Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

SPATIAL COMPUTING PER SMART DEVICES

Tesi in
Linguaggi e Modelli Computazionali LM

Relatore:
Chiar.mo Prof.
Mirko Viroli

Presentata da:
Davide Ensini

Correlatore:
Dott.
Danilo Pianini

II Sessione
Anno Accademico 2013/2014

Introduzione

Mentre continua a crescere la spinta all'interconnessione dei dispositivi che popolano il nostro vivere quotidiano, aumenta il bisogno di piattaforme e di paradigmi computazionali che assistano lo sviluppo del software, consentendo di gestire efficacemente la necessaria *context-awareness* ed il carattere sempre più opportunistico delle interazioni. La disponibilità di nuove tecnologie e lo slancio creativo, latore di nuove applicazioni, si alimentano reciprocamente concorrendo all'affermazione dello scenario del *pervasive computing*. Su tale scenario si focalizza questo lavoro di tesi, che si spinge in particolare verso il campo dello *spatial computing*, caratterizzato da un genere di applicazioni ed una densità di *device* connessi tali da permettere di utilizzare, per la computazione, una metafora che trascende gli stati dei singoli nodi: si volge l'attenzione allo spazio fisico in cui essi sono immersi, come un campo continuo in cui le variabili di interesse assumono valori.

All'interno di questo contesto, diversi obiettivi determinano l'evolversi del progetto. Primo per ordine di arrivo è *Il DISI incontra il MAMbo: il "bello" dell'Informatica*, il momento di incontro con la cittadinanza in occasione della presentazione del nuovo Dipartimento; in vista di tale evento studenti e ricercatori sono invitati a proporre manufatti da esporre, con il requisito di accompagnare la base scientifica ad elementi che suscitino stupore ed interesse, anche e soprattutto nei non addetti ai lavori. La scelta di cogliere questa opportunità porta con sé la necessità di produrre qualcosa di immediatamente tangibile e familiare, in grado di essere apprezzato da una platea ampia, che permetta di presentare tecnologie e principi dello *spatial computing* intrattenendo i visitatori. Da questo l'idea di

mostrare alcuni semplici modelli di auto-organizzazione, permettendo agli intervenuti di agire sugli elementi del sistema e di osservare le reazioni dovute allo spostamento o modifiche nella configurazione dei nodi. Le scelte successive sono in questa direzione: usare tablet e smartphone come nodi, con la possibilità per gli utenti di inserire i propri device nel sistema; creare un "tappeto magico" per la localizzazione dei dispositivi, denominato, così come tutto il sistema a partire da esso, *magic carpet*; produrre varie app da mostrare, con l'enfasi in alcuni casi sugli aspetti scientifici, in altri su quelli ludici.

Trascorso *Il DISI incontra il MAMbo*, il lavoro prosegue con nuovi obiettivi: mantenendo l'attenzione sullo spatial computing, si amplia lo sguardo a tecnologie e metodologie per l'applicazione a contesti reali e pratici. Ne è esempio la Smart-mobility, ovvero la guida intelligente di traffico pedonale o automobilistico: un caso pratico è la ricerca di amici durante un evento massivamente partecipato, in caso di mancanza di connettività ad ampio raggio; o la guida di pedoni in ambienti affollati come i musei, gestita in modo da permettere di raggiungere i punti interesse evitando ingorghi. Un altro scenario in cui lo spatial computing riveste grande importanza è quello del monitoraggio e controllo ambientale. Fanno esempio di tale contesto il controllo dinamico della distribuzione di energia in base ai bisogni, la gestione di una flotta di droni con cui si vuole dare copertura ad un'area di interesse.

Un aspetto osservato da vicino è quello delle tecnologie per il posizionamento. Oltre al "tappeto magico", costituito da una rete di *tag NFC*, si desidera rendere il sistema aperto alla cooperazione di più tecnologie per la localizzazione dei nodi. NFC diventa solo una delle alternative, grazie anche al lavoro dell' Ing. Andrea Fortibuoni, che con la sua tesi collabora all'acquisizione le tecnologie GPS e *Bluetooth Low Energy*, aprendo le porte ad applicazioni su scala geografica più ampia.

L'ultimo pezzo del puzzle è l'abilitazione alla programmazione dei nodi con un *domain specific language (DSL)*, da ottenersi 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 con-

cettuali e rende lo sviluppo più agile ed efficace. Inoltre Protelis incorpora come core il Field Calculus, che rende possibile una analisi formale degli algoritmi e dei pattern realizzati.

Magic Carpet, nato come un middleware orientato a una dimostrazione sullo spatial computing, che inizialmente coinvolgeva solo smart devices ed un tappeto di tag NFC, è dunque stato il punto di partenza per uno studio sulle tecnologie abilitanti in tale campo, che lo ha trasformato in qualcosa di diverso: una *tool-chain* per lo sviluppo e la distribuzione, su dispositivi connessi, di applicazioni di spatial computing. Essa comprende un interprete per un DSL basato su un core calculus formalizzato, e un middleware che supporta l'astrazione curando, a basso livello, le comunicazioni con il vicinato e le percezioni ambientali. Il sistema è utilizzabile su qualsiasi piattaforma basata su *Java*, ed è in grado di sfruttare NFC, Bluetooth e GPS come tecnologie di localizzazione, con la possibilità di integrarne altre in modo agevole. Al supporto per la distribuzione vera e propria si accompagna un simulatore per lo sviluppo in locale.

Indice

Capitolo 1

Spatial computing

1.1 Programmare lo spazio

"Spatial computing" non ha, nel momento in cui si scrive, una voce sull'enciclopedia wikipedia. Alla computazione si associano invece gli aggettivi "ubiquitous", "amorphous", "physical", "natural", "biological", e ovviamente i più generali "parallel" e "distributed". Nel gioco delle definizioni sarebbe lecito dire che, rispetto a quelle citate, il calcolo orientato allo spazio ne completa alcune, ne specializza altre, di altre ancora è sinonimo.

Questa varietà di approcci alla computazione si deve all'espressione di tagli prospettici diversi ed alla contaminazione con altre discipline, dalle più pure alle più orientate al mercato. Uno strumento interessante per la comprensione di come "spatial computing" si relazioni con le altre denominazioni è presentato in [6], dove si individua, come proprietà che maggiormente accomuna i vari domini, la stretta correlazione tra la computazione e la disposizione dei calcolatori nello spazio. Da questo si arriva a chiamare "spatial computers" gruppi di calcolatori disposti in uno spazio fisico, per i quali:

- la difficoltà di tramettere informazioni tra coppie di device dipenda fortemente dalla loro distanza, e
- gli obiettivi funzionali del sistema siano generalmente definiti in termini della struttura spaziale del sistema.

Questa e altre definizioni [8] mettono in risalto i due elementi che consideriamo fondanti per il modello a cui questa tesi fa riferimento: la **località delle interazioni** e la **densità elevata dei nodi**.

Tali vincoli si applicano ad un numero di situazioni destinato a crescere rapidamente con la riduzione delle dimensioni dei dispositivi connessi e con la loro sempre maggiore diffusione. Ne sono esempio le reti in cui numerosi sensori indipendenti si coordinano per convogliare le informazioni raccolte. Alla fine degli anni '90 un gruppo di ricercatori del MIT [2] spingeva la fantasia ai decenni a venire, immaginando di miscelare circuti, microsensori ed attuatori ai materiali da costruzione, ottenendo ponti in grado di fornire informazioni su carico di vento ed integrità strutturale, o in biologia programmare cellule come vettori per sostanze farmaceutiche o per produrre manufatti nell'ordine di grandezza dei nanometri. Non serve così tanta lungimiranza oggi per vagheggiare, ad esempio, l'idea di stormi di droni che, alla maniera degli uccelli, si avvantaggino sulle correnti costruendo formazioni elaborate, coordinandosi senza bisogno di guardare molto più in là del proprio becco.

Vent'anni fa i limiti tecnologici frenavano la realizzazione di sistemi come questi, ma ciò 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, si è arrivati 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 sullo spazio in cui essi sono distribuiti, non è più conveniente programmare pensando al singolo nodo, ora programmiamo lo spazio.

L'idea di base è modellare il sistema come uno spazio continuo, anziché come una rete, per manipolare programmaticamente regioni dello spazio geometrico anziché dispositivi individuali [7]. Questo rende impliciti molti dei dettagli del programma, consentendo allo sviluppatore di lavorare più direttamente sulla logica dell'applicazione di interesse, invece che costruire tutto da zero in termini di dispositivi che si scambiano bit. In questo modo non solo si semplifica lo sviluppo ma, dal momento che l'astrazione spaziale è portatrice di adattabilità a vari livel-

li, possiamo costruire algoritmi distribuiti che affrontano in modo naturale molti problemi altrimenti considerati difficili.

1.1.1 Il medium amorfo

L'astrazione per cui parliamo di "programmare lo spazio" vede ogni punto dello spazio stesso come occupato da un calcolatore. Ognuno ha un vicinato entro il quale può conoscere lo stato degli altri device. L'informazione si propaga nel medium ad una velocità massima c, per cui ogni calcolatore conosce non gli stati attuali dei propri vicini, ma gli stati passati. 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.

Questa astrazione può ovviamente essere solo approssimata nel mondo reale, ed è proprio quello che facciamo adottando questa metafora: approssimiamo uno spazio costituito da infiniti calcolatori, con una rete discreta di calcolatori. Ogni device rappresenta così una piccola regione dello spazio intorno a sé, e i messaggi trasmessi tra device vicini implementano il flusso di informazioni nei vicinati.

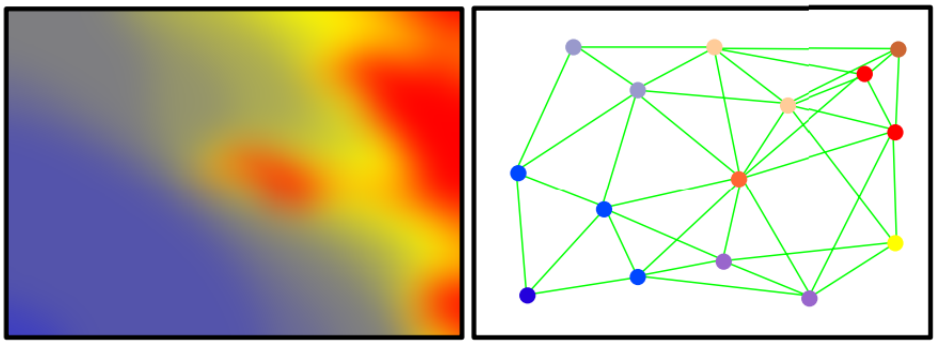


Figura 1.1: L'immagine rappresenta la temperatura in una stanza rettangolare. Disponendo una rete di sensori nella stanza, otteniamo una appossimazione discreta del campo di temperature, visualizzata nell'immagine a destra. Immagine tratta da [11].

Per costruire un sistema distribuito utilizzando questa astrazione occorre poter esprimere calcoli geometrici e flussi di informazione in regioni dello spazio, ed occorre poter disporre di qualcosa che traduca ed implementi tali specifiche su

una rete di dispositivi connessi. Ne costituisce un esempio notevole il sistema Proto.

1.1.2 Proprietà emergenti

I maggiori benefici derivanti dall'uso della metafora spaziale si identificano in scalabilità, robustezza e adattabilità [7].

Se uno dei device nella rete viene a mancare per qualche ragione, i suoi vicini si prendono carico in modo naturale dello spazio vuoto, semplicemente allargando le maglie dell'approssimazione discreta del medium amorfo. Allo stesso modo, aggiungere nuovi nodi rende semplicemente più fine la risoluzione dell'approssimazione. Qualora le variazioni nella densità portassero un device ad avere troppi o troppo pochi vicini, un semplice adattamento della soglia di vicinato potrebbe riportare il sistema ad una situazione più accettabile.

Un altro dei vantaggi immediati di questo approccio è che un sistema progettato per il medium amorfo può essere approssimato su superfici o spazi con conformazioni arbitrariamente complicate o estese (purché le si possa chiamare varietà reimanniane), semplicemente fornendo una definizione di vicinato adeguata alla situazione.

1.2 Proto

Proto funziona compilando un programma di alto livello (scritto nel linguaggio di programmazione Proto) in un programma locale che viene eseguito da ogni nodo nella rete. Il programma locale è eseguito nella Proto Virtual Machine, utilizzabile su varie piattaforme, incluso un simulatore.

Il linguaggio Proto incarna la metafora del medium amorfo discretizzato, è puramente funzionale ed utilizza una sintassi *LISP-like*. Le primitive di Proto sono operazioni matematiche su *field* (funzioni che associano ogni punto nello spazio con un valore) e sono di quattro tipi: calcoli "ordinari" locali (e.g. addizione), operazioni sul vicinato che implicano la comunicazione, operazioni di *feedback* che stabiliscono variabili di stato, ed operatori di restrizione che modu-

lano una computazione cambiandone il dominio. I programmi proto interagiscono con l'ambiente mediante sensori ed attuatori che misurano e manipolano lo spazio occupato dai dispositivi.

Si riporta qui un semplice programma Proto, con una spiegazione di massima, allo scopo di mostrarne l'orientamento alla spazialità e alle comunicazioni con il vicinato. Si rimanda, per una comprensione più approfondita, ad altre fonti, prima tra tutte il sito di riferimento per il linguaggio [17].

```
(def close (src)
   (let ((d (distance-to src)))
      (if (and (< d 10) (> d 1))
         (blue 1)
         (blue 0))))
```

La prima cosa che facciamo è definire la funzione `close` che accetta un argomento `src`. Successivamente assegnamo (`let`) alla variabile `d` il risultato della valutazione di `(distance-to src)`, che restituisce la distanza di cammino minimo da `src`. Per finire, mediante l'adattatore `blue`, accendiamo tutti i device entro una distanza definita.

Utilizzeremo un sensore per definire la regione `src`. `(sense 1)` legge un sensore con valore booleano il cui valore si può impostare, per ogni device, da interfaccia grafica. Il programma da valutare è:

```
(close (sense 1))
```

In figura 1.2 vediamo un sistema in cui ad ogni punto corrisponde un device. In arancione sono stati identificati i device per cui vale `(sense 1)`, quindi la regione sorgente, mentre sono colorati di blu quelli in prossimità di essa.

La parte più interessante per la comprensione del modello è costituita dall'implementazione di `distance-to`. Pur essendo fornita come funzione di libreria con la versione di riferimento di proto, `distance-to` non fa parte del linguaggio. Eccone una semplice implementazione.

```
(def distance-to (src)
    (rep dst inf (mux src 0 (min-hood+ (+ (nbr dst) nbr-range))
)))
```

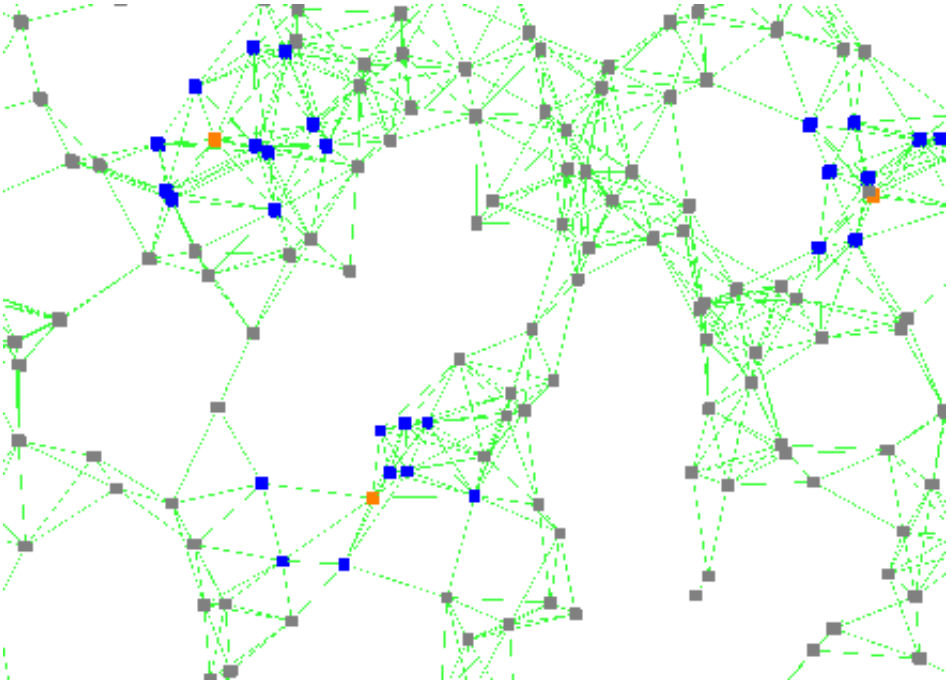


Figura 1.2: Simulazione della funzione `close`. In verde sono indicati i collegamenti esistenti, che realizzano il vicinato dei nodi.

Il costrutto `rep` stabilisce una variabile di stato di nome `dst` che prende inizialmente il valore di infinito positivo (`inf`). La parte restante dell'istruzione specifica come questo stato debba essere aggiornato nelle iterazioni successive.

Troviamo un `mux`, un costrutto che permette di specificare due espressioni diverse e scegliere, dopo averle valutate entrambe, quella indicata dalla condizione. La differenza con un `if` sta proprio nel fatto che entrambi i rami vengono valutati, il che fa sì che eventuali scambi di informazione con i vicini vengano effettuati anche nel ramo non scelto. In questo caso il `mux` si valuta in `0` laddove vale `src`, mentre in tutte le altre regioni il suo valore è definito dal terzo argomento della funzione, che osserva il vicinato.

Partendo dall'interno, si ha la somma di `(nbr dst)`, cioè i valori di `dst` condivisi da tutti i vicini, e `nbr-range`, cioè la distanza dai vicini stessi. Quello che viene sommato qui sono due field, non semplici valori scalari. La famiglia di funzioni `*-hood` lavora proprio sui field ed effettua vari tipi di aggregazione. In questo caso `min-hood+` sceglie il minore tra i valori individuati per tutti i vicini

del device, escludendo il valore del device stesso.

Quindi per ogni device in regione sorgente (cioè avente `(sense 1) = true`) la funzione `distance-to` restituirà il numero reale 0.0, mentre per ogni altro device si valuterà nella distanza di cammino minimo dalla sorgente più vicina.

Questo programma viene esequito ad ogni "round" computazionale, in modo asincrono sui vari device. Anche se la sola assunzione per il modello è che, finito un round, ad un certo punto ne inizi un altro, all'atto pratico la piattaforma assegnerà ai device una frequenza di esecuzione.

1.3 Esempi di pattern

Si mostrano qui alcuni pattern che saranno poi realizzati sul middleware oggetto di questa tesi, e che possono aiutare la comprensione della metafora computazionale adottata.

Gradient Il gradiente è sostanzialmente un alias del già visto `distance-to`. Si assegna ad ogni nodo il valore della sua distanza dalla regione sorgente. Di seguito il programma, piuttosto triviale, ed in figura 1.3 una simulazione del programma stesso.

```
(distance-to (sense 1))
```

Gradcast Gradcast o grad-value è forse la funzione dall'implementazione più complessa tra quelle che vediamo qui, ma il suo funzionamento è semplice. Essa fa sì che da una o più regioni dello spazio, identificate come sorgenti, venga diffuso un certo valore di riferimento, e che in ogni altra regione dello spazio i device facciano proprio il valore della sorgente più vicina. Si tratta quindi di una sorta di broadcast.

Per facilitarne la comprensione se ne mostra una implementazione semplice, anche se non completamente corretta.

```
(def grad-value (src f)
  (let ((d (distance-to src)))
    (rep v f (mux src f (2nd (min-hood (nbr (tup d v))))))))
```

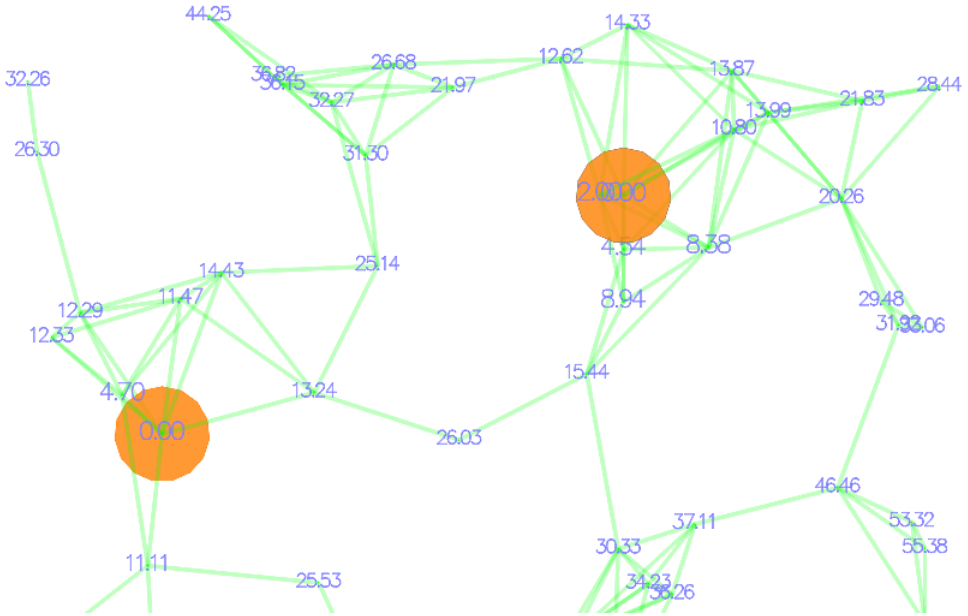


Figura 1.3: Un esempio di gradiente. Ogni dispositivo mostra la propria distanza dalla regione sorgente più vicina. Eseguito con il simulatore Proto MIT.

L'informazione condivisa nel vicinato è qui costituita da `(tup d v)`, una tupla contenente la distanza del nodo dalla sorgente più vicina ed il valore correntemente adottato per il gradcast. Quando un device è in regione `src` condividerà il proprio valore `f`, altrimenti assumerà quello selezionato dalla `min-hood`.

Il problema di questo approccio è che la mancata considerazione della propria distanza dai vicini può portare a scegliere, in situazioni limite, la sorgente sbagliata. Limiti del linguaggio impediscono una soluzione semplice a questo problema, che tuttavia risulta di poco conto nel momento in cui si osserva che l'errore tende ad annullarsi con l'aumentare della densità dei dispositivi.

Channel Chiamiamo "channel" la funzione che, scelti due estremi `src` e `dst`, individua tutti i dispositivi sul percorso minimo tra essi. Per realizzarlo facciamo originare una gradiente da ciascuno degli estremi. Poi, da uno dei due, lanciamo 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 `src` e `dst`.

```
(def channel (src dst)
  (<
    (+
        (distance-to src)
        (distance-to dst)
    )
    (+
        (grad-value(src (distance-to dst)))
        0.1
)))
```

Il valore 0.1 aggiunto al gradcast è una costante di tolleranza che serve ad evitare errori legati all'imprecisione dei calcoli in virgola mobile. In figura 1.4 una simulazione del seguente programma:

```
(blue (channel (sense 1) (sense 2)))
```

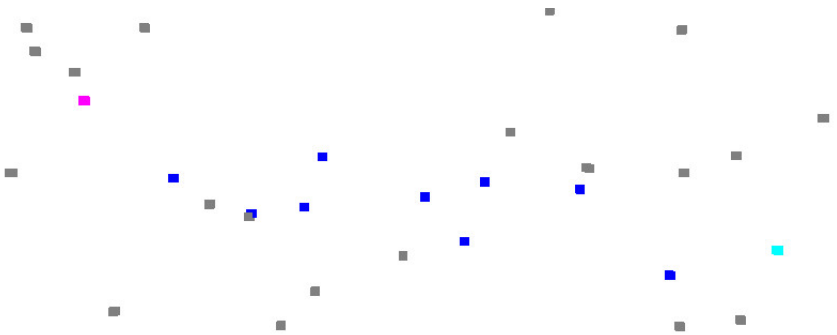


Figura 1.4: Simulazione di un channel realizzata con WebProto. I punti agli estremi, colorati diversamente, sono `src` e `dst`.

Partition Partition è una applicazione pressoché diretta del gradcast. La sua funzione è partizionare lo spazio creando varie regioni, ognuna associata ad una particolare sorgente, esattamente ciò che la funzione gradcast realizza. Dal momento che il codice applicativo aggiuntivo serve solamente ad associare un colore ad ogni nodo, non sarà qui riportato. Una simulazione di partition è comunque riportata in figura 1.5.

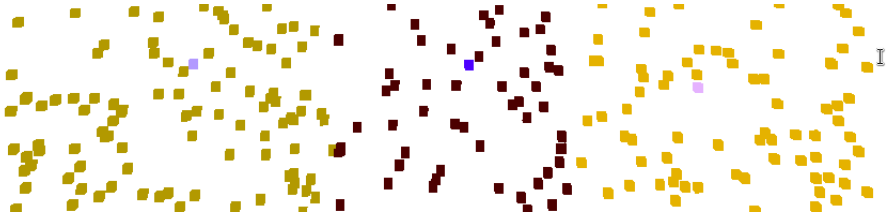


Figura 1.5: Esecuzione di partition sul simulatore WebProto. I punti di colore diverso sono i device `src`.

1.4 Applicazioni

Come si è anticipato, spatial computing è punto di incontro di varie discipline e filoni di ricerca. I pattern di coordinazione e auto-organizzazione individuati in chimica, in biologia, e altri campi, sono numerosissimi, come lo sono i tentativi di applicarli a nuovi problemi: i benefici in termini di stabilità, adattabilità e scalabilità sono molto invitanti. Di seguito si citano brevemente alcuni dei risultati prodotti, allo scopo di esemplificare le possibilità di applicazione della tecnologia.

Citiamo come primo esempio il *crowd steering*, la guida di folle in situazioni densamente popolate allo scopo di ottimizzarne il deflusso. In [15] si propone lo scenario di un museo in cui ad ogni visitatore viene fornito un dispositivo su cui sono caricate le sue preferenze in merito ai manufatti esposti. I dispositivi interagiscono tra loro ed utilizzando tecniche basate sulla diffusione di gradienti guidano gli utenti evitando le zone affollate. Un caso simile, ma in una situazione di emergenza, viene affrontato in [12].

In [4] il linguaggio Proto viene utilizzato, con alcuni adattamenti, per la guida di un gruppo di robot, avendo come obiettivo l'esplorazione di aree.

In [14] si mostra un semplice programma per il rendez-vouz di due persone ad un evento massivamente partecipato. La diffusione nello spazio computazionale di due gradienti consente l'elaborazione di un percorso per i due partecipanti che li porta ad incontrarsi a metà strada.

Un risultato notevole [9] vede spatial computing applicato ad un sistema multi-agente incaricato della gestione di una rete idrica. Tale sistema esibisce una capacità decisionale equiparabile a quella di operatori umani esperti, capacità che

emerge da una strategia di coordinazione ispirata alla biologia.

Anche le reti elettriche sono state studiate in questo ambito. In tali reti esiste già un controllo distribuito per quanto riguarda la negoziazione di energia con i nodi produttori. In [23] si propone un metodo *self-org* per la gestione delle fluttuazioni a breve termine di domanda ed offerta energetica, il cui obiettivo è migliorarne la stabilità.

In [22] si propongono algoritmi per gestire i collegamenti in una rete dinamica tridimensionale di sensori, in grado di garantire ottimalità nel consumo energetico mantenendo l'adattività a fronte dei cambiamenti nella rete.

Capitolo 2

Magic Carpet

2.1 La redazione “creativa” dei requisiti

Il punto di partenza per l’ideazione del sistema è *Il DISI incontra il MAMbo: il “bello” dell’informatica*: in occasione della giornata di presentazione alla cittadinanza di Bologna del nuovo Dipartimento di Informatica – Scienze e Ingegneria, vengono presentati presso il Museo di Arte Moderna di Bologna *(MAMbo)* prototipi e manufatti digitali per i quali si richiedono queste caratteristiche:

- *Fruibilità da parte di un pubblico non tecnico.*
- *WOW factor: i manufatti, e le loro installazioni, non devono avere necessariamente un carattere artistico ma devono colpire e attrarre il pubblico.*
- *Innovatività delle tecnologie impiegate.*
- *Fattibilità dell’installazione in un ambito museale.*
- *Interattività con l’audience.*

La scelta fatta è di mostrare alcuni semplici pattern, come channel e partition, nello spazio a disposizione, cioè su un tavolo, e con tablet commerciali, insieme ad alcuni giochi; il tutto cercando di incarnare l’astrazione dello spatial computing, per mantenere l’aspetto scientifico.

Su questa premessa, ed al fine di orientare le scelte successive, si apre una prima fase di ricerca di sperimentazione che affronta alcune tematiche tecnologiche

di rilievo. Tenendo presente che per la disponibilità di device e per le conoscenze acquisite si privilegerà la piattaforma Android, si cerca di dare risposta alle seguenti domande:

- come recuperare le informazioni relative alle distanze tra i device, o, come si rende necessario per le applicazioni ludiche, il posizionamento esatto sul tavolo?
- come realizzare comunicazioni esclusivamente locali, ipotizzando di definire vicini due device che distino meno di 20 cm?

2.2 Distanze tra device e posizionamento relativo

2.2.1 Le tecnologie per la localizzazione indoor

Il problema della localizzazione *indoor* è un tema molto caldo, grazie anche all'impulso dovuto alla progressiva realizzazione dello scenario denominato *"Internet of Things"*. Tuttavia nel nostro caso i requisiti sono piuttosto stringenti: occorre una precisione nell'ordine di pochi centimetri, un risultato ben superiore a quello tipicamente richiesto dalle applicazioni.

Si prendono in considerazione come prime alternative quelle che non richiedono l'utilizzo di infrastrutture o contributi esterni, ma si basano esclusivamente sulle tecnologie *a bordo* dei normali device in commercio.

La prima ipotesi è utilizzare gli accelerometri per misurare gli spostamenti in modo che, partendo da una configurazione nota, si possa mantenere una mappa della distribuzione dei nodi. L'idea di massima è quella di effettuare una doppia integrazione sui dati rilevati per ottenere lo spostamento, eventualmente con un filtraggio sui dati stessi. Tuttavia all'atto pratico si va incontro al fatto che la precisione dei sensori e la frequenza della rilevazione sono tali da non permettere una misura consistente, anche a fronte di un filtraggio avanzato. Ciò emerge dagli esperimenti fatti e trova conferma in letteratura, ampiamente popolata da casi in cui gli accelerometri sono utilizzati con successo per la localizzazione solo nell'ambito del *dead reckoning*, che li sfrutta in sinergia con i giroscopi per stimare

orientamento e lunghezza del passo di una persona che cammina, e tracciarne così il percorso.

La seconda possibilità presa in esame ipotizza di sfruttare il principio in uso nei sistemi sonar per stabilire la distanza tra due device; utilizzare cioè la trasmissione di suoni e la misurazione dei tempi che intercorrono tra emissione e ricezione, per ricavare, nota la velocità di propagazione del suono, la distanza tra sorgente e ricevitore; tutto ciò utilizzando il microfono e i diffusori già a bordo dello smart device.
La risoluzione della misura nello spazio dipende ovviamente dalla risoluzione della misura nel tempo. In condizioni ideali il limite alla risoluzione è legato esclusivamente alla frequenza di campionamento del segnale. Fissata la velocità di propagazione del segnale acustico a $340 m/s$, e la frequenza di campionamento al massimo consentito dalle API disponibili per la piattaforma Android, ovvero $44.1 kHz$, la risoluzione teoricamente raggiungibile è di poco superiore al centimetro.
Neanche in questa direzione si sono ottenuti risultati incoraggianti, per varie difficoltà dovute alla qualità dei microfoni e dei diffusori e per la configurazione particolarmente sfavorevole. Tecniche avanzate di analisi del segnale potrebbero essere utilizzate per superare questo problema; anche così facendo, sarebbe difficile garantire una misura veloce sui sistemi attualmente in commercio: al momento sono necessari svariati secondi. Si è valutato sconveniente proseguire su questa strada.

Ultima tra le ipotesi che non richiedono infrastruttura è l'idea di sfruttare l'intensità dei segnali a radiofrequenza.
In letteratura si trovano molte informazioni riguardo al *fingerprinting*, una tecnica largamente studiata e diffusa a livello industriale: il dispositivo da localizzare riceve segnali da varie fonti e ne misura l'intensità; nota la potenza di trasmissione si ottiene, attraverso varie tecniche, dalle più semplici a quelle che utilizzano varie soluzioni di intelligenza artificiale, una stima della posizione. Questa soluzione, che evidentemente richiede infrastruttura, viene citata tra quelle che non ne richiedono perché la precisione ottenibile con il fingerprinting rappresenta un limite su-

periore a quella che si può ottenere se si vuole cercare di ottenere la distanza tra due smartphone mediante l'intensità percepita di segnali wi-fi o bluetooth. Tra le soluzioni esaminate, nessuna è in grado di offrire la precisione cercata, nell'ordine dei centimetri; perciò questa strada viene, in questa fase, scartata.

Scartate tutte le ipotesi che facciano uso delle sole risorse a disposizione dei dispositivi, si esaminano le soluzioni che richiedono infrastruttura. È la categoria del sistema GPS e dei sistemi di localizzazione terrestri basati su celle per telefonia cellulare o ripetitori ad-hoc; dei sistemi che usano le radiofrequenze su scala più ridotta, ovvero quelli basati su wi-fi o bluetooth, o i sistemi basati su infrarossi. Ma per nessuno di questi sistemi sono stati reperiti in letteratura risultati tali da incoraggiarne l'uso per l'esigenza in essere.

Un'altra tecnologia in uso per la localizzazione sfrutta sensori ottici e riconoscimento di immagini, ed è all'apparenza una strada praticabile: si potrebbero elaborare in tempo quasi reale le immagini ricevute da una telecamera puntata sui device, i quali potrebbero eventualmente mostrare un marker di qualche tipo per facilitare il riconoscimento. Ciò darebbe risultati estremamente precisi. Tuttavia la scelta non è ricaduta su questa tecnologia.

La scelta finale è ricaduta sulla tecnologia NFC.

2.2.2 NFC

Near Field Communication (o NFC) è una tecnologia per le comunicazioni elettromagnetiche wireless a corto raggio, si parla tipicamente di pochi centimetri. Nasce nel 2004 con la fondazione di NFC Forum da parte di Nokia, Philips e Sony, ed è definita da un insieme di standard che ne specificano i protocolli di comunicazione ed il formato per lo scambio dei dati. Tali standard sono basati su quelli di RFID, Radio Frequency IDentification. La tecnologia è in grado di trasferire dati tra i device fino a $424 Kbit/s$, ed opera alla frequenza di $13.56 Mhz$. Per le sue caratteristiche di interoperabilità la tecnologia si può utilizzare con vari dispositivi (tastiere, fotocamere, SD card, console per videogiochi, etc.). Al momento, e grazie alla loro distribuzione globale, gli smartphone ed i tablet sono i principali

dispositivi abilitati all'uso di NFC, e anche i più adatti ad implementarne le tre modalità di funzionamento[13]:

- Nella modalità *peer-to-peer* si stabilisce una connessione tra due device, ed avviene uno scambio di dati
- Nella modalità *reader/writer* il device può leggere o scrivere informazioni come URL, SMS, numeri di telefono, etc. da un tag. In certe installazioni, ad esempio, gli utenti toccano con il proprio device un tag posto in un espositore, provocando la trasmissione di un URL al telefono. Questa URL può dirigere l'utente ad un sito web, dove potrà trovare ulteriori informazioni o scaricare coupon o *token* speciali.
- Nella modalità *card emulation* il device emula una carta *contactless*, permettendone l'utilizzo come strumento di pagamento, o come biglietto.

La modalità di interesse per il nostro scopo è la seconda. La localizzazione mediante la lettura di tag è una pratica già in uso nel mondo RFID, ma per distanze più grandi, tanto da spingersi in certi casi all'uso di tag attivi, cioè dotati di batteria, in contrasto con i tag passivi, cioè funzionanti grazie alla sola energia trasmessa dal device che sta effettuando la lettura. NFC compie un cambio di scala determinante per il nostro scopo: essendo la lettura di un tag possibile solo entro pochi centimetri, il fatto di essere in grado di leggere tale tag è un indice inequivocabile della prossimità.

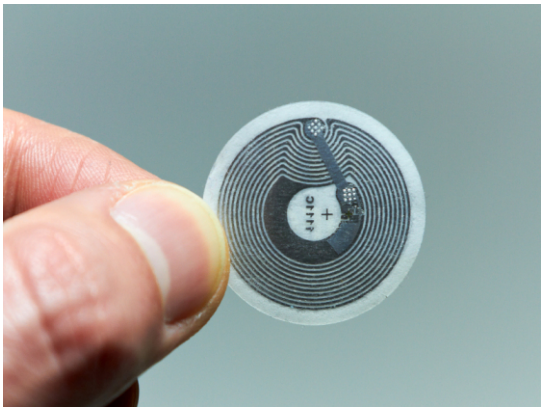


Figura 2.1: Un tag NFC.

Diversamente da RFID, non è possibile "vedere" contemporaneamente due tag NFC, poiché nel protocollo di comunicazione manca un meccanismo di controllo di accesso sulla frequenza. Sarebbe stato interessante poter leggere più tag per raffinare il posizionamento; constatata questa impossibilità, si avrà una corrispondenza biunivoca tra tag e posizione, in uno spazio di riferimento discreto.

La codifica NDEF NFC Data Exchange Format è un formato binario, utilizzato per memorizzare sui tag e veicolare informazioni di diverso tipo, insieme ad una codifica del tipo stesso. Si tratta di una specifica pubblicata da NFC Forum, è supportata dalla maggior parte dei tag in commercio ed è indipendente dal metodo usato per il trasporto.
Non è necessario attenersi a questa codifica per immagazzinare dati in un tag, tuttavia è preferibile, trattandosi di uno standard diffuso e ben supportato a livello di libreria.
NDEF definisce messaggi e record; un record contiene dati caratterizzati da un tipo, come ad esempio un media con MIME type, un URI, o un payload specificato dall'applicazione. Un messaggio è un contenitore, per uno o più record di lunghezza variabile.

Il supporto Android alla lettura dei tag NFC La piattaforma Android si fa completamente carico dell'implementazione dei protocolli per la lettura e la scrittura dei tag NFC, nascondendo al programmatore ogni aspetto della comunicazione. Questa scelta, da un lato abilitante, diventa limitante nel momento in cui il riconoscimento risulta molto meno efficace di quanto ci si aspetterebbe.

Quando l'antenna NFC è attiva il sistema effettua una ricerca continua. In caso di successo, si innesca un meccanismo che analizza il tag scoperto, esaminando il tipo dei dati contenuti, ed avvia un'applicazione che, mediante un *intent filter*, si era dichiarata interessata a tale tipo di dati. Android permette di specificare questo interesse in modo molto dettagliato, per poter individuare sempre la activity più appropriata per il tag letto. Una applicazione può richiedere di gestire tutti i tag, o solo quelli con formattazione NDEF, o solo gli NDEF con un certo MIME-type, o quelli che contengono un URI che segue un certo schema, come in questo esempio:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="http"
          android:host="developer.android.com"
```

```
            android:pathPrefix="/index.html" />
</intent-filter>
```

L'applicazione riceve un intent per ogni tag letto, e se ne può gestire il contenuto mediante una funzione *callback* che ha come argomento un oggetto rappresentante il tag stesso, a cui si può fare riferimento per ottenere (ed eventualmente modificare) tutte le informazioni relative.

Osservazioni L'esperienza di lettura e scrittura di questi tag NFC ha coinvolto due modelli di sticker compatibili NDEF, che sono stati preferiti ad altri con fattori di forma diversi per la loro versatilità e per il costo generalmente inferiore. Si sono condotte varie sessioni di lettura utilizzando diversi modelli di smart device, con comportamenti molto disomogenei, a parte per i tempi di lettura, che sono sempre nell'ordine dei decimi di secondo. Utilizzando "Samsung Galaxy S3" e "Samsung Galaxy S4" non si sono osservati particolari problemi; diversa la questione per gli altri modelli. Il "Nexus S", seppur impeccabile nelle singole letture, si è rivelato instabile, mostrando crash frequenti del sottosistema NFC, probabilmente a causa di un supporto immaturo. Tale problema in effetti è risolto in modelli successivi dello stesso produttore, che però presentano un problema molto più grosso: la lettura dei tag con "Nexus 5" e "Nexus 7 (2013)" è limitata ad un raggio talmente ristretto che per ottenere la rilevazione occorre posizionare il tag in corrispondenza di alcuni punti particolari rispetto al lettore. Se dal punto di vista della sicurezza può trattarsi di un miglioramento (NFC è utilizzato per transazioni finanziarie, che sono più sicure se il raggio è ristretto), per i nostri scopi si tratta di una perdita di fluidità, poiché la necessità di trovare la posizione adatta rallenta il procedimento di lettura in modo tale da renderlo spesso frustrante.

La sperimentazione ha coinvolto un numero esiguo di dispositivi e di tag, sicuramente insufficiente per fornire una valutazione della tecnologia, che tuttavia è stata ritenuta accettabile per il successivo utilizzo nel sistema di posizionamento approntato per *Il DISI incontra il MAMbo*.

2.2.3 Il magic carpet

Lavorando sull'idea di utilizzare un certo numero di tag NFC per definire un insieme di posizioni che i device possono assumere, e considerando il requisito di svolgere la dimostrazione su di un tavolo, si decide di disporre i tag in una griglia. Per decidere la distanza tra i tag si prendono in considerazione le dimensioni del dispositivo più ingombrante tra quelli a disposizione, aggiungendo un po' di spazio per permettere di maneggiare i device. Una volta verificato che tale distanza è tale da garantire l'assenza di conflitti in lettura, circostanza che potrebbe verificarsi se i tag adiacenti fossero troppo vicini, si stabilisce la dimensione della cella.

La griglia così predisposta è progettata come una versione in "formato tavolo" del sistema GPS, permettendo ad ogni dispositivo ivi posizionato di conoscere, in base ad una percezione sensoriale, la propria posizione. L'effettivo contenuto dei tag è dipendente dall'implementazione del middleware, e le possibilità sono varie. Si può, per esempio, scrivere su ogni tag le coordinate esatte che rappresentano la sua posizione sul globo terrestre, o su un sistema di riferimento ad hoc per il tappeto; oppure si può inserire un codice identificante una certa posizione che il middleware sarà in qualche modo in grado di risolvere, scelta più versatile perché non richiede di effettuare la riscrittura dei tag in caso di variazioni.

In ottemperanza al requisito di appetibiltà 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 2.2, 2.3 e 2.4, il tappeto ed alcuni particolari.

2.3 Le interazioni (non) 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 possiblità di

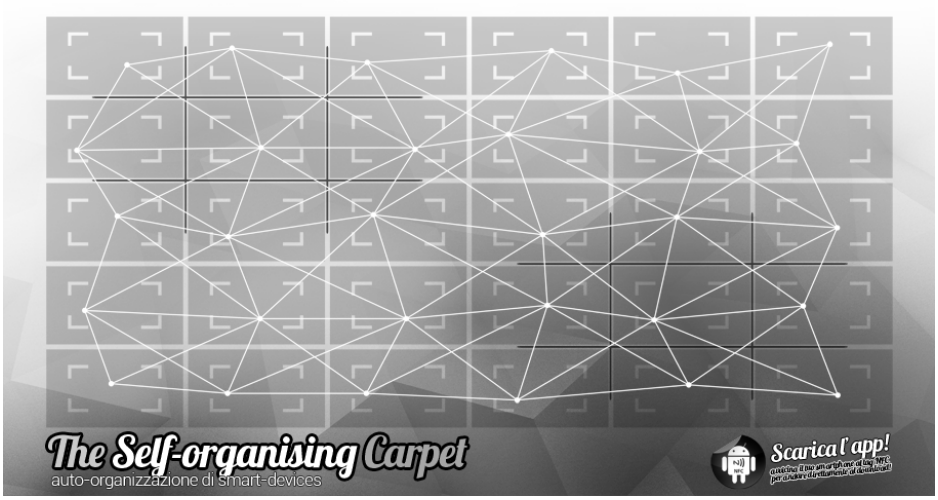


Figura 2.2: Il Magic Carpet.

generare comportamenti positivi supportati da un opportuno scambio di informazioni. Tale scambio richiede una 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 LAN, 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* [18, 3]. Una pietra miliare nella diffusione di queste tecnologie è l'integrazione, negli ultimi mesi, di un supporto alla costruzione di

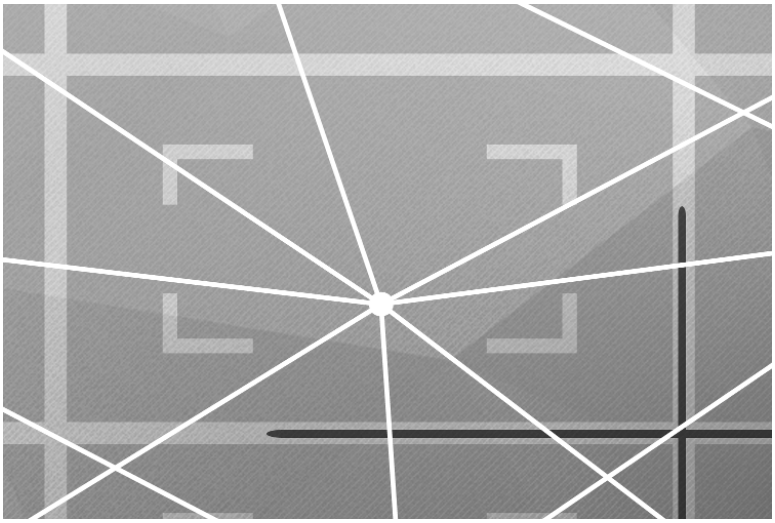


Figura 2.3: Magic carpet, particolare della griglia.

Figura 2.4: Un tag NFC posto sotto il logo consente ai device abilitati e con connessione ad internet di ottenere l'applicativo, e partecipare alla dimostrazione

reti *mesh* in uno dei sistemi operativi per smart device tra i più diffusi.

Quello appena descritto è il quadro di quell'ambito vastissimo e in pieno fermento che è 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.

Declinare questo requisito per il progetto in oggetto significherebbe imporre che le interazioni siano dirette ed indifferenti al medium per le comunicazioni, e che tutte le tecnologie di cui il device dispone siano utilizzate per ottenere la mas-

sima interoperabilità. Tenendo conto anche dell'orientamento del progetto verso lo spatial computing, e quindi verso algoritmi che inducono comportamenti globali per il sistema specificando solo interazioni locali, risulterebbe naturale identificare come vicinato di un nodo l'area entro la quale esso riesce a comunicare direttamente con i propri peer.

Tuttavia le condizioni particolari, derivanti dall'orientare il lavoro alla dimostrazione per "Il DISI incontra il MAMbo" conducono immediatamente ad una osservazione: dal momento che tutto si deve svolgere su di un tavolo, non è possibile definire in tal modo il vicinato di un nodo. Questo perché il raggio di azione di ognuna delle tecnologie a bordo dei comuni smart device, dal bluetooth alle comunicazioni cellulari, va ben oltre i confini del tavolo; i nodi risulterebbero tutti far parte dello stesso grande vicinato. Fa eccezione NFC, il cui raggio è invece troppo limitato.
Ne consegue che il limite alle comunicazioni deve essere simulato, ed occorre un meccanismo che identifichi il vicinato di un device entro un certo raggio, e restringa ad esso la visibilità.

Questo non impedirebbe di supportare comunque un ampio ventaglio di tecnologie per le comunicazioni, ma qui ci scontriamo con un limite della piattaforma Android: per apparenti ragioni di sicurezza non è possibile effettuare comunicazioni tra due dispositivi utilizzando il bluetooth o la tecnologia wi-fi direct senza che l'utente accetti esplicitamente l'instaurazione di una connessione con ognuno dei peer; essendo ciò totalmente incompatibile con quanto si vuole realizzare, si limita il supporto alle comunicazioni via IP, su wi-fi o rete cellulare.

2.4 Verso il middleware

Si riporta il punto di partenza di questo capitolo: "mostrare alcuni semplici pattern, come channel e partition, nello spazio a disposizione, cioè su un tavolo, e con tablet commerciali, insieme ad alcuni giochi; il tutto cercando di incarnare l'astrazione dello spatial computing, per mantenere l'aspetto scientifico."

La decisione successiva di utilizzare NFC come tecnologia per il posizionamento dà vita al "Magic Carpet". Le osservazioni sulle tecnologie di comunicazione a disposizione evidenziano la necessità di simulare un limite alla distanza raggiungibile.

Essendo evidente che il prodotto finale vedrà funzionare varie applicazioni costruite sulla medesima astrazione, è naturale pensare di costruire un middleware per colmare l'*abstraction gap* e su cui implementare la dimostrazione, con queste caratteristiche:

- Ogni nodo è 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 round 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.

2.4.1 L'estensione a BLE e GPS

Successivamente all'esperienza di "Il DISI incontra il MAMbo" il lavoro su Magic Carpet non si arresta, e va menzionato il contributo sperimentale dato da Andrea Fortibuoni con la sua tesi [10]. Tale lavoro esamina le tecnologie di posizionamento GPS e, in particolare, Bluetooth Low Energy (BLE), basata sui cosiddetti "Beacon".

Entrambe 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 un tavolo. Variando la dimensione dei vicinati si compie un cambio di scala tale per cui un errore di qualche metro risulta accettabile.

BLE è parte della specifica Bluetooth 4.0 (denominata anche Bluetooth Smart) e prevede una nuova modalità di comunicazione che, diversamente dalle precedenti, non necessita di connessione tra i partecipanti. La specifica permette ad un device di assumere il ruolo di *broadcaster*, e di inviare ciclicamente messaggi che

sono ricevuti da tutti i dispositivi vicini in ascolto, dove “vicino” indica un raggio di qualche decina di metri, dipendentemente dalla visibilità e dalla potenza del beacon. Utilizziamo questa modalità per diffondere un codice indicante la posizione, e tramite i device effettuiamo una scansione continua. Ognuno dei device potrà quindi ottenere la propria posizione, mediante un meccanismo, descritto in seguito, che sfrutta la misura dell’intensità dei segnali percepiti.

Nell’ambito di questa sperimentazione il middleware viene esteso per poter lavorare con tutte le tecnologie di posizionamento disponibili, privilegiando in ogni momento la più precisa. Quanto presentato di seguito ne è la versione finale.

Capitolo 3

Il middleware

3.1 L'interfaccia per lo sviluppatore

Si dà qui una descrizione del middleware, inizialmente esaminando il punto di vista dello sviluppatore di applicazioni, e le interfacce a lui fornite.

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. middleware risveglia periodicamente l'applicativo portandolo nella fase attiva, e fornendogli le informazioni ricevute dai vicini e quelle raccolte dai sensori. Elaborando tali infor-

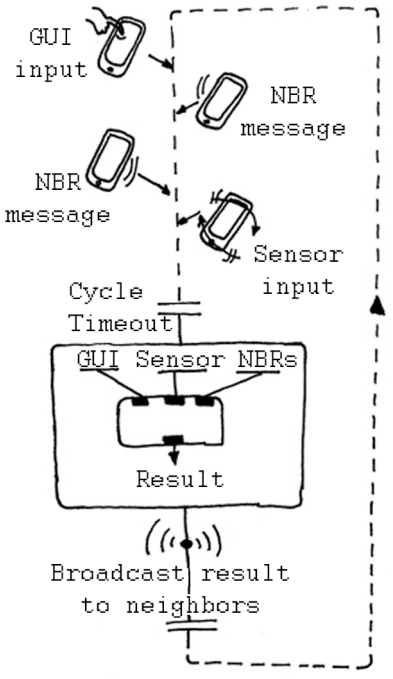


Figura 3.1: Il ciclo di attività scandito dal middleware.

mazioni l'applicativo produce un risultato che trasmetterà al middleware affinché lo diffonda tra i vicini, per poi tornare nella fase di attesa.
In figura 3.1 è schematizzato questo ciclo, con la linea tratteggiata ad indicare la fase di attesa e la linea continua per la fase attiva. Su questo schema si possono individuare i principali punti di intervento per lo sviluppo dell'applicazione:

- Il blocco "**eval**", ovvero la funzione che, dato l'input dell'utente, gli eventuali sensori e gli stati dei vicini, produce il nuovo **stato** del nodo.
- La raccolta degli **input** dall'interfaccia utente e dai sensori: anche se si tratta di stimoli che avvengono durante la fase di inattività, è possibile specificare una reazione agli eventi, per effettuare eventuali filtraggi o accumuli.
- L'**interfaccia grafica** per l'utente, sulla quale è possibile specificare sia vari tipi di input, come pulsanti, testi o forme più complicate, sia il modo in cui deve essere rappresentato lo stato dell'applicazione, utilizzando tutta l'espressività della piattaforma sottostante.

Questi punti di accesso si concretizzano in modi diversi; la tabella seguente traccia il quadro delle implementazioni necessarie per la realizzazione di una applicazione, esemplificando anche le convenzioni per i nomi utilizzate in quelle già realizzate.

stato del nodo `{nome app}Data`	L'oggetto che rappresenta lo stato del nodo, e quindi quello che verrà scambiato tra i nodi stessi.
input dai sensori `{nome app}Sensor`	Atto a veicolare le percezioni del sistema, questo oggetto sintetizza tutti i parametri ambientali disponibili.
eval `{nome app}App`	Estensione della classe astratta `BaseApp`, implementa il cuore del sistema, la logica che porta a produrre un nuovo stato per il nodo.
interfaccia grafica `{nome app}Activity`	Estensione della classe astratta `BaseActivity`, permette di specificare l'aspetto.

L'implementazione di queste quattro classi permette la specifica completa di

una applicazione. Mentre le prime due sono piuttosto triviali, la realizzazione della logica applicativa e della GUI con l'estensione di `BaseApp` e `BaseActivity` merita un esame più attento.

3.1.1 BaseApp

Prendiamo in esame BaseApp: come si può vedere dall'intestazione qui riportata, la classe fa uso dei generici di Java, il che la rende indipendente dalle specifiche applicazioni.

```
/**
 * @param <C>    Type of coordinates
 * @param <M>    Type of NBR table
 * @param <N>    Type of environmental input (sensors and gui)
 */
public abstract class BaseApp<C extends ICoordinates<?, ?>, M, N> {...}
```

L'uso dei generici per la classe rappresentante le coordinate è dovuto alla necessità di utilizzare coordinate di diverso tipo per le diverse applicazioni realizzate, come si vedrà in seguito.
`BaseApp` scandisce il passo della computazione.

```
while (true) {
    synchronized (this) {
        wait(SLEEP_TIME);
    }
    List<M> nbrs = emitter.retrieveStatuses().values();
    List<NodeInfo<C>> nodes = emitter.retrieveNodes().keySet();
    NodeInfo<C> me = emitter.retrieveMyNodeInfo();
    N env = retrieveEnvironmentalInput();
    M myNBR = generateLocalNBR(me, nbrs, nodes, env);
    this.myNBR = myNBR;
    notifyListeners();
    emitter.broadcastStatus(nbr);
}
```

Nel ciclo sopra riportato (in forma semplificata per migliorarne la leggibilità) si fa riferimento ad un membro chiamato `emitter`: si tratta di un oggetto di tipo `IEmitterManager<C,M>`, ed 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. Al termine della fase di sleep, `topologyManager` viene interpellato per raccogliere le informazioni necessarie alla valutazione del prossimo stato del nodo, un task che si compie in modo totalmente automatico e trasparente per chi sviluppa la app.
L'intervento del progettista si richiede per la specifica dei metodi astratti della classe.

```
public abstract N retrieveEnvironmentalInput();
```

Con questo metodo si completa la raccolta degli input, preparando un oggetto di tipo `{nome app}Sensor`. L'insieme dei sensori e degli input è fortemente dipendente dalla piattaforma, ed è possibile che, come accade in Android, non sia consentito accedere ad elementi dell'interfaccia o ai sensori al di fuori del contesto di una Activity. In tale scenario l'implementazione di questo metodo consisterà nel recupero delle informazioni necessarie dai registri in cui le activity le avranno poste in modo asincrono.

```
public abstract M generateLocalNBR(NodeInfo<C> me,
            List<M> nbrs, List<NodeInfo<C>> nodes, N env);
```

Tutte le informazioni raccolte sono fornite in ingresso a questo metodo, in cui è specificato l'algoritmo di spatial computing vero e proprio. Il prodotto della valutazione è un'istanza di `{nome app}Data`, che, mediante le ultime due istruzioni del loop preso in considerazione, verrà distribuita a tutti i vicini e a tutti i `listener` (in primis la Activity, per la visualizzazione del nuovo stato).

Per chiudere il focus su `BaseApp` si cita una *feature*, ancora in stato embrionale, pensata per ridurre i consumi energetici, un fattore chiave in ambito *mobile*. Il meccanismo consente l'abbassamento drastico della frequenza dei round computazionali (comunque configurabile) nel momento in cui lo stato del nodo e dei suoi vicini si stabilizza. Quando poi è necessario reagire prontamente a cambiamenti

nella topologia o nell'ambiente, la frequenza torna al livello alto. Dato che gran parte del consumo energetico è dovuto a schermo e comunicazioni, ridurre queste ultime è già un passo avanti per migliorare la durata della batteria del device.

3.1.2 BaseActivity

Quella che viene introdotta qui è la Activity costituisce il punto di accesso per il sistema, o più precisamente ne è la superclasse, 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. Comunque il middleware nasconde tutto ciò al progettista di applicazioni, per cui questi aspetti saranno considerati successivamente.

Ciò che ci interesa di questa classe è quello che *non* c'è: ovvero l'implementazione del metodo `initializeApplication`:

```
protected abstract GenericAppExecutor<?,?,?,?> initializeApplication(..);
```

`GenericAppExecutor` è una sorta di configuratore del sistema, ed occorre definirne uno specificando come tipi generici le classi appena descritte:

```
/**
 * @param <C>        The type of coordinates
 * @param <T>        {Nome App}App
 * @param <M>        {Nome App}Data
 * @param <N>        {Nome App}Sensors
 */
public abstract class GenericAppExecutor<C extends ICoordinates<?, ?>,
        T extends BaseApp<C, M, N>, M, N>
        implements IAppUpdateListener<M>, IPositionSensorListener {
```

Per utilizzare quest'ultima classe occorre sovrascrivere il metodo astratto `update(M state)`, che è il punto ideale per la specifica dell'aggiornamento della GUI: esso viene invocato alla fine di ogni ciclo computazionale e riceve come argomento lo stato aggiornato del nodo. Essendo pensato per facilitare le modi-

fiche dell'interfaccia grafica che seguono l'aggiornamento, questo metodo viene eseguito dal thread preposto a tale attività.
Inoltre `BaseActivity` si occupa di standardizzare l'accesso ad alcuni semplici input tattili, che vengono trasmessi direttamente alla app, sollevando l'implementatore dall'onere di ricevere e trasmettere tali input.

3.1.3 Dettagli implementativi

Per un setup funzionante ci sono altre questioni da prendere in considerazione, ma, in attesa di chiarire vari aspetti e tracciare un quadro più completo, si ribadisce che il middleware è progettato in modo da abilitare l'implementazione di un sistema di spatial computing con le quattro semplici implementazioni descritte nella sezione qui conclusa.

3.2 Architettura

La simulazione delle interazioni locali tra i nodi è implementata con una architettura client-server. I nodi si registrano ed inviano informazioni sulla propria posizione, il server le elabora e le confronta con quelle ottenute dagli altri nodi, e fornisce ad ognuno informazioni aggiornate sul proprio vicinato, che comprendono la lista dei vicini, le loro distanze e l'indirizzo a cui contattarli. La parte del sistema che, interagendo con il server, si occupa del mantenimento della lista aggiornata dei vicini, in modo che sia disponibile quando è necessario comunicare il proprio stato, prende il nome di `Topology`. Questo servizio offre al "core" del sistema un metodo per l'invio di aggiornamenti sulla posizione.

```
public void move(final IGeoTech geoTech); //IGeoTech aggrega
                        //informazioni relative alla posizione
```

Un altro sottosistema, detto `Emitter`, sfrutta il servizio `Topology` e si occupa delle comunicazioni con i vicini, sia in ingresso che in uscita. Il servizio offre tre metodi di interfaccia:

```
public NodeInfo<C> retrieveMyNodeInfo();
public Map<NodeInfo<C>, T> retrieveStatuses();
public void broadcastStatus(T t);
```

L'interazione dei servizi qui presentati con il core del sistema è schematizzata in figura 3.2. In figura 3.3 si guarda più da vicino al core stesso: oltre alle già note `BaseActivity` e `BaseApp` trova posto il servizio `TechManager`. Si tratta di un filtro che raccoglie tutti gli input sulla posizione raccolti dalle varie sorgenti disponibili e, tenendo conto anche della priorità, inoltra la variazione al servizio `Topology` solo quando si ha una informazione rilevante, e comunque senza mai superare una certa frequenza.

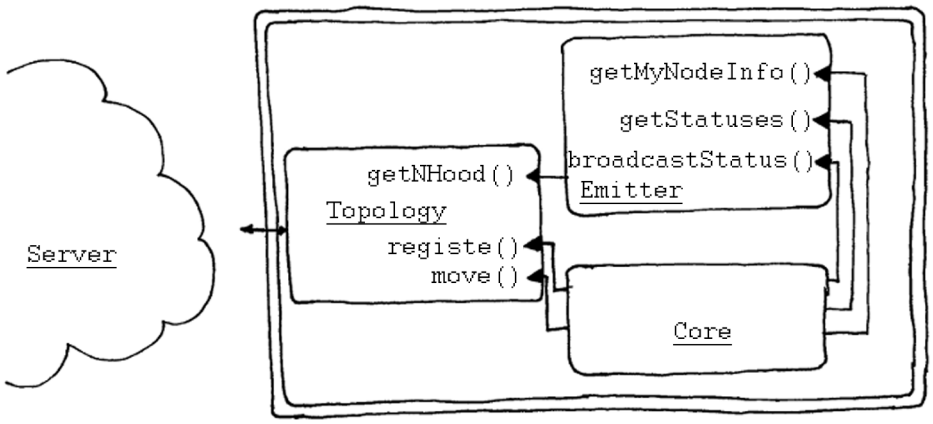


Figura 3.2: Principali interazioni del sistema.

3.2.1 Server, Topology e GeoTech

Si presentano qui alcuni aspetti delle entità che fanno parte del percorso che porta dalla percezione della posizione alla consapevolezza riguardo al proprio vicinato, schematizzato in figura 3.4.

La piattaforma Android consente di specificare una reazione al ricevimento di nuove informazioni da ciascuno dei sensori utilizzati, in modo simile. Per abilitare questo comportamento sono richieste varie azioni nei momenti chiave del *lifecycle* dell'Activity interessata, in questo caso `BaseActivity`. Per organizzare il

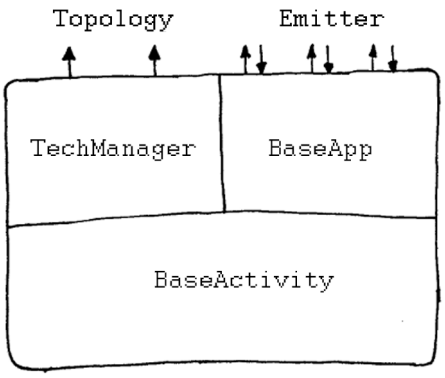


Figura 3.3: Componenti del "core".

codice in modo da migliorarne la leggibilità si crea una classe per ciascuna delle tecnologie interessate, all'interno della quale si implementano le operazioni necessarie. In questo modo i metodi che scandiscono il lifecycle di `BaseActivity` hanno questo aspetto:

```
protected void onResume() {
    super.onResume();
    nfcSupport.onResume(); //nfcSupport Istanza di TechNFC
    bleSupport.onResume(); //bleSupport Istanza di TechBLE
    gpsSupport.onResume(); //gpsSupport Istanza di TechGPS
}
```

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. Sia l'avvicinamento che l'allontanamento sono notificati a `GeoTechManager`, nel primo caso specificando il codice associato al tag.

Per i beacon BLE si fa uso di una libreria che ne semplifica la gestione, essendo il supporto diretto fornito da Android piuttosto carente. Attivata la scansione, viene fornita periodicamente una lista di oggetti `IBeacon`, una classe definita dalla libreria stessa. Tale lista è inoltrata a `GeoTechManager`.

Il supporto per GPS funziona in modo analogo, e ci porta a completare

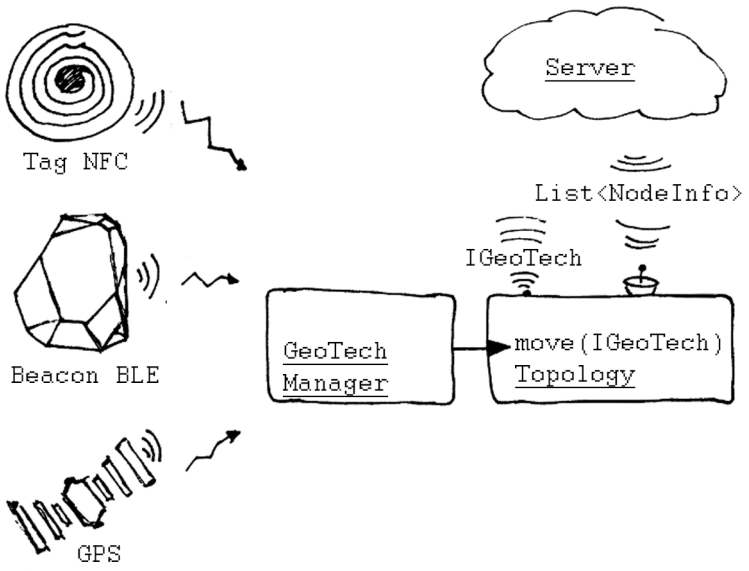


Figura 3.4: Le entità coinvolte nella determinazione del vicinato

l'interfaccia di `GeoTechManager`:

```
public interface IGeoTechManager {

    //NFC
    void newTag(int tag);
    void lostTag();

    //BLE
    void newBeaconList(IBeacon[] beacons);
    void lostBeacon();

    //GPS
    void newGPSCoords(ICoordinates coords);
    void lostGPSCoords();

    void addPositionSensorListener(IPositionSensorListener l);
    void removePositionListener(IPositionSensorListener l);
}
```

Tale classe riceve quindi tutte le informazioni disponibili sulla posizione. Il

suo ruolo di filtro consiste nel 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. Riprendendo l'esempio di poche righe fa, all'allontanamento del tag NFC (funzione `lostTag()`) segue immediatamente il recupero della migliore informazione tra quelle di classe inferiore, e la conseguente propagazione.

Il destinatario dell'informazione aggregata è il sottosistema Topology. Organizzato nelle due classi `NeighborhoodManager` e `GeoClient`, si occupa delle comunicazioni con il server, inviando gli aggiornamenti sulla posizione del nodo e stando in ascolto per quelle sul vicinato. L'informazione inviata al server è aggregata in un oggetto di tipo `IGeoTech` e serializzata.

```
public interface IGeoTech {
    int getTag();
    IBeacon[] getBeacons();
    Cartesian2dCoordinates getGPSCoords();
}
```

La risposta del server è una lista contenente tutte le informazioni d'interesse sui dispositivi facenti parte del vicinato. Si ricevono aggiornamenti ogni volta che nel 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. Tali associazioni sono specificate con una sintassi molto semplice in un file di configurazione che viene caricato all'avvio del sistema. Convenientemente, è possibile specifi-

care una sola volta le coordinate e l'orientamento del tappeto NFC per ottenere l'associazione della posizione corretta ad ognuno dei tag che lo compongono.

Quando da un client si riceve un oggetto IGeoTech, 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; in caso contrario si elabora una media delle coordinate di tutti i beacon, 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.

Vari accorgimenti sono stati adottati per assicurare che la frequenza con cui i device ricevono informazioni sul vicinato non sia troppo alta né troppo bassa, per evitare che i nodi vadano incontro a sovraccarichi o, in caso contrario, a *starvation*.

Il server è dotato di una GUI, scritta sia per Android che per le librerie `javax.swing`, che fornisce semplici informazioni sui nodi associati ed il loro posizionamento.

3.2.2 Servizio Emitter

IEmitterManager, il cui uso da parte di `BaseApp` è stato mostrato in precedenza, è implementato in due versioni, che si differenziano per il protocollo di comunicazione scelto. La logica tuttavia è la stessa.

```
public interface IEmitterManager<C extends ICoordinates<?, ?>, T> [...] {
    Node getId();
    NodeInfo<C> retrieveMyNodeInfo();
    Map<NodeInfo<C>, T> retrieveStatuses();
```

```
    void broadcastStatus(T t);
}
```

Il servizio mantiene una mappa che associa ogni nodo del vicinato al corrispondente ultimo stato noto. Le chiavi di tale mappa riflettono continuativamente la topologia, in perfetta sincronia con le informazioni disponibili al servizio preposto. I valori contenuti sono memorizzati ogni volta che si riceve un messaggio da un nodo vicino, ma solo se il nodo è già presente come chiave; la circostanza opposta si può verificare quando, entrando o uscendo un nodo dal vicinato di un altro, per un breve periodo i due non sono concordi sull'esistenza della relazione di vicinato.

3.2.3 Comunicazioni

Tutte le comunicazioni di rete sono supportate dal package `networkSupport`. Il suo scopo è fornire una interfaccia di più alto livello rispetto alle semplici socket; il supporto permette di implementare semplici protocolli di livello applicativo mediante la definizione di classi che specificano la struttura dei messaggi. Tutti i messaggi sono inglobati in una unità di trasporto dal formato standard per il quale è possibile specificare i vari possibili *opcode* in un tipo enumerato.

La serializzazione degli oggetti avviene nel formato JSON, tramite la libreria Gson[1]; per il trasporto delle informazioni così codificate sono disponibili implementazioni che sfruttano le socket, UDP o TCP.

3.2.4 La modalità a coordinate intere

In tutto il progetto si utilizzano i generici di Java per quanto riguarda il tipo delle coordinate. Questo perché per alcune delle applicazioni ludiche realizzate per *"Il DISI incontra il MAMbo"* non è importante conoscere la distanza tra i device, purché se ne possa conoscere la posizione sulla griglia. Per semplificare lo sviluppo di questo tipo di applicazioni quindi si può utilizzare, come tipo per le coordinate, `IntGridCoordinates`, i cui campi sono interi.

Quando si lavora in tale modalità viene preso in considerazione, come unico sensore per la posizione, NFC, ed è possibile lavorare solo sul tappeto.

3.3 Il simulatore

Oltre all'implementazione per Android ne esiste un'altra, inizialmente prodotta da Marco Alessi e Andrea Fortibuoni e successivamente estesa e ristrutturata per le nuove esigenze del progetto, interamente basata sulle librerie `javax.swing`, in grado quindi di essere eseguita su pc. Essendo il middleware strutturato in modo da poter utilizzare le stesse classi per implementare una App sia su Android che su JVM (fatta eccezione, ovviamente, per l'interfaccia grafica), questa versione si propone come un ottimo supporto allo sviluppo, perché permette di verificare in modo molto agile il comportamento del sistema mediante la simulazione in locale di sistemi composti da decine di nodi.

Inoltre, supportare l'esecuzione in modo indipendente dalla piattaforma Android significa velocizzare l'accesso a tutti i sistemi non Android che però possono accedere a Java, come Arduino e Raspberry Pi.

Capitolo 4

Le App

Si dà qui una descrizione delle applicazioni già realizzate per la piattaforma in collaborazione con Marco Alessi e Andrea Fortibuoni, ovvero quelle mostrate a *"Il DISI incontra il MAMbo"* e quella progettata nell'ambito del lavoro di tesi di Andrea Fortibuoni.

Queste applicazioni sono realizzate in java, senza supporto di altro genere a chiudere l'*abstraction gap* con l'approccio spatial computing, supporto che potrebbe venire da una libreria specifica o da un DSL. Ciò porta inevitabilmente a scrivere classi lunghe e poco leggibili che saranno qui riportate solo parzialmente. Farà eccezione la applicazione channel, di cui si mostrerà il codice applicativo proprio per apprezzare la differenza con la stessa specifica espressa dopo l'abbattimento del gap, discusso in seguito.

4.1 Channel

La prima applicazione realizzata è il channel, già presentata nel capitolo su spatial computing. È subito evidente la sproporzione tra la quantità di codice necessaria per descriverne il comportamento in java, e quella che occorre con il DSL Proto. Va detto che questa implementazione fa qualcosa in più: disponendo anche delle coordinate dei vicini, oltre che delle loro distanze, si fa comparire

sul display di ognuno dei device una freccia che punta al prossimo nodo lungo il channel.

La classe `ChannelSensor` definisce l'input ambientale disponibile, attraverso il quale il device riconosce la natura della regione in cui si trova.

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

In `ChannelData` si definisce la struttura dell'informazione condivisa nei vicinati. Se ne riportano i metodi pubblici:

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

Il codice che definisce il comportamento dell'applicazione si trova in `ChannelApp`, che estende `BaseApp`. Si riporta di seguito l'implementazione della funzione `generateLocalNBR`, che elabora il nuovo stato del nodo.

```
@Override
public ChannelData generateLocalNBR(
        NodeInfo<Cartesian2dCoordinates> me,
        List<ChannelData> nbrs,
        List<NodeInfo<Cartesian2dCoordinates>> nodes, ChannelSensor env) {

    //Computazione della distanza dalla regione src meno distante
    double dstSrc = INVALID;
    double distance;
    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;
            }
        }
    }

    //Computazione della distanza dalla regione dst meno distante
    Cartesian2dCoordinates nextNode = null;
    double dstDst = INVALID;
    if (env.isDst()) {
        dstDst = 0;
    } else {
        for (int i = 0; i < nbrs.size(); i++) {
            ChannelData h = nbrs.get(i);
            if (h == null || h.getDistSrc() < dstSrc)
                continue;
            distance = nodes.get(i).getDistance();
            if (h.getDistDst() != INVALID
                    && (dstDst == INVALID ||
                        h.getDistDst() + distance < dstDst)) {
                dstDst = h.getDistDst() + distance;
                nextNode = nodes.get(i).getPosition().getCoordinates();
            }
        }
    }

    // Calcolo e diffusione della distanza tra sorgente e destinazione.
    // In mancanza di gradcast si utilizza qui un gradiente modificato
    // con 'split horizon', ovvero realizzato inviando informazioni diverse
    // ai vicini a seconda della loro distanza dalle regioni src e dst.
    // Le due implementazioni si equivalgono nel comportamento, ma non ci
    // sono misteri su quale sia leggibile e quale no.

    double dstApart = INVALID;
    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() > inDstFromDstSVal))))) {
                minDstFromDst = h.getDistDst();
                minDstFromDstSVal = h.getDistApart();
            }
        }
    dstApart = minDstFromDstSVal;
    }

    // Gestione della soglia per il conteggio all'infinito in caso di
    // scomparsa di uno dei due estremi

    if (dstSrc > MAX_DST) {
        dstSrc = INVALID;
        dstApart = INVALID;
    }
    if (dstDst > MAX_DST) {
        dstDst = INVALID;
        dstApart = INVALID;
    }

    // Controllo dell'appartenenza o meno al canale, e conseguente
    // determinazione della direzione da indicare sul display con
    // la freccia

    boolean channel = dstApart != INVALID
```

```
            && dstSrc + dstDst <= dstApart + tolerance;
    int arrowDir = INVALID;
    Cartesian2dCoordinates myPos = null;
    if (me != null) {
        myPos = me.getPosition().getCoordinates();
    }
    if (channel && dstSrc > 0 && dstDst > 0 && myPos != null
            && nextNode != null) {
        int maxBlur = 0;
        int tolerance = 0;
        if (nextNode.getY() > (myPos.getY() +
                    2 * maxBlur + tolerance)) {
            if (nextNode.getX() > (myPos.getX() +
                        2 * maxBlur + tolerance)) {
                arrowDir = 45;
            } else if (myPos.getX() > (nextNode.getX() +
                        2 * maxBlur + tolerance)) {
                arrowDir = 315;
            } else {
                arrowDir = 0;
            }
        } else if (myPos.getY() > (nextNode.getY() +
                    2 * maxBlur + tolerance)) {
            if (nextNode.getX() > (myPos.getX() +
                        2 * maxBlur + tolerance)) {
                arrowDir = 135;
            } else if (myPos.getX() > (nextNode.getX() +
                        2 * maxBlur + tolerance)) {
                arrowDir = 225;
            } else {
                arrowDir = 180;
            }
        } else if (nextNode.getX() > (myPos.getX() +
                    2 * maxBlur + tolerance)) {
            arrowDir = 90;
        } else {
            arrowDir = 270;
```

```
        }
    }
        return new ChannelData(dstSrc, dstDst,
                        dstApart, channel, arrowDir);
}
```

Quanto visto fin qui definisce completamente la logica dell'applicazione. È il momento di descrivere l'interfaccia grafica. Per realizzarla si estende `BaseActivity` in `ChannelActivity` sovrascrivendo in questo modo `initializeApplication`:

```
@Override
protected GenericAppExecutor<?, ?, ?, ?> initializeApplication(
            GridProperties grid, boolean saveEnergy) {

    GenericAppExecutor<Cartesian2dCoordinates, ChannelApp,
                ChannelData, ChannelSensor> channel =
            new GenericAppExecutor<Cartesian2dCoordinates, ChannelApp,
                            ChannelData, ChannelSensor> (
                this, ChannelApp.class, ChannelData.class,
                false, grid, saveEnergy) {

        @Override
        public void update(ChannelData state) {
                    .
                    .
            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);
            }
```

```
            // Si imposta la rotazione per l'immagine della freccia
            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);
        }
    };
    addGestureSensorListener(channel.getApp());
    return channel;
}
```

Questo conclude l'implementazione della app channel. Una immagine della app in funzione sul magic carpet è mostrata in figura 4.1.

Figura 4.1: La app channel in una foto scattata durante "Il MAMbo incontra il DISI".

4.2 Partition

Partition, diversamente da channel, sfrutta l'idea di gradcast, riducendo sostanzialmente la complessità dell'implementazione alla diffusione del gradiente. La logica applicativa è già molto più compatta rispetto a quella dell'esempio precedente.

```
@Override
public PartitionData generateLocalNBR(
            NodeInfo<Cartesian2dCoordinates> me,
            List<PartitionData> nbrs,
            List<NodeInfo<Cartesian2dCoordinates>> nodes,
            PartitionSensor env) {

    // Il gradiente da src
    double dstSrc = INVALID;
    int tempColor = 0;
    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;

                // Determinata la src, se ne assume il colore
                tempColor = h.getColor();
            }
        }
    }
```

```
    // Conteggio all'infinito
    double maxDst = (nRows - 1) * v_step + (nCols - 1) * h_step + 4
            * maxBlur + tolerance;
        if (dstSrc > maxDst) {
        dstSrc = INVALID;
    }
        return new PartitionData(dstSrc, tempColor);
}
```

L'implementazione per Android consente la scelta tra tre colori per i device individuati come `src`. In figura 4.2 la app ed il gradimento mostrato dai visitatori più giovani.

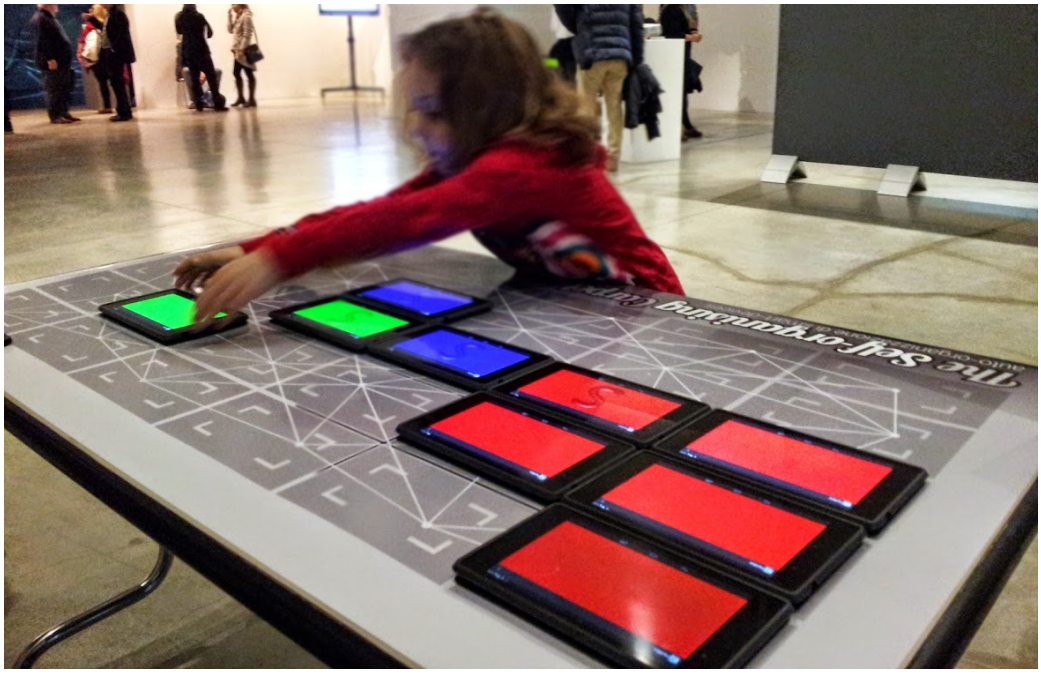


Figura 4.2: Una bimba impegnata con "il gioco dei colori".

4.3 Eigthpuzzle

8-puzzle è una versione ridotta di *15-puzzle* (figura 4.3), meglio noto come "gioco del quindici", e consiste nell'ordinare i numeri in una griglia effettuando

solo traslazioni che portino ad occupare la casella vuota con una delle tessere adiacenti.

La riduzione di scala è dovuta al numero di device disponibili, ma già con 8 tessere il gioco avrebbe tempi di gran lunga superiori al tempo di attenzione che ci si può aspettare venga dedicato ad una attrazione simile. Perciò si realizza per la demo anche una versione semplificata, sempre risolvibile in 6 mosse.

Figura 4.3: Il gioco del 15

Questa app, come quelle che seguono, 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 e le regole sono di fatto violate, la partita viene annullata.

Il raggiungimento della configurazione desiderata viene riconosciuto mostrando un messaggio di congratulazioni su tutti i device, come in figura 4.4.

4.4 Tic Tac Toe

Di gran lunga lo sport più praticato nelle scuole, meglio noto con il nome di *tris*, si presta molto bene all'occasione, per la possibilità di coinvolgere coppie di visitatori in sfide appassionanti. Se ne realizzano due versioni, di cui una a schema libero, che introduce un apprezzabile aspetto di novità in un gioco che notoriamente non può concludersi che in pareggio, salvo errori clamorosi.

L'implementazione è in grado di mantenere attive contemporaneamente due partite sullo stesso tavolo. In figura 4.5 una vittoria è stata appena attribuita al giocatore contrassegnato dal simbolo x. È evidenziata la serie vincente.

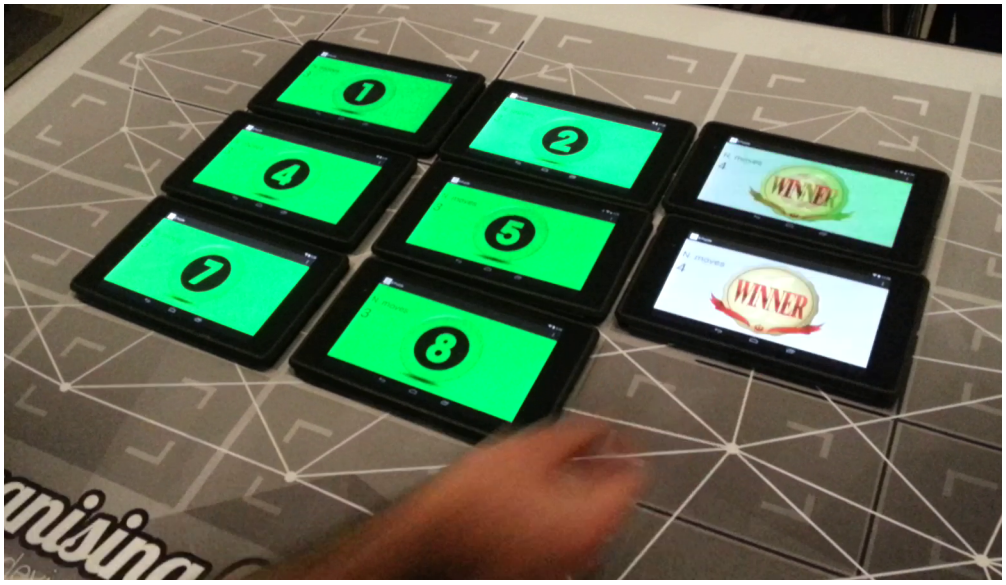


Figura 4.4: Eightpuzzle. L'ultimo tassello è appena stato posizionato correttamente e la configurazione è stata riconosciuta. In questo momento l'informazione si sta propagando tra i device.

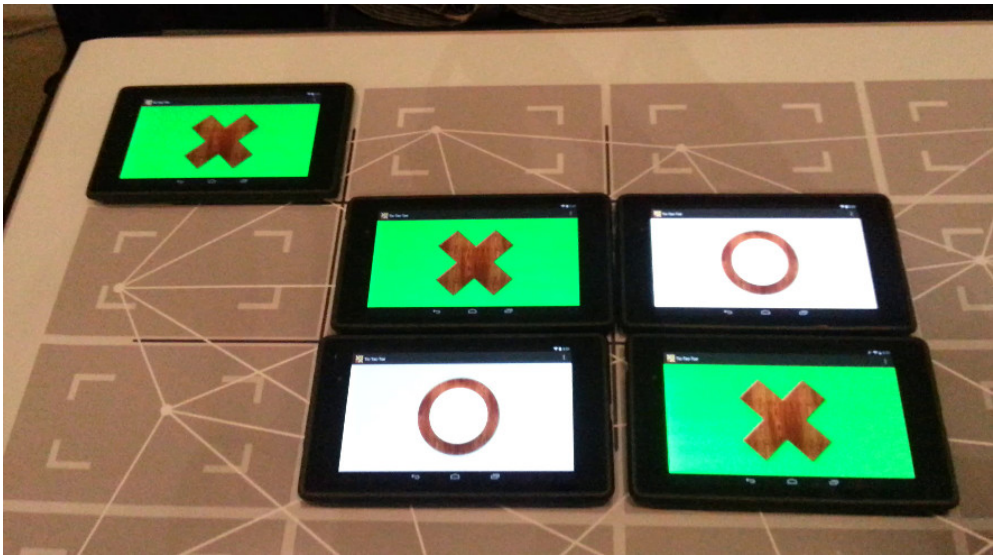


Figura 4.5: Tic tac toe. La *x* vince. Pessima partita per *o*.

4.5 Phuzzle

Phuzzle unisce *photo* e *phuzzle*: il nome è descrittivo di una applicazione in cui i dispositivi cooperano per unire i propri display realizzandone uno più grande, e rendere in tal modo una immagine. L'aggregazione avviene secondo vari criteri; quello prevalente è il tentativo di allargare il più possibile la superficie aggregata, eventualmente tollerando "buchi" o distorsioni dell'immagine.

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 una estensione a macchine di vario tipo.

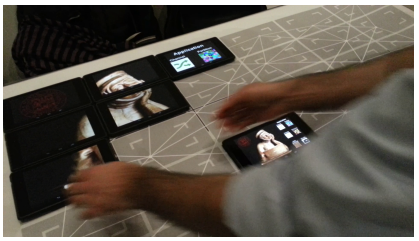
In figura 4.6 si utilizza phuzzle con una immagine di presentazione utilizzata in occasione della demo.

4.6 Flagfinder

Flagfinder si distingue dalle altre app sin qui viste per essere l'unica non orientata al 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. La app guida l'utente alla ricerca di un determinato punto di interesse costituito da un device "bandiera".

Nella dimostrazione realizzata, di cui figura 4.7 riporta alcune immagini, la bandiera è collocata sul 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 segnalare la strada verso l'aula in cui si trova il tappeto. Una volta che il device "cercatore" è posto al suo fianco, la bandiera si considera trovata.

(a) Vari schermi di dimensioni diverse coesistono sul carpet.



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

(c) Immagine completamente formata.

Figura 4.6: Varie aggregazioni di dispositivi per la resa dell'immagine.

(a) Esterno di via Sacchi, la app guida l'utente verso l'edificio.

(b) L'utente ha raggiunto il tappeto. Il finder, continua a puntare verso la bandiera.

Figura 4.7: Immagini tratte dalla demo Flag Finder.

Capitolo 5

DSL per campi computazionali

In questo capitolo si presenta l'elemento che completa il quadro del supporto fornito allo sviluppo di sistemi di spatial computing.

Si è mostrata sin qui una infrastruttura che fornisce ad un gruppo di device la possibilità di comunicare agevolmente con i propri vicini, e solo con essi, e si è considerata anche la tematica delle comunicazioni opportunistiche. Si è presentato il supporto alle varie tecnologie per la localizzazione disponibili, che consentono l'utilizzo su scale geografiche diverse.

Quello che "manca" si è manifestato durante lo sviluppo delle applicazioni: l'espressione in Java di alcuni concetti che ritornano frequentemente, come la selezione del minimo tra i valori percepiti, o la diffusione di un gradiente o di un gradcast, non sono assolutamente agevoli (emblematico l'esempio di channel, sezione 4.1). La distanza tra il modello del medium amorfo e la semplice programmazione orientata agli oggetti è ampia, e va colmata in qualche modo.

Si è visto alla sezione 1.2 come il linguaggio Proto consenta espressioni compatte per buona parte dei concetti che ci interessano, e così altri linguaggi o insiemi di pattern. Nel progettare un simile supporto per il middleware oggetto di questa tesi poniamo l'attenzione ad un approccio che, oltre a tentare di chiudere l'abstraction gap, compie un passo importante nella formalizzazione del modello di riferimento.

5.1 Field Calculus e Protelis

Per costruire una metodologia consistente per la coordinazione di sistemi distribuiti è necessario proporre qualcosa di più che una mera specifica. Serve l'abilità di prevedere, ad un dato livello di precisione, il comportamento di un certo modello.

Questa idea viene sviluppata in [21]. Si parte dall'osservazione che molti dei modelli formali, linguaggi di programmazione ed infrastrutture, creati per supportare lo spatial computing, condividono l'idea ricorrente che attraverso processi di diffusione, ricombinazione e composizione, l'informazione iniettata in qualche device possa produrre, a livello globale, dei *campi computazionali* in continua evoluzione. Field Calculus è un *core calculus* introdotto proprio con l'intento di identificare e formalizzare un insieme di ingredienti chiave per i linguaggi di programmazione che supportano la creazione di questi campi, e così spianare la strada all'analisi di proprietà chiave dei modelli di coordinazione: *soundness*, *expressiveness*, *self-stabilisation*, *topology independence*, e relazioni con la semantica spazio-temporale delle computazioni spaziali. Field Calculus ha molto in comune con Proto, essendo ispirato ad esso. Ma essendo, rispetto ad esso, "ristretto" ai costrutti fondamentali, è anche utilizzabile come una base per derivare, e quindi verificare e confrontare, molti degli altri approcci esistenti.

Field Calculus è un impianto teorico che necessita di un interprete, e di una architettura che gestisca gli aspetti di comunicazione, esecuzione ed interfacciamento con hardware e sistema operativo, mantenendo la portabilità tra gli ambienti simulati e i dispositivi connessi reali, e possibilmente assicurando che esattamente lo stesso codice possa essere utilizzato per l'esecuzione in ogni contesto.

Una implementazione viene presentata in [14], il sistema Protelis, introdotto allo scopo di superare le limitazioni pratiche che impediscono la diffusione dei linguaggi *field-based* già esistenti. Un nodo Protelis è costituito da un parser che traduce un programma in una rappresentazione valida di una semantica Field Calculus, la quale poi è eseguita ad intervalli regolari da un interprete, che si fa carico degli aspetti di interazione con il vicinato e con l'ambiente. Realizzato in

Java, Protelis adotta una sintassi Java-like, ed è puramente funzionale. A titolo di esempio si confrontano in figura 5.1 le sintassi Proto e Protelis della funzione gradiente.

```
(def distance-to (src)
    (rep d inf
        (mux src
            0
            (min-hood+ (+
                    (nbr d)
                    nbr-range)
))))
```

```
def distanceTo(src) {
    rep (d <- Infinity) {
        mux(src) {
            0
        } else {
            minHood (nbr(d) +
                    nbrRange)
}}}
```

Figura 5.1: Funzione gradiente secondo la sintassi Proto, a sinistra, e Protelis, a destra.

In aggiunta a quanto visto per Proto, si segnala la possibilità di importare metodi definiti in classi Java.

5.2 Core formale, interprete e middleware: la toolchain

Si sceglie di adottare Protelis come linguaggio per il sistema progettato in questa tesi, completandone così l'inquadramento, come descritto in figura 5.2. Alla base si troverà la Java virtual machine che esegue il middleware, il quale a sua volta lancia l'interprete Protelis. La specifica Protelis racchiude in sé l'elemento core della toolchain, la specifica formale che ne abilita la verificabilità.

La struttura del middleware si presta abbastanza bene all'innesto dell'interprete Protelis, senza necessità di intervenire su componenti infrastrutturali. Ciò che occorre fare è raccordare l'interprete con i servizi per la localizzazione ed l'interazione con il vicinato. L'implementazione disponibile per l'interprete richiede, per essere istanziata, un oggetto che rappresenti il vicinato, e per esso stabilisce una interfaccia, di nome `IEnvironment`. Solo i metodi principali di questa interfaccia sono qui riportati, non solo per brevità, ma anche perché effettivamente

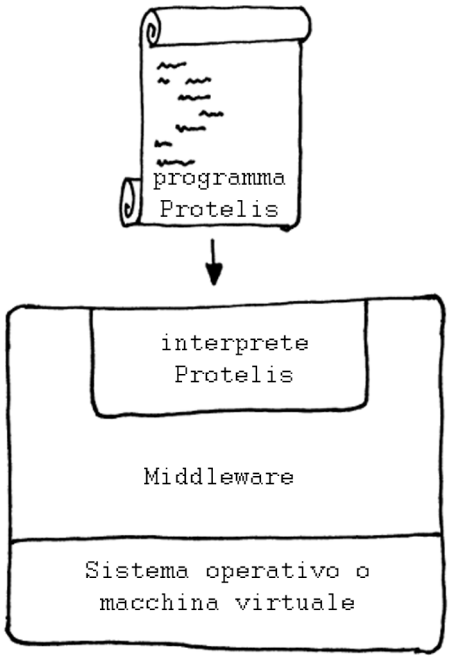


Figura 5.2: I layer della toolchain.

non necessari allo scopo. Ciò è dovuto al fatto che `IEnvironment` (e così tutto l'interprete Protelis) è parte di un sistema più ampio, denominato Alchemist [16], l'interprete usa solo parte di tale specifica.

```
/*
 * @param <T>
 *
 * /
public interface IEnvironment<T>
        extends Serializable, Iterable<INode<T>> {
    void addNode(INode<T> node, IPosition p);
    double getDistanceBetweenNodes(INode<T> n1, INode<T> n2);
    INeighborhood<T> getNeighborhood(INode<T> center);
    IPosition getPosition(INode<T> node);
    void removeNode(INode<T> node);
}
```

L'implementazione di questa interfaccia nel middleware richiede che queste informazioni possano essere fornite ad ogni valutazione del programma. Tali informazioni sono mantenute dal servizio `Topology`, ma sono anche rese disponibili per `BaseApp` ad ogni ciclo, e proprio nel momento in cui sono utili alla valutazione (vedi sezione 3.1). Perciò risulta opportuno implementare `IEnvironment` in una nuova estensione di `BaseApp`, che sarà una app al pari di tutte le altre realizzate, e prenderà il nome di `ProtelisApp`. La valutazione ciclica del nuovo stato sarà in questo caso interamente sostituita dalla valutazione del `ProtelisProgram`.

Diverso è il discorso per le comunicazioni con i vicini. L'interprete Protelis utilizza complessi meccanismi di caching per ottimizzare le trasmissioni, ed esistono casi in cui il messaggio che viene diffuso nel vicinato non sia, fisicamente, lo stesso per tutti. Il middleware tuttavia offre, ad alto livello, solo la possibilità di inviare, mediante il servizio `Emitter`, messaggi broadcast, e questo di fatto lo rende inutilizzabile allo scopo. Essendo l'esigenza di mandare messaggi diversi a diversi nodi limitata a `ProtelisApp`, si sceglie di non modificare i componenti esistenti, ma semplicemente di non utilizzarli per questo scopo, implementando invece un supporto specifico per l'invio delle strutture dati costruite dall'interprete.

Vale la pena di sottolineare uno degli aspetti più interessanti di questa toolchain, ovvero l'apertura alla mobilità del codice, supportata da Protelis e quindi potenzialmente da tutta la piattaforma. Sostanzialmente a livello di linguaggio è presente la possibilità di diffondere programmi nel medium, così come si diffonde qualunque altra informazione, di mettere tali programmi in esecuzione. Ma ciò che soprattutto è importante è il fatto che anche tale possibilità sia supportata al livello del core formale, rendendo anche questo aspetto verificabile e misurabile con strumenti matematici.

5.3 Nuove implementazioni

Al momento attuale purtroppo non è possibile vedere il sistema in azione sugli smart device, poiché l'interprete Protelis è implementato con costrutti introdotti nella versione 8 di Java, che non è supportata da Android. Cade a pennello in

questo caso la disponibiltà del simulatore, che permette, di fatto, di scrivere e verificare programmi indipendentemente dalla piattaforma specifica.

Si decide di dotare il sistema delle astrazioni per input e output più semplici presenti in Proto:

- I tre sensori booleani, richiamati con le istruzioni `(sense 1)`, `(sense 2)` e `(sense 3)`, realizzati mediante pulsanti.
- Gli attuatori led, utilizzati per mostrare semplici output, che sono attivati mediante la restituzione di una tupla di tre valori `double` come risultato dell'esecuzione del programma. I tre valori sono interpretati come le componenti cromatiche RGB in un range da 0 a 1.

Le implementazioni di gradiente e gradcast sono molto simili a quelle già viste. Di seguito la specifica.

```
def distanceTo(src) {
    rep (d <- Infinity) {
        mux(src) {
            0
        } else {
            minHood (nbr(d)+ nbrRange)
}}}
```

```
def gradcast(src, value) {
    let d = distanceTo(src);
    rep(val <- value){
        mux(src) {
            value
        } else {
            (minHood (nbr([d, val]))).get(0);
}}}
```

Channel La nuova specifica di channel introduce un elemento, la possibilità di identificare una regione come ostacolo, tale cioè da non poter far parte del canale. Il componente abilitante per questa funzione è un nuovo tipo di gradiente,

che a sua volta tiene conto degli ostacoli. Il costrutto `if` permette di separare le regioni.

```
def distanceToObs(src, obs) {
    if(obs) {
        Infinity
    } else {
        distanceTo(src)
}}
```

Di seguito il resto della specifica di channel in Protelis.

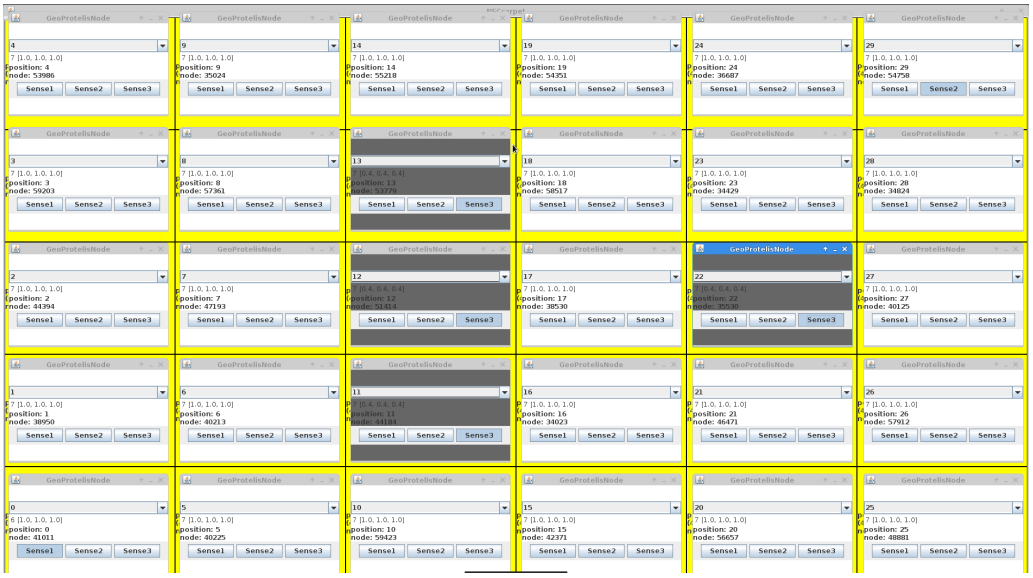
```
// booleanToDouble() mappa i field channel e obstacle su una
// rappresentazione codificata dal colore
def booleanToDouble(channel, obstacle){
    if (obstacle) {
        [0.4,0.4,0.4] // La regione ostacolo viene resa in grigio
    } else {
        if(channel) {
            [0.0,1.0,0.0] // il canale in verde
        } else {
            [1.0,1.0,1.0] // il resto dello spazio in bianco
}}}

// Anche la distanza tra due nodi si valuta evitando gli ostacoli
def distance (src, dst, obs) {
    gradcast(src, distanceToObs(dst, obs));
}

// Il canale, ancora definito a meno di una tolleranza,
def channel(src, dst, obs) {
    (dstSrc != Infinity &&
            (distanceToObs(src, obs) + distanceToObs(dst, obs) <
            0.00005 + distance(src, dst, obs)))
}

booleanToDouble(channel(sense1, sense2, sense3), sense3);
```

In figura 5.3 sono mostrati vari momenti dell'evoluzione di un channel

simulato con una griglia di 30 device virtuali, come succede sul magic carpet.

Partition Anche in questo caso, disponendo di gradcast, si puo dare di partition una definizione triviale. L'aspetto più complesso è l'associazione dei tre possibili input alla corrispondente combinazione additiva di colori, tramite la quale si determina la caratterizzazione di una partizione:

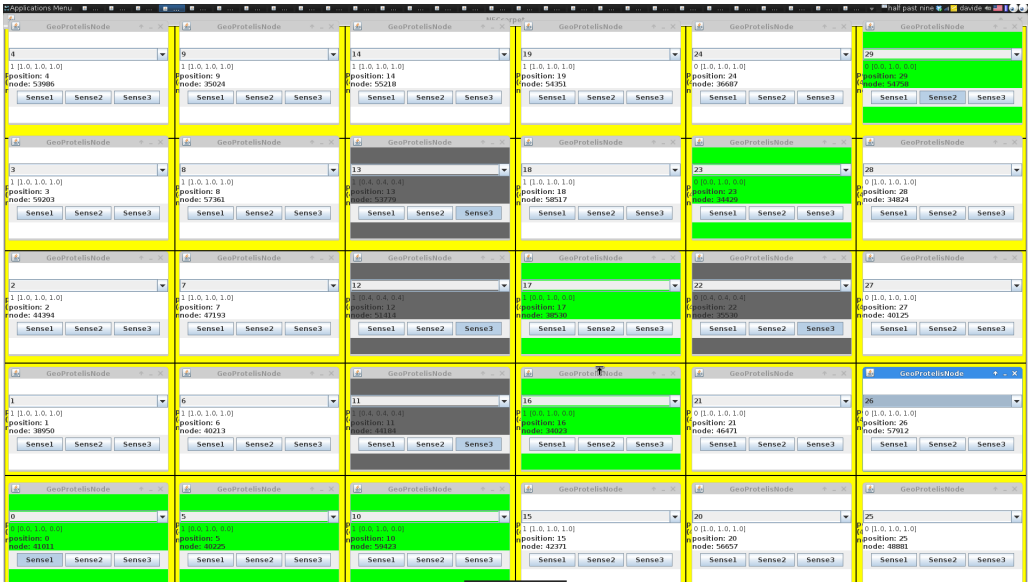
```
// Conversione da booleano a Double
def btd(val){
    if(val) {
        1.0
    } else {
        0.0
    }
}

def partition(red, green, blue) {
    gradcast (red || green || blue, [btd(red), btd(green), btd(blue)]);
}

partition(sense1, sense2, sense3);
```



(a) Sono state individuate le regioni `obs`, ben visibili, e le regioni `src` e `dst`, agli estremi opposti del display. Le informazioni si stanno propagando, il canale è ancora tutto da costruire

(b) A partire da `src`, man mano che si propaga il gradcast recante la distanza tra gli estremi, i device che si riconoscono parte della regione channel si colorano di verde.

Figura 5.3: Varie fasi dell'evoluzione di un channel.



(c) Il canale è completamente formato!

(d) Un malfunzionamento interrompe il canale, uno dei device che ne facevano parte passa, di fatto, nella regione `obs`.

Figura 5.3: Varie fasi dell'evoluzione di un channel.

(e) Un nuovo canale ripristina il collegamento tra gli estremi.

Figura 5.3: Varie fasi dell'evoluzione di un channel.

Conclusioni

In questo lavoro di tesi si è partiti dall'idea di produrre una installazione per la rassegna *"il DISI incontra il Mambo"*, svoltasi nel Marzo 2014. Si è quindi prodotto un sistema che permettesse di mostrare alcuni elementi dell'approccio computazionale denominato "spatial computing", miscelando elementi scientifici e di intrattenimento, e cercando di destare curiosità. Il sistema presentato consiste in un gruppo di smart device posti su un tavolo e messi in condizione, mediante un middleware, di comunicare esclusivamente con i propri vicini. Le applicazioni realizzate sfruttano tale supporto per mostrare alcuni pattern di aggregazione spaziale sui quali l'utente può intervenire, riconfigurando o spostando i device, ma non solo: al contenuto scientifico si accompagnano veri e propri giochi, che comunque non perdono del tutto di vista l'approccio spatial, essendo realizzati esclusivamente con interazioni locali.

La tecnologia NFC è stata esaminata e scelta tra le alternative per il particolare quanto essenziale compito di consentire al sistema di determinare la posizione dei device sul tavolo, in modo da fornire una topologia della rete con un errore nell'ordine dei centimetri. Ciò si ottiene mediante la realizzazione di una griglia di tag NFC, denominata, come l'intero sistema a partire da essa, magic carpet. Le informazioni contenute nei tag consentono al device che le legge di ottenere, tramite il middleware, la propria posizione. Si sono messi in luce vantaggi dell'uso di questa tecnologia, quali l'economicità dei tag, la velocità nella lettura e la precisione , e punti deboli come la non omogeneità nella distanza alla quale un tag viene letto, che varia molto a seconda del device lettore, introducendo errori potenzialmente ampi e difficoltà di utilizzo.

In un momento successivo il sistema è statp esteso per consentire la possibilità di utilizzare in modo integrato diverse tecnologie per il posizionamento, consentendo di spaziare dalla scala dei centimetri a quella dei kilometri. Oltre a NFC sono ora disponibili il supporto per la lettura di Beacon BLE e del segnale GPS.

Si è analizzato il campo dello spatial computing con particolare attenzione alla metafora del medium amorfo e delle strutture che in molti degli approcci presentati in letteratura caratterizzano ed operano su regioni dello spazio in base a determinate proprietà, i field computazionali. L'appetibilità di questo approccio deriva dalla sua formalizzazione in un core calculus che abilita una analisi rigorosa di proprietà di interesse nei programmi e nei pattern spaziali espressi con i linguaggi che condividono questo core. Tenendo in mente questo, e con l'obiettivo parallelo di superare l'abstraction gap presente tra spatial computing e programmazione orientata agli oggetti, quanto realizzato viene inquadrato nell'ambito di una toolchain che dal Field Calculus arriva alle applicazioni per Android. La specifica del comportamento di una applicazione si può quindi ora esprimere in modo formale e platform independent grazie all'integrazione di un interprete per il linguaggio Protelis, estensione di Field Calculus. Il middleware realizzato continua a farsi carico degli aspetti topologici e di comunicazione.

Nella realizzazione di questo sistema si sono affrontate numerose tematiche e sono tanti i possibili fronti che sono rimasti parzialmente, se non totalmente, inesplorati.

Un tema particolarmente caldo nella ricerca è quello delle comunicazioni opportunistiche, scartate in questo sistema anche a causa delle limitazioni in tal senso della piattaforma Android. Per via della grande attenziona alla sicurezza dell'utente di fatto si limita molto la libertà nell'uso delle varie interfacce a bordo dei device, richiedendo, nel migliore dei casi, invasive conferme ad ogni connessione, come avviene per bluetooth o per wi-fi direct. Tuttavia non sembra essere questa la direzione intrapresa dall'elettronica consumer: costituisce un esempio altisonante il grande rivale di Android, che supporta nativamente le reti *mesh* nelle ultime versioni del proprio sistema operativo. In generale la direzione sembra essere quella di una sempre maggiore apertura e interoperabiltà. Una soluzione nell'immediato

per la piattaforma Android potrebbe venire dal supporto migliorato al Bluetooth LE nella versione 5.0 della piattaforma. Le nuove modalità operative della specifica Smart Bluetooth sono ora pienamente disponibili, e sembrano offrire una ottima opportunità per le comunicazioni di prossimità.

BLE potrebbe essere una opportunità anche per la determinazione della distanza tra due device, semplicemente mediante la misura della potenza del segnale ricevuto. Senza dimenticare che approcci simili si sono solitamente dimostrati efficaci solo con l'impiego di pesanti infrastrutture, vale la pena di tenere d'occhio la nuova specifica e le sue varie modalità di funzionamento, che promettono la capacità di misurare la prossimità con errori di pochi metri o pochi centimetri, a seconda della potenza di trasmissione.

Un fronte che meriterebbe attenzione, viste le forti implicazioni sulla stabilità della piattaforma, è quello del controllo del mezzo di comunicazione condiviso, vale a dire lo spettro delle frequenze wi-fi. Il modello di comunicazione scelto, qualunque esso sia, arriva a dover fare i conti, crescendo il numero dei nodi, con la limitatezza della banda. Una piattaforma con aspirazioni di solidità non può esimersi dall'affrontare questo aspetto, guardando anche alle esperienze condotte in altri campi, come le reti di sensori.

Internet of Things non è più una sigla riservata agli addetti ai lavori, e milioni di utenti fanno esperienza quotidiana di cosa ciò significhi. Il lavoro qui presentato ha cercato un orientamento in questo campo in grande fermento, caratterizzato da una crescita che suscita curiosità e interesse.

Bibliografia

[1] A java library to convert json to java objects and vice-versa. https://code.google.com/p/google-gson/. Accessed: nov 2014.

[2] 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.

[3] 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.

[4] 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.

[5] Jacob Beal and Jonathan Bachrach. Programming manifolds. In André DeHon, Jean-Louis Giavitto, and Frédric Gruau, editors, *Computing Media and Languages for Space-Oriented Computation*, number 06361 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[6] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. *arXiv preprint arXiv:1202.5509*, 2012.

[7] 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.

[8] André DeHon, Jean-Louis Giavitto, and Fréderic Gruau. 06361 executive report–computing media languages for space-oriented computation. *InProceedings dehon_et_al: DSP*, page 1025, 2007.

[9] F. Dötsch, J. Denzinger, H. Kasinger, and B. Bauer. Decentralized real-time control of water distribution networks using self-organizing multi-agent systems. In *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, pages 223–232, Sept 2010.

[10] 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.

[11] K. McGuire and J. et al. Beal. Thinking in proto. http://proto.bbn.com/, 2011.

[12] 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.

[13] B. Ozdenizci, Kerem Ok, V. Coskun, and M.N. Aydin. Development of an indoor navigation system using nfc technology. In *Information and Computing (ICIC), 2011 Fourth International Conference on*, pages 11–14, April 2011.

[14] Danilo Pianini, Jacob Beal, and Mirko Viroli. Practical aggregate programming with PROTELIS. In *ACM Symposium on Applied Computing (SAC 2015)*, 2015. To appear.

[15] 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 on*, pages 667–674. IEEE, 2011.

[16] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.

[17] MIT Proto. Mit proto. software available at http://proto. bbn. com. *Retrieved December*, 2014.

[18] Ahmad Usman and Sajjad Haider Shami. Evolution of communication technologies for smart grid applications. *Renewable and Sustainable Energy Reviews*, 19:191–199, 2013.

[19] Mirko Viroli, Jacob Beal, and Matteo Casadei. Core operational semantics of proto. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1325–1332. ACM, 2011.

[20] Mirko Viroli, Jacob Beal, and Kyle Usbeck. Operational semantics of proto. *Science of Computer Programming*, 78(6):633–656, 2013.

[21] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing*, pages 114–128. Springer, 2013.

[22] Yu Wang, Lijuan Cao, Teresa A. Dahlberg, Fan Li, and Xinghua Shi. Self-organizing fault-tolerant topology control in large-scale three-dimensional wireless networks. *ACM Trans. Auton. Adapt. Syst.*, 4(3):19:1–19:21, July 2009.

[23] H.F. Wedde, S. Lehnhoff, C. Rehtanz, and O. Krause. Bottom-up self-organization of unpredictable demand and supply under decentralized power management. In *Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Second IEEE International Conference on*, pages 74–83, Oct 2008.

Ringraziamenti

Alla fine di un percorso lungo e non privo di difficoltà, desidero ringraziare tutti quelli che lo hanno condiviso con me, più o meno da vicino, incoraggiandomi, motivandomi e credendoci fino all'ultimo.

Un ringraziamento speciale al Prof. Mirko Viroli per il supporto impagabile, non solo dal punto di vista scientifico.

Ad Andrea e Marco, per la collaborazione entusiasta a questo progetto.

A Michele e a Cristian, per le grafiche di questo elaborato, e non solo.

Mi si consenta la ripetizione. A tutti voi che ci siete, e ci sarete, grazie.

You are the real MVP.