possibile ricevere dati dai

nodi vicini. I limiti rilevati riguardano anche in questo caso il supporto sui dispositivi in dotazione, inoltre vi è un problema nell'utilizzo contemporaneo del BLE nel framework per rilevare i beacon presenti nell'ambiente.

Come si può notare questo studio verso le comunicazione opportunistiche può avere grandi spinte, date dallo sviluppo tecnologico. Lo scenario delle reti di comunicazione di tipo wireless sta rapidamente evolvendo verso i sistemi pervasivi in cui i dispositivi wireless, di diversi tipi e grandezze, costituiscono parte integrante dell'ambiente in cui sono immersi, ed interagiscono continuamente ed in maniera trasparente con gli utenti che vi vivono o che lo attraversano. Si parla a tal proposito anche di ambienti intelligenti e uno dei filoni di ricerca principali riguarda le reti opportunistiche.

Le reti opportunistiche non si appoggiano su alcuna infrastruttura fissa, né cercano di auto-configurarsi in una infrastruttura wireless temporanea costituita da nodi vicini. Sfruttano le opportunità di contatto che si verificano fra i nodi (dispositivi wireless di piccola taglia) trasportati dagli utenti nelle loro attività quotidiane (ad esempio a lavoro, sugli autobus, a scuola o all'università, ecc.).

I messaggi vengono scambiati ogni qualvolta si renda possibile, ovunque sia possibile ed il successo della loro trasmissione è strettamente legato alle dinamiche sociali in cui sono coinvolti gli utenti che trasportano i dispositivi ed alla storia degli incontri tra individui. Data la mobilità estremamente elevata che caratterizza questo nuovo scenario di reti, e la nota rumorosità delle comunicazioni wireless, l'affidabilità delle trasmissioni emerge come uno dei fattori di principale interesse. Infatti, le comunicazioni possono aver luogo soltanto durante i periodi di contatto tra i nodi e devono essere estremamente veloci ed efficaci.

Questo porta a dover fare uno sforzo di progettazione per nuovi protocolli di comunicazione che si diversifichino da quelli oggi più diffusi e basati sulla ritrasmissione dei dati mancanti. Le ritrasmissioni infatti, nella maggior parte dei casi potrebbero non poter essere eseguite per mancanza di tempo. Una strategia valida per gestire l'affidabilità delle comunicazioni opportunistiche in simili scenari estremi (caratterizzati cioè da scarse risorse e scarsa connettività) prevede l'utilizzo combinato di tecniche di codifica dei dati e strategie di instradamento di tipo epidemico. Questo approccio sfrutta la ridondanza sia delle informazioni, sia dei percorsi. La ridondanza delle informazioni dà robustezza a fronte della perdita dei dati in rete poiché è necessario che soltanto un sottoinsieme dei codici generati arrivi a de-

stinazione per consentire la ricostruzione corretta delle informazioni. La ridondanza dei percorsi invece è necessaria poiché non è possibile prevedere la sequenza dei contatti che può portare i dati a destinazione e pertanto è necessario distribuire l'informazione in più direzioni.

Le reti opportunistiche, caratterizzate dalla presenza di dispositivi con limitata autonomia energetica e risorse limitate, offrono attualmente lo scenario che meglio traduce il concetto di sistemi pervasivi. Di particolare interesse è il caso delle reti di sensori sparse in cui i sensori sono disposti nell'ambiente con funzione di monitoraggio ed i dati che collezionano vengono raccolti da degli agenti mobili che passano nelle vicinanze. Questi agenti possono utilizzare le informazioni acquisite dai sensori per eseguire applicazioni dipendenti dal contesto o possono semplicemente inoltrarle fino a quando raggiungono l'infrastruttura dove vengono elaborati e memorizzati.

Le interazioni fra i sensori immersi nell'ambiente e gli agenti sono soltanto un primo passo di un sistema di comunicazione globale completamente opportunistico in cui quest'ultimi si scambiano le informazioni che trasportano fino a quando i dati pervengono alle destinazioni più lontane. In questo scenario, le comunicazioni wireless completano naturalmente le interazioni tra gli utenti e si verificano ogni qualvolta gli utenti si incontrano oppure si avvicinano casualmente, dovunque questa interazione avvenga.

Per supportare un simile framework, è necessario sviluppare nuovi paradigmi di comunicazione che tengano in considerazione l'assenza di link stabili tra i nodi che comunicano (connettività intermittente) e che assumano quindi la disponibilità di brevi periodi di contatto per comunicare. Inoltre i nuovi paradigmi di comunicazione devono generalmente assumere l'assenza di un percorso completo fra i nodi sorgente e destinatario e sfruttare invece forme di instradamento delle informazioni che sono simili al modo in cui avvengono le interazioni sociali fra le persone. Strategie di instradamento basate su codifica dei dati offrono una valida soluzione per supportare il framework emergente dei sistemi pervasivi.

Quanto presentato in questa tesi vuole essere un orientamento in questo campo in grande fermento, una “rivoluzione” tecnologica verso una nuova forma di interazione “smart” con tutto ciò che ci circonda e caratterizza la nostra vita quotidiana, il cosiddetto “Internet of Things”. Secondo una stima della società Gartner, nel 2020 ci saranno 26 miliardi di oggetti connessi a livello globale, ABI Research stima che saranno più di 30 miliardi. Altri istituti parlano di 100 miliardi.

Le aspettative degli esperti sono che “Internet of Things” cambierà il nostro modo di vivere in modo radicale. Gli oggetti intelligenti, con capacità decisionale, permetteranno risparmio energetico sia a livello personale (domotica e smart-home) sia a livello macroscopico (smart-city e smart grid).

# Bibliografia

- [1] Android developer bluetooth. <http://developer.android.com/guide/topics/connectivity/bluetooth.html>.
- [2] Android developer bluetooth low energy. <http://developer.android.com/guide/topics/connectivity/bluetooth-le.html>.
- [3] A java library to convert json to java objects and vice-versa. <https://code.google.com/p/google-gson/>.
- [4] A node.js module for implementing ble peripherals. <https://github.com/sandeepmistry/bleno>.
- [5] Raspberry pi foundation what is a raspberry pi? [www.raspberrypi.org/help/what-is-a-raspberry-pi/](http://www.raspberrypi.org/help/what-is-a-raspberry-pi/).
- [6] Wikipedia. bluetooth - wikipedia, the free encyclopedia. <http://en.wikipedia.org/wiki/Bluetooth>.
- [7] Wikipedia. bluetooth low energy - wikipedia, the free encyclopedia. [http://en.wikipedia.org/wiki/Bluetooth\\_low\\_energy](http://en.wikipedia.org/wiki/Bluetooth_low_energy).
- [8] Wikipedia. raspberry pi - wikipedia, the free encyclopedia. [http://en.wikipedia.org/wiki/Raspberry\\_Pi](http://en.wikipedia.org/wiki/Raspberry_Pi).
- [9] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F Knight Jr, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.
- [10] Khaled A Ali and Hussein T Mouftah. Wireless personal area networks architecture and protocols for multimedia applications. *Ad Hoc Networks*, 9(4):675–686, 2011.

- [11] Jonathan Bachrach, James McLurkin, and Anthony Grue. Protoswarm: a language for programming multi-robot systems using the amorphous medium abstraction. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, pages 1175–1178, Estoril, Portugal, 2008. International Foundation for Autonomous Agents and Multiagent Systems.
- [12] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. *arXiv preprint arXiv:1202.5509*, 2012.
- [13] Jacob Beal and Richard Schantz. A spatial computing approach to distributed algorithms. In *45th Asilomar Conference on Signals, Systems, and Computers*, pages 1–5, 2010.
- [14] Davide Ensini. Spatial computing per smart device. Master’s thesis, Scuola di Ingegneria e Architettura - Università di Bologna, dec 2014.
- [15] H. Kasinger F. Dötsch, J. Denzinger and B. Bauer. Decentralized realtime control of water distribution networks using self-organizing multi-agent systems. In *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference*, pages 223—232. IEEE, 2010.
- [16] Andrea Fortibuoni. Sviluppo di una infrastruttura location-based per l’auto-organizzazione di smart-devices. Master’s thesis, Scuola di Ingegneria e Architettura - Università di Bologna, jul 2014.
- [17] Sara Montagna, Mirko Viroli, Matteo Risoldi, Danilo Pianini, and Giovanna Di Marzo Serugendo. Self-organising pervasive ecosystems: A crowd evacuation example. In ElenaA. Troubitsyna, editor, *Software Engineering for Resilient Systems*, volume 6968 of *Lecture Notes in Computer Science*, pages 115–129. Springer Berlin Heidelberg, 2011.
- [18] Danilo Pianini, Sara Montagna, and Mirko Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference*, pages 667–674. IEEE, 2011.

- [19] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.
- [20] Luca Santonastasi. Libreria per comunicazioni ad-hoc per android basate su bluetooth. Master’s thesis, Scuola di Ingegneria e Architettura - Università di Bologna, dec 2014.
- [21] Ahmad Usman and Sajjad Haider Shami. Evolution of communication technologies for smart grid applications. *Renewable and Sustainable Energy Reviews*, 19:191–199, 2013.
- [22] Mirko Viroli, Jacob Beal, and Kyle Usbeck. Operational semantics of proto. *Science of Computer Programming*, 78(6):633–656, 2013.



# Ringraziamenti

Si conclude la mia lunga carriera universitaria, un percorso non certo semplice ma che mi ha dato delle belle soddisfazioni e mi ha fatto crescere da tutti i punti di vista.

Desidero innanzitutto ringraziare il Professor Viroli per avermi dato la possibilità di esplorare un mondo a cui sono particolarmente interessato, a partire dall'esperienza al MAMbo fino ad arrivare a questa tesi.

Ringrazio Davide per i consigli e la collaborazione al progetto Magic Carpet, senza dimenticare Andrea F. e Andrea L., i miei compagni di innumerevoli progetti in questa laurea magistrale.

Tutto questo è stato reso possibile grazie al supporto della mia famiglia, soprattutto nei momenti più difficili. Infine un ringraziamento speciale va a Barbara.