UNIVERSITÀ DEGLI STUDI DI PADOVA

—

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

—

TESI DI LAUREA IN INGEGNERIA INFORMATICA

PROGETTAZIONE DI UN SISTEMA DI NAVIGAZIONE INDOOR PER PALMARE TRAMITE RETI DI SENSORI WIRELESS

RELATORE: Ch.mo Prof. Luca Schenato

LAUREANDO: *Stefano Dazzo*

Padova, 24 ottobre 2011

UNIVERSITÀ DEGLI STUDI DI PADOVA

—

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

—

TESI DI LAUREA IN INGEGNERIA INFORMATICA

DESIGNING AN INDOOR NAVIGATION SYSTEM FOR HANDHELD USING WIRELESS SENSOR NETWORKS

RELATORE: Ch.mo Prof. Luca Schenato

LAUREANDO: *Stefano Dazzo*

Padova, 24 ottobre 2011

Alla mia famiglia

Indice

Elenco delle tabelle

Elenco delle figure

Sommario

Nella tesi verrà affrontata la progettazione di un sistema di navigazione indoor per un palmare attraverso l'utilizzo di una rete di sensori wireless, sfruttando solamente le informazioni derivanti dalle comunicazioni radio fra i nodi della rete.

Il progetto si compone di due parti principali, una legata alla realizzazione del collegamento tra un nodo della rete con un palmare, che come vedremo nel seguito corrisponderà ad uno smartphone Android di ultima generazione. Il collegamento diretto purtroppo non è fattibile a causa dei limiti imposti dall'hardware utilizzato, sia dal nodo della rete sia dal palmare. Verrà descritto come strumento di collegamento tra i due dispositivi una scheda bluetooth, che permette di inoltrare i messaggi dalla rete al palmare e viceversa attraverso il protocollo bluetooth. Tale scheda verrà connessa ad un nodo della rete grazie alla presenza di una porta seriale su entrambi i dispositivi.

La seconda parte della tesi presenta una breve descrizione di alcuni progetti proposti in letteratura per risolvere il problema della localizzazione, ed in fine verrà descritto un semplice ed intuitivo algoritmo di localizzazione per validare la fattibilità di una sistema così costruito.

Il risultato finale è un prototipo che per dimensioni e caratteristiche potrebbe essere considerato un prodotto commercializzabile, anche se l'accuratezza dell'algoritmo di localizzazione proposto non è sempre elevata e richiede ulteriori studi a riguardo.

Introduzione

Le reti di sensori (Wireless Sensor Network, WSN) sono composte da dispositivi senza fili per il monitoraggio dei parametri ambientali in luoghi naturali ma anche industriali. In realtà questo tipo di dispositivi non sono semplicemente sensori dotati di un'antenna per la comunicazioni senza fili ma sono composti da una hardware e software dedicato capace di integrarsi nell'ambiente in cui sono dislocati. L'idea nasce alla fine dello scorso secolo dai laboratori della NASA, con il proposito di creare un insieme di piccolissimi elementi in grado di configurarsi e comunicare tra di loro creando una rete di comunicazione. Il desiderio era quello di creare una *smart dust*, letteralmente polvere intelligente, un rete di piccoli elementi estremamente pervasiva e di ridotti costi, in figura 1 riportiamo alcuni esempi di nodi che compongono le WSN.

Le WSN, date le loro caratteristiche, stanno sempre più prendendo piede, non solo nell'ambito accademico per il gran numero di applicazioni possibili e per il coinvolgimento di diverse discipline scientifiche, ma anche in ambito industriale e commerciale per le svariate possibilità di impiego, strumenti di analisi molto robusti, i costi ridotti di questi dispositivi e la facilità di controllo anche in zone inaccessibili direttamente. Le applicazioni si possono distinguere in tre diverse categorie:

- monitoraggio: una rete costruita per tenere traccia con continuità di grandezze fisiche relative ad una certa area geografica in ambienti aperti oppure in ambienti chiusi come gli edifici o fabbriche, alcuni esempi possono essere il monitoraggio dell'acqua, dell'aria, temperatura, ma anche di possibili incendi in zone boschive;

- localizzazione: applicazioni di questo tipo sfruttano le proprietà della rete di sensori per stabilire la posizione di un nodo mobile all'interno della rete. Queste tecniche vengono studiate sia in ambito militare che in quello civile,

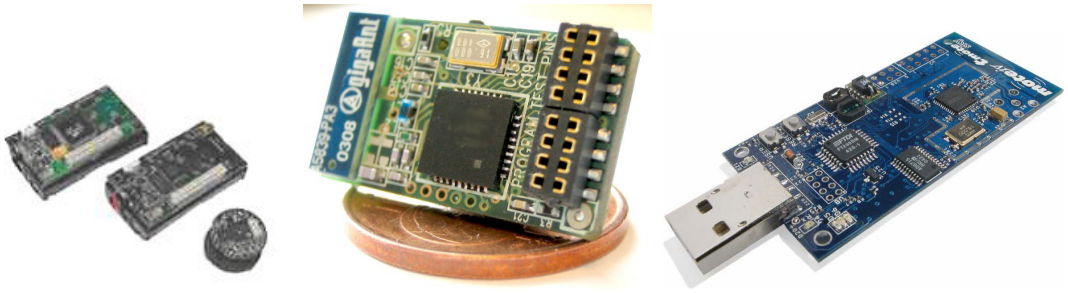


Figura 1: Alcuni tipi di nodi che formano le WSN.

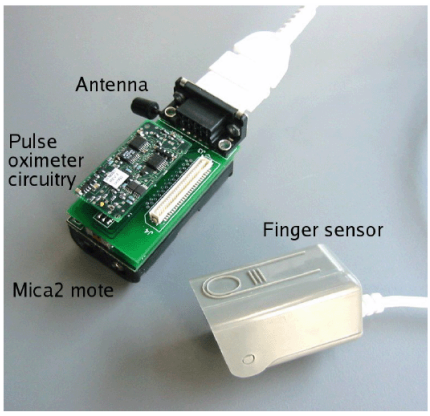


Figura 2: Esempio di nodo per applicazioni elettromedicali.

per esempio il controllo di zone a rischio di un cantiere, la localizzazione di vigili del fuoco all'interno di un edificio in fiamme oppure il monitoraggio dei degenti di un ospedale, consentendo al medico un costante controllo del paziente lasciandolo libero di muoversi all'interno della struttura (in figura 2 si riporta il un nodo per applicazioni elettromedicali);

- controllo: quest'ultimo gruppo di applicazioni è finalizzato al riconoscimento di determinati eventi da parte dei nodi della rete ed il successivo adattamento delle strutture a questi eventi, basti pensare al controllo in ambito agricolo della temperatura e dell'umidità di una serra e l'adattamento di questi valori aumentando o diminuendo la ventilazione, oppure il controllo della strumentazione di una fabbrica o nel campo della videosorveglianza.

Il problema che andremo a trattare nel seguito di questa tesi riguarda la localizzazione che si distingue dal tracking. Quest'ultimo infatti è volto a stabilire una traiettoria, cioè un percorso che l'agente mobile, nel nostro caso un nodo mobile della rete, all'interno di un certo ambiente deve seguire per raggiungere

una determinata destinazione partendo da un punto conosciuto, come ad esempio i sistemi GPS. Mentre la localizzazione si occupa di stabilire la posizione attuale dell'agente mobile all'interno dell'ambiente.

Andremo ad affrontare il problema con l'obbiettivo di costruire un sistema di localizzazione per un palmare. Ovvero andremo a costruire un'architettura di comunicazione tra il nodo mobile ed il palmare, che consenta a quest'ultimo di ricevere i dati dai nodi fissi, altrimenti detti nodi àncora cioè nodi di cui è nota la posizione e che rimango fissi nell'ambiente, e di calcolare e visualizzare la posizione corretta del nodo mobile. Nella fattispecie andremo a descrive il collegamento tra un nodo di una WSN del tipo Tmote-Sky con uno smartphone Android. Il collegamento non può essere effettuato direttamente a causa dei limiti imposti dall'hardware (purtroppo la porta USB presente sullo smartphone ne consente l'utilizzo solo come slave), si è dunque pensato di frapporre tra i due dispositivi una scheda bluetooth (connessa alla porta seriale del Tmote-Sky). Realizzando uno schema di connessione come mostrato nella figura 3.

Ad ora non è ancora stato presentato un sistema commerciabile per la localizzazione in ambienti indoor. Il contributo di questo progetto è quello di creare un prototipo che per dimensioni e caratteristiche (costi e capacità di calcolo) sia conveniente da produrre su larga scala. In ambito accademico il problema della localizzazione viene studiato già da alcuni anni e sono state proposte diverse soluzioni, alcune che sfruttano le proprietà degli ultrasuoni, come il sistema presentato nell'articolo [16], altri che utilizzano un sitema interferometrico, si veda l'articolo [22]. Algoritmi come Motetrack [20] o Radar [18] sfruttano le caratteristiche del campo elettromagnetico. Di particolare interesse è un algoritmo recentemente sviluppato presso l'Università di Padova presentato nell'articolo [23] che utilizza tecniche di correzione dell'errore. L'algoritmo che andremo a utilizzare per risolvere il problema della localizzazione è molto intuitivo e di semplice implementazione, restituendo dei risultati accettabili in termini di errore assoluto tra la posizione reale e quella calcolata. La tecnica utilizzata effettua la media pesata delle coordinate dei nodi àncora rispetto alla distanza di quest'ultimi dal nodo mobile. Distanza calcolata in base alla potenza del segnale radio ricevuto. Questa soluzione risente tuttavia dei disturbi presenti nell'ambiente quali riflessioni, diffrazioni e scattering, e quindi necessita di un'ulteriore fase di studio.

Nei capitoli 1 e 2 andremo a descrivere rispettivamente gli strumenti software

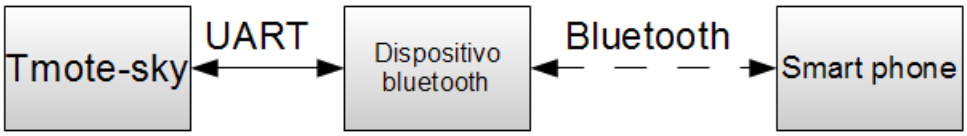


Figura 3: Schema di connessione tra Tmote-sky e un dispositivo smartphone attraverso comunicazione bluetooth.

e hardware utilizzati per la realizzazione del sistema, ponendo particolare attenzione alle caratteristiche del sistema Android che presenta delle caratteristiche interessanti nella sua architettura. Nel capito 3 verrà descritto il collegamento seriale tra il Tmote e la scheda bluetooth per proseguire con la descrizione del programma di comunicazione. In fine nel capitolo 4 dopo una breve descrizione di alcune tecniche di localizzazione note, presenteremo una soluzione implementata e i risultati ottenuti.

Capitolo 1

Ambiente Software

1.1 Introduzione a Google Android

Android[7] è una piattaforma per dispositivi mobili che include un sistema operativo, un middleware e alcune applicazioni di base. L'Android SDK, ovvero il kit per lo sviluppo del software, offre gli strumenti e le API necessarie per iniziare a sviluppare le applicazioni utilizzando il linguaggio Java. Basato sul kernel Linux ad oggi è arrivato alla versione 2.3 Gingerbread, del 21 ottobre 2010 basata sulla versione del kernel Linux 2.6.35.

Le sue caratteristiche principali sono:

- Framework per le applicazioni che consente il riutilizzo e la sostituzione delle componenti,
- Dalvik, la virtual machine ottimizzata per dispositivi mobili,
- Browser integrato, basato sul motore del browser open source WebKit,
- Grafiche ottimizzate alimentate da una library grafica 2D personalizzata e grafiche 3D basate sulla specificazione OpenGL ES 1.0 (hardware acceleration opzionale)
- SQLite per l'immagazzinamento dei dati
- Media support per comuni formati audio, video, e immagini (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- Telefonia GSM (dipendente dall'hardware)

- Bluetooth, EDGE, 3G, e Wi-Fi (dipendente dall'hardware)
- Fotocamera, GPS, bussola, ed accelerometro (dipendente dall'hardware)
- Ricco ambiente di sviluppo, che include un emulatore, strumenti per debugging, profiling della memoria e delle prestazioni e un plugin per l'IDE Eclipse.

Google Android fu inizialmente sviluppato da Android Inc., startup acquisita nel 2005 da Google. I cofondatori di Android Inc., Andy Rubin (a sua volta cofondatore di Danger), Rich Miner (cofondatore di Wildfire. Communications, Inc.), Nick Sears (vice presidente di Tmobile), e Chris White (principale autore dell'interfaccia grafica di WebTV). iniziarono a lavorare per Google e svilupparono una piattaforma basata sul Kernel Linux. Il 5 novembre 2007 l'Open Handset Alliance presentò Android, costruito sulla versione 2.6 del Kernel Linux. La piattaforma è basata sul database SQLite, la libreria dedicata SGL per la grafica bidimensionale e supporta lo standard OpenGL ES 2.0 per la grafica tridimensionale. Le applicazioni vengono eseguite tramite la Dalvik virtual machine, una Java virtual machine adattata per l'uso su dispositivi mobili. Android è fornito di una serie di applicazioni preinstallate: un browser, basato su WebKit, una rubrica e un calendario.

1.1.1 Android SDK

Google già dalle prime versioni di Android distribuisce il cosiddetto Android SDK[2] (giunto ormai alla versione 2.2), un insieme di tool di sviluppo multipiattaforma, compatibile con Windows, Mac e Linux, in grado di mettere a disposizione degli sviluppatori tutti gli strumenti utili per conoscere e usare Android. L'SDK fondamentalmente comprende l'emulatore che consente agli sviluppatori di provare le applicazioni come se le stessero visualizzando ed utilizzando in un terminale Android vero e proprio, eccezion fatta per particolari funzioni quali l'accelerometro che necessita dell'hardware. L'SDK comprende, inoltre, apposite sottosezioni ricche di esempi ed implementazioni, di oggetti e temi standard per Android, dai quali gli sviluppatori possono trarre spunti utili o addirittura riutilizzare il codice senza ricopiarlo all'interno delle loro applicazioni, ma come verrà spiegato in seguito, sarà sufficiente richiamare quel segmento di codice da un'app

all'altra così da risparmiare spazio e rendere più efficienti le applicazioni. Elemento di forza dell'SDK è che non dipende da alcun tipo di ide, si è liberi di utilizzare l'ide che più si preferisce, Google consiglia di utilizzare Eclipse, per il quale esiste un apposito plugin da installare separatamente, che consente di integrare tutta una serie di funzionalità aggiuntive molto comode per sviluppare in particolare progetti Android. Tra gli altri software che si possono utilizzare come ide si può citare NetBeans, forse il più diffuso ide per java, o in alternativa qualsiasi altro ide che supporti java e che consenta la compilazione e l'archiviazione (in collaborazione con l'SDK) del file progetto all'interno dell'archivio *.apk*, l'estensione di tutte le app Android. Tramite l'SDK e il sito: `http://developer.Android.com`, tutti gli sviluppatori esperti di java che vogliono imparare a programmare per Android trovano un'ampia documentazione che spiega in modo approfondito come usufruire di tutte le librerie di sistema, e anche come funziona il sistema operativo stesso al fine di ottimizzare le app e generare prodotti che seguano criteri di efficienza, massime performance e stabilità operativa, per favorire il riutilizzo e la portabilità tra differenti versioni e terminali. Recentemente GoogleLab ha reso disponibile AppInventor per Android, un ambiente di sviluppo online che consente di sviluppare applicazioni Android in modo rapido, semplice e intuitivo grazie alla programmazione a blocchi di codice. Per utilizzare AppInventor, non è necessario essere uno sviluppatore. AppInventor non richiede alcuna conoscenza di programmazione, questo perché, invece di scrivere codice, si progetta tramite un'interfaccia visuale il modo in cui devono apparire gli oggetti e gli stili dell'app, e tramite l'editor di blocchi di codice si definisce il comportamento dell'applicazione stessa.

1.1.2 L'architettura di Android

Il diagramma seguente (fig. 1.1) mostra le componenti principali del sistema operativo di Android[7]. Le sezioni che lo compongono sono descritte più dettagliatamente di seguito.

Applicazioni

Android funziona con un set di applicazioni di base, che comprende un email client, un programma SMS, un calendario, le mappe, il browser, i contatti e altro. Tutte le applicazioni sono scritte in linguaggio Java.

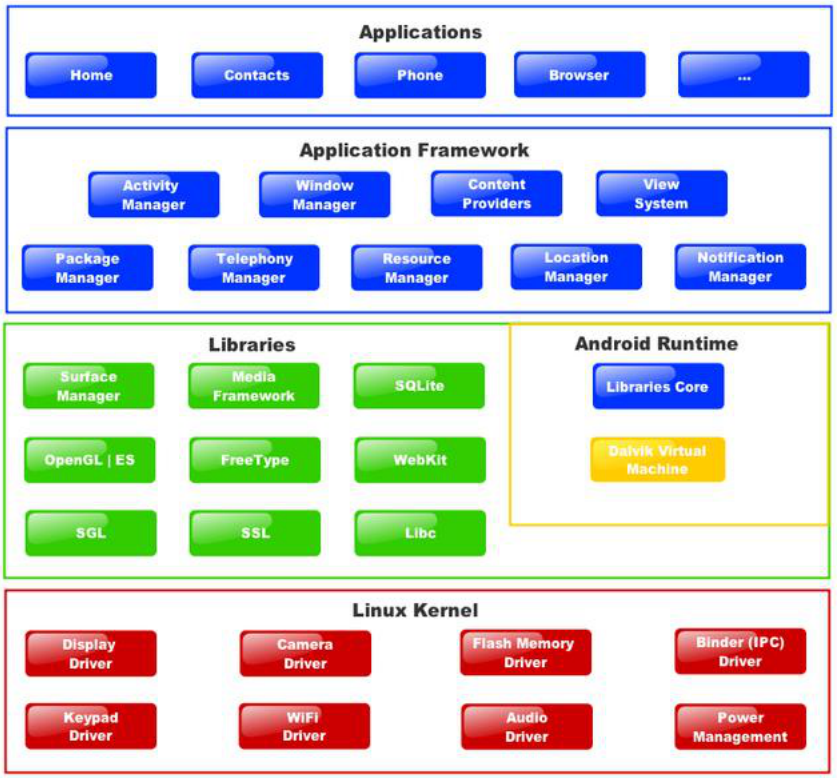


Figura 1.1: Dettaglio dell'architettura di un sistema Android[7].

Framework per applicazioni

Gli sviluppatori hanno pieno accesso alle stesse framework API usate dalle applicazioni di base. L'architettura delle applicazioni è progettata per semplificare il riutilizzo dei componenti; ogni applicazione può rendere pubbliche le sue capacità e tutte le altre applicazioni possono quindi farne uso (sono soggette ai limiti imposti dalla sicurezza del framework). Questo stesso meccanismo consente all'utente di sostituire le componenti standard con versioni personalizzate. Alla base di ogni applicazione si trova un set di servizi e sistemi, tra cui:

- Un gruppo ricco ed estensibile di viste che possono essere usate per costruire un'applicazione; contiene liste, caselle di testo, pulsanti, e addirittura un browser web integrato.
- Dei Content Providers che permettono alle applicazioni di accedere a dati da altre applicazioni (come i Contatti), o di condividere i propri dati .
- Un Manager delle risorse, che offre l'accesso a risorse non-code come strings localizzate, grafica, files di layout.

- Un Manager delle notifiche, che permette a tutte le applicazioni di mostrare avvisi personalizzati nella status bar.

- Un Manager delle attività, che gestisce il ciclo di vita delle applicazioni e fornisce un backstack di navigazione comune.

Librerie

Android comprende un set di librerie C/C++ usate da varie componenti del sistema di Android. Questi elementi sono presentati allo sviluppatore attraverso il framework per applicazioni di Android. Sono elencate di seguito alcune delle librerie principali:

- System C library - un'implementazione BSD-derived della libreria standard C system (libc), disegnata per dispositivi basati su Linux.

- Media Libraries - basate sull'OpenCORE di PacketVideo; le librerie supportano la riproduzione e la registrazione di molti popolari formati audio e video, compresi file di immagini, inclusi MPEG4, H.264, MP3, AAC, AMR, JPG, e PNG.

- Surface Manager - gestisce l'accesso al display subsystem e compone layer grafici 2D e 3D da applicazioni multiple.

- LibWebCore - un motore di browser moderno che fa funzionare sia il browser Android sia la visualizzazione web implementata.

- SGL - il motore grafico 2D sottostante.

- 3D libraries - un'implementazione basata su APIs OpenGL ES 1.0; le librerie usano sia accelerazione hardware 3D (quando disponibile) sia quella inclusa, un rasterizer software 3D altamente ottimizzato.

- FreeType - rendering di bitmap e vector font.

- SQLite - un motore di database relazionale potente e leggero disponibile per tutte le applicazioni.

Runtime

Android comprende un set di librerie centrali che forniscono la maggior parte

Figura 1.2: Catena di compilazione di un programma per un sistema Android.

delle funzionalità disponibili nelle librerie di base del linguaggio di programmazione Java. Ogni applicazione di Android gira col suo proprio processo, con la sua propria istanza sulla virtual machine Dalvik. Dalvik è stata scritta in modo che un dispositivo possa eseguire VMs multiple in modo efficiente. La VM Dalvik esegue i files nel formato Dalvik Executable (.dex), che è ottimizzato per utilizzare un minimo spazio di memoria. A differenza dello stack based della classica JVM, la VM è register-based e funziona con classi compilate da un compilatore Java, trasformate in un formato .dex dallo strumento interno dx. La VM Dalvik si appoggia sul kernel Linux per funzioni di base come la gestione di threading e situazioni di livelli minimi di memoria. Bytecode interpreter può essere eseguito su sistemi con CPU "lente"(250-500 Mhz), con poca RAM (64MB) e senza swap. La compilazione del codice crea un file Dalvik Executable Format di ridotte dimensioni, senza nessun tipo di compressione, che nel caso medio sono minori dei un file JAR, ed una sostanziale differenza semantica con il bytecode Java. In particolare il codice viene compilato con il compilatore standard Java, in seguito viene convertito nel formato .dex (con l'utility dx). Il passo conclusivo, come schematizzato in figura 1.2, è la creazione del package apk che include le risorse di cui necessità l'applicazione.

Kernel Linux

Android si appoggia sulla versione 2.6 di Linux per servizi del sistema centrale come sicurezza, gestione della memoria, esecuzione, network stack, e driver model. Il kernel funziona anche da abstraction layer tra l'hardware e il resto del software, le varie Mobile Manufacters inseriscono i drivers per i propri Hardwares.

1.1.3 Fondamenti delle applicazioni per Android

Le applicazioni Android[2] sono scritte nel linguaggio di programmazione Java. Il codice Java compilato, insieme con tutti i dati e files di risorse richiesti dall'applicazione, viene salvato in un archivio di files contrassegnati da un suffisso APK.

Il file è il veicolo per distribuire l'applicazione e installarlo sui dispositivi mobili. Ad ogni modo, ogni applicazione Android ha un'esistenza propria, indipendente dalle altre, con le seguenti caratteristiche:

- Di default ogni applicazione viene eseguita all'interno del proprio processo linux. Android avvia il processo quando ciascuna parte del codice dell'applicazione necessita di essere eseguita, e chiude il processo quando non è più necessario e le risorse del sistema sono richieste da altre applicazioni.

- Ogni processo ha una sua propria macchina virtuale (VM), per cui il codice dell'applicazione viene eseguito separatamente dal codice di tutte le altre applicazioni.

- Per impostazione predefinita, ad ogni applicazione è assegnato un unico Linux User ID. I permessi sono impostati in modo che i files dell'applicazione siano visibili soltanto a quell'utente e solo all'applicazione stessa, sebbene ci siano modi per esportarli in altre applicazioni. È possibile organizzare due applicazioni affinché condividano lo stesso User ID, nel qual caso esse saranno in grado di vedere i rispettivi file. Per risparmiare risorse di sistema, le applicazioni con lo stesso ID possono anche essere eseguite nello stesso processo Linux, e condividono la medesima VM.

Componenti delle applicazioni

Una caratteristica chiave di Android è che una applicazione può fare uso di elementi di altre applicazioni (purché tali applicazioni lo consentano). Ad esempio, se l'applicazione deve visualizzare un elenco a scorrimento di immagini e in un'altra applicazione è stata sviluppata una barra a scorrimento più adatta e resa disponibile agli altri, è possibile richiamare quella barra quando serve, piuttosto di svilupparne una per conto proprio. La propria applicazione non deve integrare il codice dell'altra o essere collegata ad essa; piuttosto esegue semplicemente quel pezzo di codice quando lo necessita. Affinché tutto ciò funzioni, il sistema deve essere in grado di avviare un processo di un'applicazione quando ogni sua parte è necessaria, e istanziare gli oggetti Java per quella specifica parte. Pertanto, a differenza della maggior parte delle applicazioni presenti su altri sistemi operativi, le applicazioni di Android non hanno un punto di accesso unico per ciascun oggetto/funzione (non c'è una funzione *main()*); al contrario ci sono componenti essenziali che il sistema può istanziare ed eseguire al momento opportuno. Ci

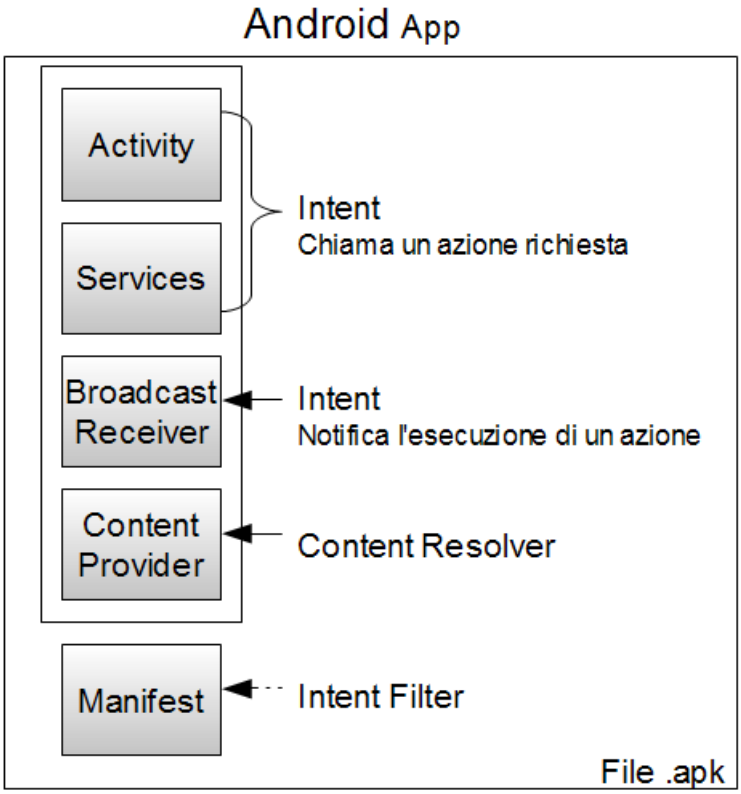


Figura 1.3: Principali componenti di un'applicazione per sistemi Android.

sono quattro tipi di componenti, come si può osservare nello schema in figura 1.3 e che vengono descritti in dettaglio di seguito.

Activities

Un'attività presenta un'interfaccia utente visuale per ciascuna azione che l'utente intenda intraprendere. Ad esempio, un'attività potrebbe presentare un elenco di voci di menù che gli utenti possono scegliere o potrebbe mostrare una collezione di foto con le loro didascalie. Un'applicazione di messaging potrebbe avere un'attività che mostra un elenco di contatti a cui inviare messaggi, una seconda attività per scrivere il messaggio al contatto scelto, e altre attività per rivedere vecchi messaggi o modificare le impostazioni. Anche se lavorano insieme per formare un'interfaccia utente coesa, ogni attività è indipendente dalle altre, ognuna è implementata come una sottoclasse della classe base Activity. Un'applicazione potrebbe consistere di una sola attività o, come le applicazioni di messaging sopra menzionate, ne può contenere parecchie. Quali sono le attività, e quante ce ne sono, dipende dall'applicazione e dalla sua struttura. In genere, una delle attività è contrassegnata come la prima che dovrebbe essere presentata all'utente quando

l'applicazione viene avviata.

Services

Un servizio non dispone di una interfaccia visiva, ma viene eseguito in background per un periodo di tempo indefinito. Ad esempio, un servizio potrebbe eseguire un brano musicale in background mentre l'utente si occupa di altre attività, oppure potrebbe recuperare i dati attraverso la rete o qualcosa da calcolare e fornire il risultato ad un'attività che ne ha bisogno. Ogni servizio estende la classe di base Service.

Broadcast receivers

Un ricevitore broadcast è un componente che non fa altro che ricevere e reagire alle notifiche. Molte notifiche sono generate dal codice di sistema, per esempio, gli annunci che il fuso orario è cambiato, che il livello della batteria è basso, che una foto è stata scattata, o infine, che l'utente ha modificato la lingua di preferenza. Le applicazioni possono anche avviare le notifiche, ad esempio, per far consentire ad altre applicazioni che alcuni dati sono stati scaricati sul dispositivo e sono disponibili per l'uso.

Content providers

Un fornitore di contenuti rende uno specifico insieme di dati di un'applicazione disponibile ad altre applicazioni. I dati possono essere memorizzati nel file system, in un database SQLite, o in qualsiasi altro modo. Tuttavia, le applicazioni non chiamano questi metodi direttamente, ma fanno uso di un oggetto di tipo ContentResolver e si limitano a chiamarne i propri metodi. Un ContentResolver può parlare con qualsiasi provider di contenuto, collabora con il provider per gestire tutte le comunicazioni interprocesso in cui è coinvolto.

Il manifest file

Prima che Android possa avviare una componente applicativa, deve sapere che la componente esiste, pertanto, le applicazioni dichiarano le loro componenti in un file manifesto che si trova nel pacchetto Android, ovvero il file .APK, che contiene anche il codice dell'applicazione stessa, i files e le risorse. Il manifesto è un file XML strutturato ed è sempre chiamato AndroidManifest.xml per tutte le applicazioni. Oltre a dichiarare i componenti dell'applicazione, il manifesto svolge molte altre funzionalità come la denominazione di tutte le librerie alle quali l'applicazione deve essere linkata (oltre a quelle predefinite nella libreria Android) e l'identificazione di tutte le autorizzazioni che l'applicazione si aspetta

siano concesse.

Componenti di attivazione: intents

I Content Provider si attivano quando sono interessati da una richiesta proveniente da un ContentResolver. Gli altri tre componenti ovvero le attività, i servizi e i Broadcast Receivers sono attivati da messaggi asincroni chiamati Intents. Un intent è un oggetto di tipo Intent che contiene il contenuto del messaggio. Per quanto riguarda le attività e i servizi, un intent, tra le altre cose, chiama l'azione richiesta e specifica l'URI dei dati su cui operare. Ad esempio, potrebbe trasmettere la richiesta da parte di un'attività di fornire un'immagine all'utente o lasciare che l'utente modifichi il testo. Al contrario, per quanto riguarda i Broadcast Receivers, l'oggetto Intent chiama l'azione annunciata; ad esempio, potrebbe annunciare alle componenti interessate che il pulsante della fotocamera è stato premuto.

Intent filters

Un oggetto intent può esplicitamente nominare un componente obbiettivo; se accade ciò, Android trova quel componente (in base alle dichiarazioni nel file manifest) e lo attiva. Se il componente obbiettivo (target component) non è esplicitamente indicato, Android deve necessariamente individuare il componente migliore per rispondere all'intent. Ciò viene eseguito mettendo a confronto l'oggetto intent con gli intent filters dei potenziali obbiettivi. Gli intent filters di un componente informano Android circa i tipi di intent che il componente è in grado di gestire. Come accade per tutte le altre informazioni essenziali del componente, sono dichiarati nel file manifesto.

1.1.4 Android Bluetooth

Le librerie Bluetooth[1] sono disponibili in Android solo dalla versione Android 2.0 (SDK API livello 5). È anche importante ricordare che non tutti i dispositivi Android necessariamente includono l'hardware Bluetooth.

L'hardware Bluetooth permette di cercare e connettersi ad altri dispositivi nel raggio d'azione del dispositivo. L'inizializzazione di un link di comunicazione avviene mediante Sockets Bluetooth, con il quale è possibile trasmettere e ricevere flussi di dati tra applicazioni installate su dispositivi differenti. Bluetooth è un protocollo di comunicazione progettato per brevi distanze, per comunicazioni peer-to-peer con un larghezza di banda ridotta. Già nella versione 2.1 di

Android è supportata solo la comunicazione cifrata, ciò significa che è possibile effettuare solo delle connessioni tra dispositivi associati. Il quadro di applicazione consente di accedere alle funzionalità Bluetooth tramite le API Bluetooth Android. Utilizzando le API Bluetooth, un'applicazione Android è in grado di eseguire le seguenti operazioni:

- cercare altri dispositivi Bluetooth;
- interrogare il dispositivo Bluetooth per identificare i dispositivi associati;
- stabilire canali RFCOMM;
- collegarsi ad altri dispositivi attraverso la scoperta di servizio;
- trasferimento dei dati da e verso altri dispositivi;
- gestione di connessioni multiple.

In Android i dispositivi Bluetooth e le connessioni sono gestite secondo le seguenti classi[12]:

- BluetoothAdapter. Il Bluetooth Adapter rappresenta il dispositivo Bluetooth locale, cioè il dispositivo Android su cui l'applicazione è in esecuzione.
- BluetoothDevice. Ogni dispositivo remoto con cui si desidera comunicare è rappresentato come BluetoothDevice.
- BluetoothSocket. La chiamata createRfcommSocketToServiceRecord su un oggetto Bluetooth Device crea un Bluetooth Socket che permette di effettuare una richiesta di connessione al dispositivo remote e iniziare una comunicazione.
- BluetoothServerSocket. Creando un Bluetooth Server Socket (utilizzando il metodo listenUsingRfcommWithServiceRecord) sul dispositivo locale, è possibile ricevere le richieste di connessione in entrata da un Bluetooth Sockets su dispositivi remoti.

Un oggetto Bluetooth Adapter offre metodi per la lettura e l'impostazione delle proprietà del hardware Bluetooth. Se l'hardware Bluetooth è acceso, e se si dispone dei permessi per la gestione del Bluetooth definiti sul file Manifest, è

possibile, ad esempio, accedere al nome di presentazione del dispositivo (friendly name, una stringa arbitraria che gli utenti possono impostare e quindi utilizzare per identificare il dispositivo) e all'indirizzo hardware.

Il processo con il quale due dispositivi possono trovarsi per connettersi è chiamato discovery. Prima di poter stabilire un Bluetooth Socket per le comunicazioni, il Bluetooth Adapter locale deve essere associato con il dispositivo remoto (ovvero autorizzato per effettuare il collegamento). Ancor prima di collegarsi, due dispositivi hanno prima bisogno di scoprirsi l'un l'altro. In questo caso per iniziare un'operazione di discovery il dispositivo deve impostare le proprietà del Bluetooth Adapter come visibile ad altri dispositivi.

Una volta individuato il dispositivo e stabilita la connessione è possibile effettuare la comunicazione e lo scambio di informazioni. Le API di comunicazione Bluetooth sono contenute nel protocollo RFCOMM (Radio Frequency COMMunications protocol per bluetooth). RFCOMM supporta la comunicazione seriale RS232 attraverso il livello L2CAP (Logical Link Control and Adaptation Protocol layer).

Quando si crea un'applicazione che utilizza Bluetooth come connessione peer-to-peer, è necessario implementare sia un Bluetooth Server Socket per la ricezione delle connessioni che un Bluetooth Socket per avviare e gestire un nuovo canale di comunicazioni.

1.2 NesC

Il NesC[8] è una variante del linguaggio di programmazione C per lo sviluppo di applicazione enbedded e nella fattispecie strettamente legato all'architettura del sistema TinyOS, tanto da influenzarne la filosofia. Per alcuni versi questo linguaggio rappresenta un'estensione del C, introducendo un modello di programmazione orientato agli eventi, mentre per altri ne riduce le funzioni, come ad esempio limitando l'utilizzo dei puntatori. L'utilizzo della sintassi del linguaggio C ha come vantaggio quello di produrre un codice efficiente per tutti i microcontrollori utilizzati nei motes, rendendo disponibili tutte le istruzioni necessarie per interfacciarsi con l'hardware dei dispositivi, e come ulteriore vantaggio la famigliarità di molti programmatori. Le proprietà base che dettano le linee guida dell'architettura del NesC sono[9]:

- un linguaggio statico: non c'è allocazione dinamica della memoria nei programmi scritti in NesC di conseguenza l'albero delle chiamate e completamente conosciuto in fase di compilazione. In questo modo l'analisi e l'ottimizzazione del programma sono rese più semplici e accurate. L'allocazione dinamica viene evitata grazie all'architettura modulare del linguaggio e l'utilizzo di interfacce parametrizzate;

- analisi dell'intero programma: i programmi scritti in NesC sono soggetti a un'analisi e un'ottimizzazione completa del codice, per motivi di sicurezza e per migliorare le performance dell'applicazione;

- supporta e riflette l'architettura del TinyOS: il NesC è basato sul concetto delle componenti e supporta il modello di concorrenza guidato agli eventi del TinyOS. Inoltre il NesC utilizza indirizzi assoluti per l'accesso ai dati rendendo possibile un'analisi sulle possibili condizioni di concorrenza nel codice;

Sviluppare un'applicazione in NesC significa realizzare una serie di componenti che verranno poi assemblati per produrre il codice eseguibile. Ogni componente ha due obbiettivi da realizzare: deve definire le specifiche del suo comportamento e deve implementare tali specifiche.

Linguaggio ad eventi

Per realizzare l'obbiettivo del risparmio energetico il NesC permette di definire un elaborazione event-driven in cui le componenti di un'applicazione vengono mandate in esecuzione solo al verificarsi dell'evento associato a ciascun componente. All'interno di un'implementazione è possibile sollevare eventi (*event*, tipicamente verso moduli di livello più alto) tramite la parola chiave *signal*, mentre è possibile invocare comandi (*command*, tipicamente su componenti più vicini all'hardware) tramite la parola chiave *call*. I comandi sono richieste di un servizio fornito da un componente di livello più basso e non sono bloccanti. Generalmente un comando deposita dei parametri nel frame (con questo termine si indica un'area di memoria riservata per eseguire i calcoli) e può attivare un *task*, ma è possibile anche che questo chiami un altro comando. In quest'ultimo caso, il componente non può attendere per un tempo indeterminato la fine della chiamata, allora il comando termina e ritorna un valore che indica se la chiamata ha avuto successo o meno. Gli eventi sono, direttamente o indirettamente, dei gestori delle inter-

ruzioni hardware. Il componente di livello più basso trasforma un'interruzione hardware in un evento che può essere provocato da un'interruzione esterna, da timer, o dal contatore, e poi propaga tale richiesta ai livelli più alti. Similmente ai comandi, un evento può depositare dei parametri nel frame e può attivare un *task*. Un evento può richiamare altri eventi e alla fine chiamare un comando, come a formare una catena che prima sale e poi scende. Per evitare che questa catena si possa chiudere viene impedito ai comandi di generare eventi. Ad esempio un comando o un evento `f` in un'interfaccia `i` è chiamato `i.f`, questo viene definito premettendo la parola chiave *commad* o *event*. Una *command call* è come una normale chiamata di una funzione con la premessa della parola chiave *call*, allo stesso modo *event signal* è come una chiamata di una funzione con premessa la parola chiave *signal*.

I componenti

Un'applicazione NesC è un insieme di componenti collegati tramite interfacce (tra i componenti che formano un'applicazione ci sono anche quelli dello stesso sistema operativo). Questo approccio separa la costruzione dei componenti dalla composizione degli stessi. La scomposizione di un'applicazione in componenti è vantaggiosa in quanto permette di creare un livello di astrazione sulle componenti hardware del dispositivo, specialmente in un ambiente in cui le applicazioni fanno abbondante uso di istruzioni a basso livello.

Interfacce

Un'interfaccia dichiara una serie di funzioni che possono essere di due tipi: comandi ed eventi. I comandi devono essere implementati dal componente che li fornisce mentre gli eventi devono essere implementati dal componente che li usa. Un comando generalmente è una richiesta di servizio, mentre l'evento segnala il completamento di un servizio. Gli eventi possono essere generati anche in modo asincrono, per esempio in seguito ad un'interruzione hardware o all'arrivo di un pacchetto di una comunicazione radio. Ogni interfaccia è bidirezionale e modella un servizio offerto/utilizzato dal componente. Si possono usare o fornire più di un'interfaccia e istanze multiple di una stessa interfaccia (ovviamente rinominandole). L'aspetto importante è che le interfacce sono bidirezionali rendendo più facile la gestione delle interruzioni hardware A differenza di interfacce unidirezionali basate su procedure calls che costringono ad un sistema di hardware polling o ad avere due interfacce separate per le operazioni hardware e la gestione delle

corrispondenti interruzioni.

I tipi di componenti

Esistono due tipi di implementazione di componenti: i moduli e le configurazioni. I moduli forniscono il codice esecutivo, implementando una o più interfacce. Il modulo implementa inoltre gli eventi e i comandi in modo molto simile a come vengono implementati i sottoprogrammi in C. Esistono due differenza sostanziali, la prima riguarda la definizione dell'evento o del comando, che viene preceduta dal nome della relativa interfaccia. Mentre la seconda riguarda il valore di ritorno degli eventi e dei comandi da una particolare macro funzione, che indica la corretta o meno terminazione della chiamata. Le configurazioni sono usate per legare insieme altre componenti, collegando le interfacce usate da alcune componenti alle interfacce fornite da altre, in questo modo risultano essere gli unici punti di accesso al codice per la costruzione delle applicazioni. Infatti ogni applicazione in NESC è descritta da una configurazione detta toplevel configuration che lega insieme i componenti usati. Si crea in questo modo una sorta di stratificazione dell'applicazione, dove a livello più basso si posso individuare i componenti che realizzano le operazioni legate alla piattaforma hardware. Mentre salendo nella stratificazione si evidenziano le astrazioni di particolari servizi fino a giungere alla configurazione che collega i diversi servizi.

Assemblaggio e compilazione dei componenti

Per realizzare un'applicazione è necessario collegare i vari componenti. Questa operazione viene definita *wiring* produce il cosiddetto grafo dei componenti. In figura 1.4 viene mostrato un esempio che ora analizzeremo in dettaglio. I componenti:

- *ComponentD*,
- *ComponentF*,
- *Application*,

sono delle configurazioni e, come abbiamo già spiegato, servono a collegare tra loro componenti preesistenti. Il resto dei componenti sono dei moduli. Il componente *ComponentD* fornisce un'interfaccia che verrà usata dal componente Application (indicato in giallo) e utilizza un'interfaccia che sarà fornita da ComponentF. Il componente *ComponentF* fornisce una sola interfaccia che corrisponde a quella fornita dal componente Radio . La configurazione *Application*

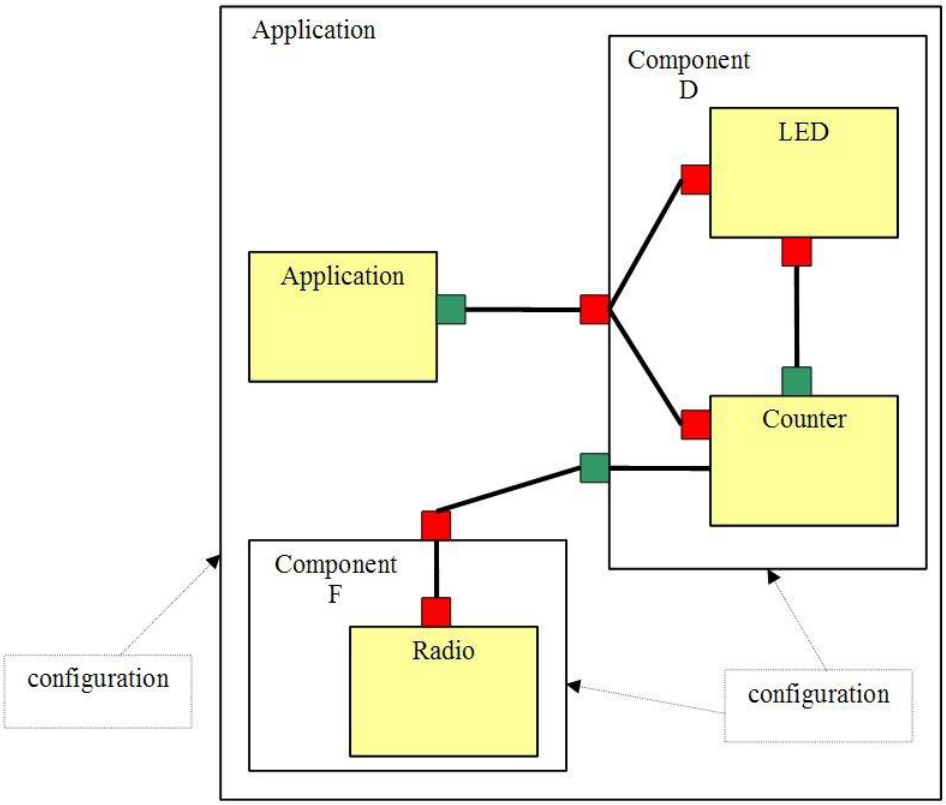


Figura 1.4: Esempio di wiring di un'applicazione sviluppata in NesC.

costituisce l'applicazione vera e propria. Non utilizza ne fornisce interfacce ma effettua il semplice collegamento tra i componenti *Application*, *ComponentD* e *ComponentF*. Come possiamo notare la ragione per cui un componente si distingue in moduli e configurazioni è per favorire la modularità di un applicazione. Ciò permette allo sviluppatore di assemblare velocemente le applicazioni. In effetti, si potrebbe realizzare un'applicazione solo scrivendo un componente di configurazione che assembli componenti già esistenti. D'altra parte questo modello incoraggia l'aggiunta di librerie di componenti tali da implementare algoritmi e protocolli che possano essere utilizzati in una qualsiasi applicazione.

Il sistema operativo TinyOS è programmato in NesC, e nel momento in cui un'applicazione viene compilata, i componenti di TinyOS vengono compilati insieme ad essa e il risultato costituisce l'intero software del sensore. Questo approccio consente un ingente risparmio di energia e di memoria, tuttavia limita molto la versatilità, infatti non è possibile installare più applicazioni indipendenti sullo stesso sensore e non è possibile effettuare linking dinamico di librerie esterne o riconfigurazione dinamica di parte del codice presente sul sensore.

1.3 TinyOS

Il TinyOS[11] è un sistema operativo open-source, sviluppato dalla University of California at Berkeley. Data la possibilità di modificarne il codice, questo sistema operativo è diventato un'importante piattaforma di sviluppo per ogni soluzione proposta nel campo delle reti di sensori. In effetti grossi contributi sono stati forniti dalla comunità di sviluppatori, lo testimonia la lunga lista di progetti attivi relativi a tutti i campi della ricerca, da protocolli per l'instradamento dei pacchetti alla localizzazione, dalla realizzazione di un interfaccia grafica per lo sviluppo delle applicazioni all'estensione del compilatore ncc per il supporto di nuove piattaforme hardware. A differenza delle tradizionali architetture hardware, dove disponiamo di grandi quantità di memoria, complessi sottosistemi per la gestione dei dati in ingresso e per quelli in uscita, forti capacità di elaborazione e sorgenti di energia praticamente illimitate, nelle reti di sensori ci troviamo a confronto con sistemi di piccole dimensioni, fonti di energia limitate, scarsa quantità di memoria, modeste capacità di elaborazione, etc. Sono necessarie quindi soluzioni molto semplici ed efficienti, e che soprattutto riducano alla massimo i consumi di energia. Lo scopo dichiarato dei progettisti di TinyOS era infatti quello di[10]:

- ridurre i consumi di energia;
- ridurre il carico computazionale e le dimensioni del sistema operativo;
- supportare intensive richieste di operazioni che devono essere svolte in concorrenza e in maniera tale da raggiungere un alto livello di robustezza ed un efficiente modularità.

Tutto ciò si può riassumere in alcuni aspetti fondamentali dell'architettura del TinyOS, che vengono elencati di seguito:

- sistema basato su componenti (Component-based architecture);
- modello di concorrenza orientato ai processi e agli eventi (Tasks and event-based concurrency);
- utilizzo della tecnica split-phase operations (ogni funzione viene portata a termine senza aspettarne il risultato finale, questo verrà successivamente segnalato da un evento).

Quello che si ottiene è un sistema con un nucleo molto snello. Il consumo di energia rappresenta un fattore critico e l'approccio basato sugli eventi utilizza il microprocessore nella maniera più efficiente possibile. Quando il sistema viene sollecitato da un evento questo viene gestito immediatamente e rapidamente. In effetti non sono permesse condizioni di bloccaggio né attese attive che sprecherebbero energia inutilmente. Quando invece non ci sono attività da eseguire il sistema mette a riposo il microprocessore che viene risvegliato all'arrivo di un evento. Nei tradizionali modelli, infatti, è necessario allocare una certa quantità di memoria sullo stack per ogni attività in esecuzione, in questo modo il sistema è soggetto a frequenti commutazioni di contesto per servire ogni tipo di richiesta, come l'invio di pacchetti, la lettura di un dato su un sensore, etc. Naturalmente i nodi non dispongono di grosse quantità di memoria ed il modello ad eventi ben si addice ad un sistema continuamente sollecitato. Dunque per le applicazioni non viene effettuato alcun cambio di contesto, la memoria viene infatti considerata come un unico e lineare spazio fisico, che viene assegnato alle applicazioni a compile time, viene così eliminato qualsiasi tipo di overhead, che causerebbe inutile spreco di energia. Per soddisfare il requisito della modularità, TinyOS favorisce lo sviluppo di una serie di piccoli componenti, ognuno con una ben precisa funzione, che realizza un qualche aspetto dell'hardware del sistema o di un'applicazione. Ogni componente poi, definisce un'interfaccia che garantisce la riusabilità del componente ed eventualmente la sua sostituzione.

Capitolo 2

Componenti hardware

2.1 Tmote Sky

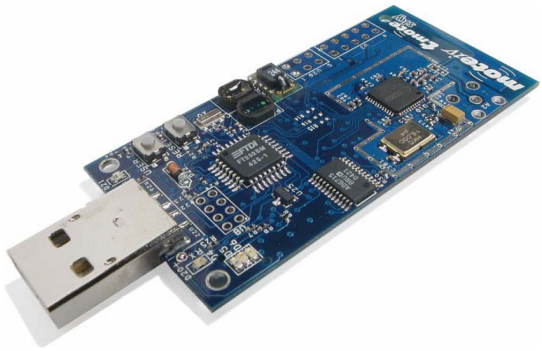


Figura 2.1: Tmote Sky[3].

I mote utilizzati nell'esperimento, a cui faremo riferimento nel testo, sono i Tmote Sky[3] prodotti della MoteIv corporation (fig. 2.1); sono dispositivi a bassissima potenza dotati di diversi sensori ambientali. Vediamo nei particolari di cosa è composto:

Microprocessore

Prodotto dalla Texas Instruments, l'MSP430 è caratterizzato da 10kB di RAM, 48kB di flash ROM e un'architettura 16bit RISC. Ha un oscillatore (DCO) che opera sopra gli 8 MHz e un oscillatore al quarzo utile alla calibrazione del primo. È dotato inoltre di 8 porte ADC interne e altrettante esterne, necessarie per leggere i valori dei sensori o dei livelli di batterie.

Radio

Tmote utilizza il Chipcon CC2420; fornisce tutto il necessario per comunicazioni

Livello di potenza	Potenza in uscita (dBm)	Consumo corrente (mA)
31	0	17.4
27	-1	16.5
23	-3	15.2
19	-5	13.9
15	-7	12.5
11	-10	11.2
7	-15	9.9
3	-25	8.5

Tabella 2.1: Potenza di trasmissione radio e relativi consumi di corrente.

wireless affidabili basate sullo standard IEEE 802.15.4. Il chip radio viene controllato dall'MSP430 tramite il bus SPI e possono essere programmati i parametri fondamentali per la trasmissione: potenza, frequenza e gruppo. La radio lavora, infatti, attorno ai 2.4GHz con una modulazione O-QPSK, ma è possibile selezionare tra 16 canali differenti numerati da 11 a 26, con i quali ci si può spostare dalla frequenza base con step di 5 MHz per canale. Per quanto riguarda la potenza in trasmissione, sono disponibili 32 diversi livelli, quantizzati in 8 possibili valori elencati in tabella 2.1 dal più elevato (31) al più debole (3).

L'integrato CC2420 fornisce un indice di robustezza del segnale ricevuto (RSSI) e un indicatore di qualità della connessione (LQI). L'interfaccia USB Tmote Sky utilizza un USB controller per gestire la comunicazione seriale col PC. Una volta inserito sulla relativa borchia, il Tmote-Sky verrà visualizzato come dispositivo nelle macchine Linux o OSX, e come COM in Windows; gli verrà inoltre assegnato un indirizzo identificativo univoco. Ad esempio col comando motelist si verifica quali dispositivi sono collegati al computer (esempio in tabella 2.2), in questo caso siamo in ambiente Linux e abbiamo due Tmote collegati: al primo viene assegnato l'identificativo /dev/ttyUSB0, al secondo /dev/ttyUSB1; la prima colonna indica invece il codice che identifica il singolo nodo.

Antenna

L'antenna è interna, posta parallelamente alla superficie della piastra, ma esiste la possibilità di montarne di esterne tramite il connettore apposito. Ha un range di copertura di 50 metri indoor e oltre 125 metri outdoor, ma non è perfetta-

>motelist		
Reference	Device	Description
NAV00065	/dev/ttyUSB0	DEI NAVLAB mote
NAV00063	/dev/ttyUSB1	DEI NAVLAB mote

Tabella 2.2: Esempio output del comando motelist.

mente omnidirezionale e l'irragiamento può risentire della presenza o meno delle batterie e persino della posizione del mote (si ha un incremento delle prestazioni posizionandolo in verticale piuttosto che disteso).

Flash esterna

L'ST M25P80 40MHz fornisce 1024kB per memorizzare dati: è composta di 16 segmenti di 64kB l'uno ed è collegata al microcontrollore tramite l'SPI, lo stesso bus che collegava calcolatore e radio. Questa condivisione delle risorse richiede un minimo di gestione; infatti lo stesso mezzo può essere utilizzato da un unico componente alla volta e non è possibile, per esempio, scrivere nella flash e ascoltare la radio contemporaneamente.

Sensori

Come già detto, c'è la possibilità di integrare questi mote con dei sensori ambientali: Sensirion AG SHT11 o SHT15; tra i due la differenza principale è l'accuratezza e, per entrambi la misura, ottenuta per conversione dall'ADC, è a 14 bit. L'MSP430 dispone comunque, anche in assenza dei sensori suddetti, di un indicatore del livello delle batterie e uno per la temperatura interna del microcontrollore; entrambi non sono molto precisi e richiedono di essere calibrati.

Connettori per l'espansione

I connettori per l'espansione sono utili per controllare, tramite Tmote, altri dispositivi (per esempio relè, display LCD e altre periferiche digitali); ce ne sono due, uno a 6 e uno a 10 pin. Considerando l'alimentazione a batterie, è d'obbligo tenere in considerazione il livello minimo per il quale il dispositivo funziona correttamente. Tale livello non è lo stesso per tutti i componenti, come mostrato in tabella 2.3.

Il Tmote ha a disposizione due vie di comunicazione seriale, UART1 e UART0, la prima viene utilizzata per la connessione del dispositivo con il PC, per un suo utilizzo come Base Station e quindi monitoraggio della rete di sensori. Questa

	MIN	MAX	Unità
Tensione al microcontrollore durante l'esecuzione del programma	1,8	3,6	V
Tensione al microcontrollore durante la programmazione della flash	2,7	3,6	V
Tensione di alimentazione radio	2,1	3,6	V

Tabella 2.3: Range di tensione.

porta è, inoltre, connessa al JTAG che ne rende possibile la riprogrammazione del microcontrollore. La seconda, UART0, come si può osservare nello schema a blocchi (fig. 2.2), che descrive le connessioni del microcontrollore. È possibile accedere a questa porta dal blocco di 10 pin di estensione del Tmote, dunque può essere utilizzata per la connessione e lo scambio di messaggi con un modulo Bluetooth esterno. Il protocollo per la comunicazione seriale viene descritto nella documentazione di TinyOs in TEP113 [15].

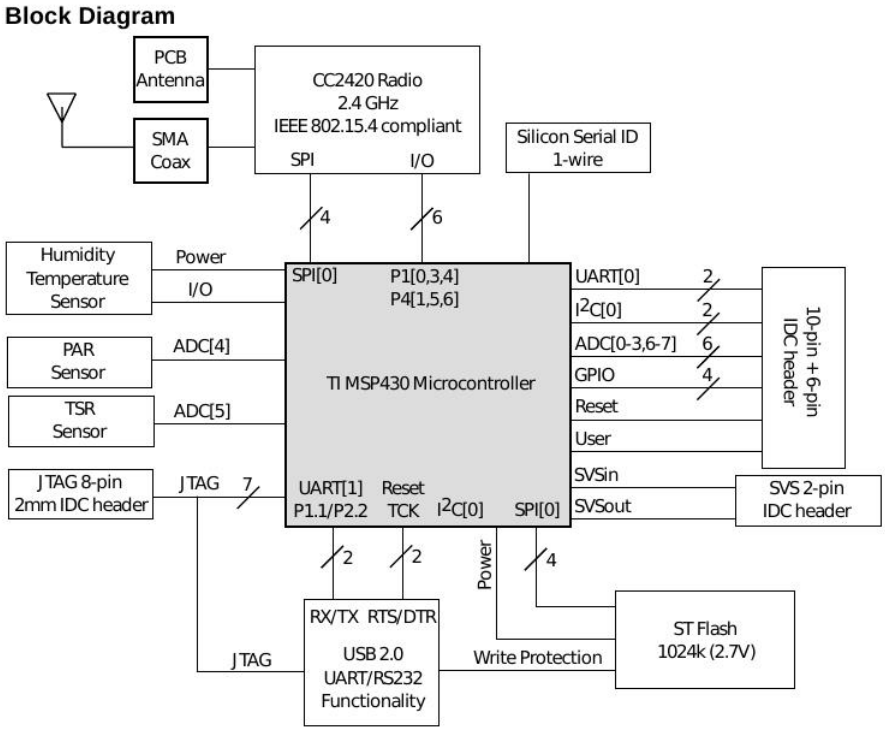


Figura 2.2: Diagramma a blocchi del Tmote-sky.

2.2 Parani-ESD100V2

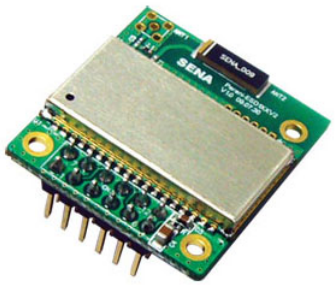


Figura 2.3: Parani-ESD100V2.

Per ottenere una connessione tra il TmoteSky e il cellulare Android è necessario utilizzare una scheda Bluetooth da collegare al mote, per effettuare il forwarding dei messaggi in entrambe le direzioni. A questo scopo è stata scelta la scheda Parani-ESD (2.3), una scheda Bluetooth commercializzata dalla SENA Technologies Inc.

Parani-ESD è un dispositivo per le comunicazioni wireless che sfrutta lo standard Bluetooth, ha dunque la possibilità di comunicare con altri dispositivi Bluetooth che supportano il profilo SPP (Serial Port Profile). Può essere utilizzata per sostituire un cavo RS232, con la possibilità di aumentare la distanza di comunicazione. Il protocollo Bluetooth implementato supporta FHSS (Frequency Hopping Spread Spectrum), sistema utilizzato per minimizzare le interferenze radio quando diminuisce la probabilità di intercettazione dei dati, Parani-ESD supporta, inoltre, l'autenticazione e la crittografia dei dati. Tutti i parametri di questo dispositivo possono essere impostati e controllati attraverso degli AT command, questi possono essere impostati attraverso un terminale ad esempio HyperTerminal di Windows, e utilizzare la comunicazione Bluetooth senza dover modificare i programmi di comunicazione seriale già esistenti. Elenchiamo in tabella 2.4 alcune caratteristiche princilpali della scheda.

L'insieme di comandi AT command è di fatto un linguaggio standard per il controllo dei modem, e sono riconosciuti da tutti i modem per computer. Parani-ESD mette a disposizione un insieme di AT command esteso per consentire il controllo e la configurazione dei parametri per la comunicazione Bluetooth. Parani-ESD risponde ai comandi con quattro tipi di messaggio, OK, ERROR,

CONNECT e DISCONNECT. La scheda possiede quattro modi operativi e tre stati per l'esecuzione dei comandi che vengono elencati nella tabelle 2.5.

In figura 2.4 riportiamo lo schema e in figura 2.5 la descrizione dei connettori utilizzati per il collegamento della scheda parani-ESD100 con altri dispositivi.

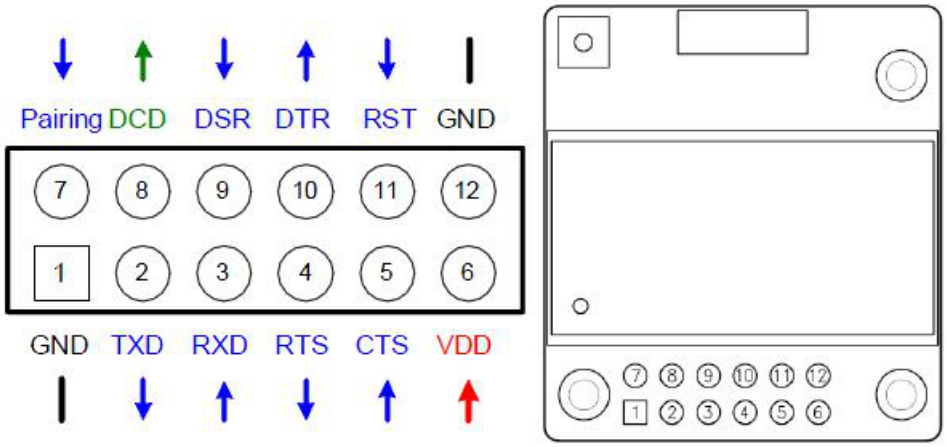


Figura 2.4: Connettori presenti sulla scheda Parani-ESD100V2.

Pin #	Signal	Direction	Description	Signal Level
1	GND	-	Power Ground	Ground
2	TxD	Output	UART Data Output	TTL
3	RxD	Input	UART Data Input	TTL
4	RTS	Output	UART Ready to Send	TTL
5	CTS	Input	UART Clear to Send	TTL
6	VDD	Input	DC Input (3.0~3.3V)	Power
7	Pairing	Input	Pairing Input (Active Low)	TTL
8	DCD	Output	Bluetooth Connect Detect (Active Low)	TTL
9	DSR	Input	Data Set Ready	TTL
10	DTR	Output	Data Terminal Ready	TTL
11	RST	Input	Reset (Active Low)	TTL
12	GND	-	Power Ground	Ground

Figura 2.5: Descrizione dei connettori presenti sulla scheda Parani-ESD100V2.

Gli utenti possono creare una connessione tra un'unità Parani-ESD e altri dispositivi bluetooth seguendo le seguenti fasi:

- Fase 1. Accendere ESD1 e impostare i parametri di default usando il segnale RST.

- Fase 2. Impostare il segnale di accoppiamento di ESD1 ad uno stato basso e mantenere così il segnale per 2 secondi.

- Fase 3. Gli utenti possono scoprire e connettersi a ESD1 utilizzando il software, oppure l'interfaccia utente di altri dispositivi bluetooth con cui si vuole connettere.

- Fase 4. Attendere che il dispositivo ESD1 e gli altri dispositivi bluetooth si connettano tra loro. Questa attività può richiedere circa 10 secondi per stabilire una connessione. Se ci sono molti dispositivi bluetooth nelle vicinanze, il tempo di connessione può aumentare.

- Fase 5. Ora ESD1 è in attesa di una connessione dall'ultimo dispositivo bluetooth connesso. L'ultimo dispositivo bluetooth collegato può connettersi al ESD1.

In figura 2.6 viene schematizzato il processo di accoppiamento con un dispositivo bluetooth generico con l'utilizzo di segnali di accoppiamento.

ESD1	Status	Pairing Signal	Other Bluetooth Deivce	Status
1. Reset	Mode0	HIGH		
2. Drop pairing signal	Mode3	LOW		
			3. Inquiry and connect to ESD1	
4. Connected	Slave	HIGH	4. Connected	Master

Figura 2.6: Schema del processo di accoppiamento di una scheda Parani-ESD100V2.

Inoltre in figura 2.7 riportiamo lo schema elettrico di collegamento tra parani-ESD100 e un generico dispositivo nel caso in cui il livello di alimentazione di MICOM sia di 3,3V e non venga utilizzato il controllo di flusso.

2.3 Smartphone

Per completare questa parte introduttiva degli strumenti hardware utilizzati per lo sviluppo del progetto diamo una breve panoramica dello smartphone utilizzato. Nel caso particolare si dispone dello smarphone LG Optimun One della

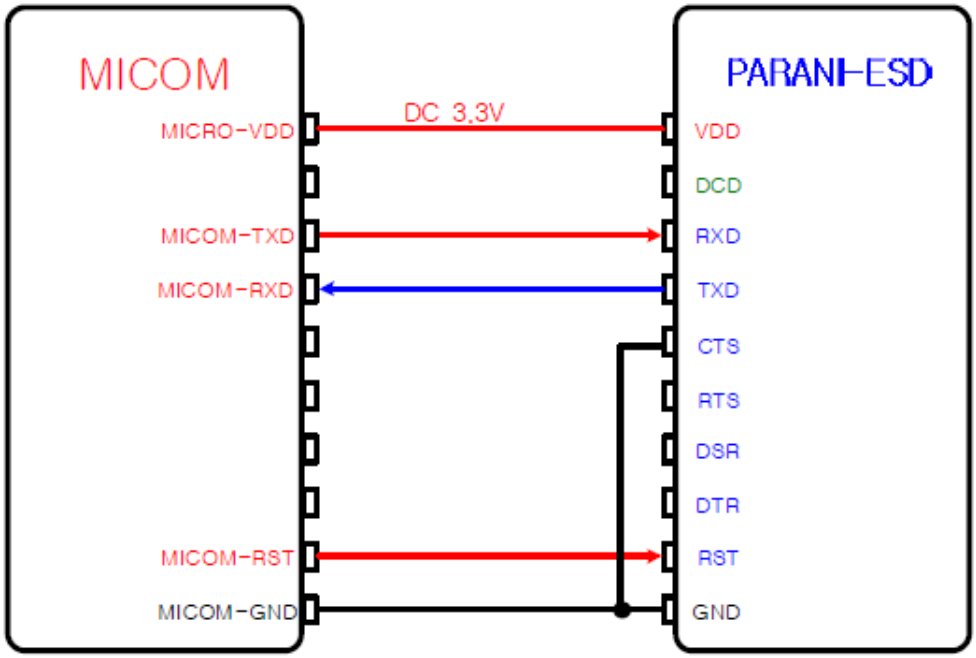


Figura 2.7: Schema delle connessioni fisiche dei connettori tra una scheda Parani-ESD100V2 e un generico dispositivo.

serie P500[6] dotato di sistema operativo Google Android 2.2 (fig. 2.8). È importante sottolineare tuttavia che una qualsiasi altra marca e modello di cellulare può essere utilizzato, purché, cosa indispensabile, abbiano installato lo stesso sistema operativo con versione equivalente o maggiore e abbiano in dotazione una scheda per il collegamento bluetooth. In tabella 2.7 elenchiamo le principali caratteristiche di nostro interesse.

Figura 2.8: Smartphone LG Optimus One P500.

Serial Interface	Serial UART speed up to 921.6kbps CTS/RTS flow control, DTR/DSR for loop-back & full transfer 2.54mmPin Header 2X6 (12pin)
Bluetooth Interface	Bluetooth v2.0 + EDR Profile: Serial Port Profile Class 1 Working distance: Nominal 100m
Power	Supply voltage: 3.3V DC Supply current: 10mA - 60mA
Environmental	Operating temperature: -30 80 °C Storage temperature: -40 85 °C Humidity : 90% (Non-condensing)
Physical properties	Dimension 27.5 mm L (1.08 in) 30.0 mmW (1.18 in) 14.0 mm H (0.55 in) Weight 6 g
RF Information	
Radio Frequency Range	2.402 2.480GHz
Number of Frequency Channel	79 channels
Transmission Method	FHSS(Frequency Hopping Spread Spectrum)
Modulation Method	1Mbps: GFSK(Gaussian Frequency Shift Keying) 2Mbps: p/4 DQPSK(pi/4 rotated Differential Quaternary Phase Shift Keying) 3Mbps: 8DPSK(8 phase Differential Phase Shift Keying)
Radio Output Power	+18dBm
Receiving Sensitivity	-90dBm

Tabella 2.4: Caratteristiche principali di Parani-ESD100V2[5].

Operation Mode	
Mode	Description
Mode0	Waiting for AT commands
Mode1	Attempting to connect to the last connected Bluetooth device
Mode2	Waiting for a connection from the last connected Bluetooth device
Mode3	Waiting for the connection from another Bluetooth device

Tabella 2.5: Modi operativi della scheda Parani-ESD100V2[5].

Operation Status	
Status	Description
Standby	Waiting for AT commands
Pending	Executing tasks
Connect	Transmitting data

Tabella 2.6: Stati di esecuzione dei comandi della scheda Parani-ESD100V2[5].

MISURE E DIMENSIONI	
Peso:	129 gr.
Altezza:	114 mm.
Larghezza:	59 mm.
Profondità:	13 mm.
AUTONOMIA	
Batteria:	Li-Ion 1500 mAh
Standby:	100 h
Conversazione:	4 h
HARDWARE E SO	
S.O.:	Android OS
Versione:	2.2 Froyo
Processore:	Qualcomm MSM7227 a 600 MHz
RAM:	512 MB
ROM:	256 MB
DATI E CONNETTIVITA'	
Usb:	Si (2.0)
Miniusb:	Si
Pc Sync:	Si
Bluetooth:	Si (2.1 con A2DP)
Wifi:	Si (802.11 b/g)
MEMORIA	
Interna:	150 Mb
Esterna:	microSD e microSDHC fino a 32 GB

Tabella 2.7: Dati tecnici dello smartphone LG Optimus One.

Capitolo 3

Sviluppo

In questo capitolo andremo ad analizzare nel dettaglio lo schema di collegamento elettrico e le principali caratteristiche del software sviluppato. Il software si compone di tre applicazioni, due sviluppate in Nesc per la programmazione del Tmote e una in Java per la programmazione sul sistema Android. Le applicazioni prendono il nome dal progetto BlueTmote.

- la prima applicazione, BlueTmoteAppC.nc, è stata sviluppata per testare la funzionalità della comunicazione bluetooth tra il cellulare Android e il Tmote, valutarne i punti che richiedono maggiore attenzione e un particolare sviluppo;

- la seconda componente sviluppata, SerialBluetoothC.nc, in Nesc si occupa della gestione della comunicazione sul bus USART0 del Tmote collegato al dispositivo bluetooth Parani;

- BlueTmote.java, dal lato dello smatphone, è la terza applicazione sviluppata per la gestione della comunicazione bluetooth, attraverso le API messe a disposizione dal sistema Android, per lo scambio di informazioni con il Tmote. Nella Fattispecie realizza l'operazione di deployment dei nodi fissi, impostando per ciascuno di essi le coordinate x-y;

Nelle prossime sezione andremo per tanto ad analizzare il software come elencato in precedenza, evidenziando le problematiche che richiedono una particolare attenzione e i punti di interesse nello sviluppo.

3.1 Schema di collegamento

La scheda bluetooth Parani mette a disposizione diversi pin per la comunicazione e il controllo delle scheda come si può osservare in figura 2.4. Ai fini di questo progetto si segue lo schema di connessione descritto nella figura 2.7 tra Parani e un dispositivo generico. Nella tabella seguente (3.1) vediamo la connessione tra i pin delle due schede, mentre si riporta in appendice A l'immagine del collegamento ottenuto.

Tmote pin	Parani pin
pin 2, Uart RX	pin 2, Uart TX
pin 4, Uart TX	pin 3, Uart RX
pin 9, Ground	pin 1, Ground; pin 5, Clear To Send (CTS); pin 12, Ground

Tabella 3.1: Collegamento tra i pin del Tmote e della scheda Parani.

Per quanto riguarda l'alimentazione la scheda Parani viene alimentata da un pacchetto di due batterie stilo da 1,5V collegate in serie. Le batterie sono unite ai pin 1 e 9 della scheda Parani. Si osserva che tuttavia la scheda è già collegata al pin di ground del Tmote, questo tipo di collegamento è utile per mantenere stabile il valore di tensione sul canale di comunicazione tra le due schede e facilitare la loro sincronizzazione. Inoltre è necessario collegare il pin 11 di reset della scheda Parani con il pin di alimentazione Vcc, quando il pin di reset viene cortocircuitato con il ground la scheda Parani si resetta alle impostazioni di fabbrica.

Un ulteriore aspetto da tenere presente, perché non ben documentato nei manuali del Tmote e nella documentazione del TinyOs, è la caratteristica operativa del pin 4 ovvero il pin di trasmissione. Tale pin opera nello stato di idle-low, come descritto in [13], ciò significa che nei periodi di idle il pin mantiene la tensione sul pin di trasmissione bassa e nel momento in cui avviene la trasmissione del segnale la porta allo stato alto. Questo causa la mancata sincronizzazione con altri dispositivi che richiedono la tensione sul pin corrispondente a livello alto, come richiesto dallo standard di comunicazione RS232. Per modificare questa caratteristica del Tmote è sufficiente eseguire due istruzioni, elencate di seguito, messe a disposizione dal TinyOs nella fase di inizializzazione del dispositivo.

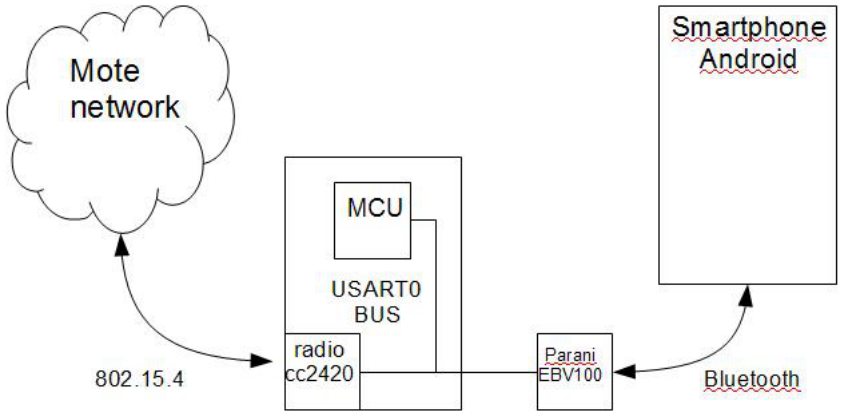


Figura 3.1: Schema del collegamento tra Tmote e Parani, in cui si evidenzia la condivisione del bus.

```
TOSH_MAKE_UTXD0_OUTPUT();
TOSH_SET_UTXD0_PIN();
```

Il risultato della connessione tra i dispositivi Tmote e Parani è schematizzato nella figura 3.1 , nella quale viene messo in risalto la condivisione del bus tra i due chip radio, uno integrato e l'altro esterno al Tmote e la collocazione di questo dispositivo all'interno di una rete di sensori.

3.2 L'applicazione per TmoteSky

L'applicazione è formata da due componenti principali la prima, BlueTmote, che si occupa della gestione della condivisione del bus USART0, dove sono collegati il chip radio e i pin di comunicazione seriale RS232, e a quest'ultimi il chip bluetooth esterno. Mentre la seconda componente, SerialBluetooth, si occupa della configurazione dei parametri del bus come porta seriale e di implementare i metodi di invio e ricezione dei pacchetti al chip bluetooth. Questo tipo di implementazione è stata pensata per rendere indipendente la parte applicativa che necessita di comunicare attraverso il chip bluetooth e quelle operazioni che distinguono le applicazioni l'una dall'altra. In questo modo si può pensare ad una applicazione senza doversi preoccupare di sviluppare ogni volta le funzioni di comunicazione.

3.2.1 BlueTmote

L'applicazione per TinyOs è composta dal file di configurazione BlueTmoteAppC.nc (appendice B) nel quale si possono osservare le componenti di cui necessità l'applicazione per l'esecuzione. Mettiamo in evidenza di seguito le componenti per l'arbitraggio del bus.

```
#define TEST_ARBITER_RESOURCE   "Test.Arbiter.Resource"
  ...
  // Resource arbiter
  components new FcfsArbiterC(TEST_ARBITER_RESOURCE) as Arbiter;
  enum {
    RESRADIO_ID = unique(TEST_ARBITER_RESOURCE),
    RESBLUE_ID = unique(TEST_ARBITER_RESOURCE),
  };
  App.ResRadio -> Arbiter.Resource[RESRADIO_ID];
  App.ResBlue -> Arbiter.Resource[RESBLUE_ID];
  ...
```

Per la gestione della risorsa bus viene utilizzato un protocollo di tipo First Come First Serve (FCFS), che dà in gestione la risorsa al primo componente che ne fa la richiesta. Il sistema TinyOs mette a disposizione anche un protocollo di Round Robin (RB) per l'accesso alla risorsa. Un aspetto importante da sottolineare è che viene lasciato al programmatore e quindi al programma il compito di rilasciare la risorsa una volta che sono state terminate le operazione di cui si necessita su di essa.

Nello stesso file vengono fatti i collegamenti con le componenti di configurazione e comunicazione del bus, che analizzeremo nella sezione 3.2.2.

```
  ...
  components SerialBluetoothC;
  App.SerialBluetooth -> SerialBluetoothC;
  App.BluetoothControl -> SerialBluetoothC.StdControl;
  ...
```

Implementazione

Questo modulo di configurazione viene implementato nel file BlueTmoteC.nc. Dove possiamo notare due fasi dell'esecuzione del programma, la prima che consente la configurazione di Parani e la seconda che utilizza le funzioni forwarding dei pacchetti radio, ricevuti dalla rete dei nodi disposti nell'ambiente da monitorare, verso lo smartphone. L'invio degli AT-COMMAND permette la configurazione, l'interrogazione e la gestione dei collegamenti bluetooth di Parani, come descritto in [5]. Nello sviluppo della nostra applicazione ci serviremo esclusivamente di tre comandi:

- **AT**: che permette di testare la presenza e il funzionamento del collegamento seriale tra il Tmote e il Parani, la risposta a tale comando è un semplice ok;
- **AT-BTMODE,3**: per cambiare il modo operativo della scheda e consentire la ricezione di richieste di connessione da parte di dispositivi bluetooth;
- **ATZ**: è il comando di reset software, ma che deve essere inviato a Parani dopo l'invio di particolari comandi, come ad esempio i comandi che modificano il modo operativo della scheda, perché i cambiamenti effettuati possano diventare effettivi.

Il frammento di codice seguente evidenzia l'esecuzione di questi comandi.

```
event void BootTimer.fired()
{
  error_t error = SUCCESS;
  if(counter == 1){
    error = call SerialBluetooth.sendCommand(AT);
  }else if(counter == 2){
    error = call SerialBluetooth.sendCommand(BTMODE3);
  }else if(counter == 3){
    error = call SerialBluetooth.sendCommand(ATZ);
  }else if(counter == 4){
    error = call SerialBluetooth.setAttendConn();
  }
...
```

All'invio di ogni comando segue l'attesa della risposta di Parani, la cui ricezione viene segnalata con un evento dal modulo per la comunicazione e fatto scattare il timer per l'invio del comando successivo.

Dopo l'invio dell'ultimo comando che rende effettivo il cambiamento del modo operativo, il programma si mette in attesa della connessione con lo smartphone, e dunque che Parani si connetta allo smartphone e segnali al Tmote che la connessione è stabilita. Da questo momento ogni informazione inviata dal Tmote a Parani viene inoltrata via bluetooth verso lo smartphone e viceversa. Dunque una volta ottenuta la connessione bluetooth parte la scoperta dei nodi della rete e l'invio delle informazioni ottenute allo smartphone.

Un ultimo aspetto da analizzare è la gestione della risorsa bus, questa viene garantita dall'utilizzo dell'interfaccia Resourse di TinyOs [14]. Questa interfaccia collegata al bus UART permette, attraverso il metodo di request, di richiedere il bus per le differenti operazioni da eseguire. Nel nostro caso siamo chiamati a gestire le comunicazioni verso la scheda bluetooth e quelle in direzione del chip radio CC2420. Nel frammento di codice che segue si fa vedere l'utilizzo di due istanze dell'interfaccia Resource di TinyOs per la gestione della Radio e del Bluetooth. Attraverso questa interfaccia è possibile eseguire la request o effettuare il release della risorsa di cui si ha necessità. La chiamata di richiesta viene messa in una coda con le altre richieste e gestita secondo il protocollo FCFS

```
// Resource arbiter
interface Resource as ResRadio;
interface Resource as ResBlue;
...
call ResRadio.request();
call ResBlue.release();
```

Una volta che la risorsa è disponibile viene segnalato un evento granted che ci da la garanzia di un utilizzo esclusivo della risorsa. Al termine dell'utilizzo sarà compito dell'utente di rilasciare la risorsa, e renderla così disponibile ad altri utilizzi.

3.2.2 SerialBluetooth

Questa componente mette a disposizione i metodi per la configurazione del bus e comunicazione con il chip bluetooth. Nel file di configurazione, SerialBluetoothC.nc (appendice B) osserviamo che vengono messe a disposizione due interfacce, StdControl per avviare e fermare le attività che richiedono l'utilizzo del bus, questo in un ottica di programmazione di tipo split operarion, e l'interfaccia SerialBluetooth (mostrata nel frammento di codice che segue) che definisce i metodi relativi alla gestione e comunicazione del bus.

```
configuration SerialBluetoothC
{
  provides interface SerialBluetooth;
  provides interface StdControl;
}
...
```

Mentre di seguito mettiamo in evidenza il wiring con le componenti messe a disposizione dal TinyOs per l'utilizzo del bus con lo standard seriale RS232 [15]. In particolare UartC.UartStream per l'invio e la ricezione delle informazioni come stream di byte, e UartC.Msp430UartConfigure per modificare i parametri del bus e rendere possibile la sincronizzazione tra Tmote e Parani.

```
...
implementation
{
  components new Msp430Uart0C() as UartC;

  SerialBluetoothP.UartStream -> UartC.UartStream;
  SerialBluetoothP.Msp430UartConfigure <- UartC.Msp430UartConfigure;
}
```

Implementazione

Prima cosa vengono settati i parametri del bus con la definizione della struttura dati seguente.

```
  msp430_uart_union_config_t msp430_uart_tmote_config = {
```

```
    {
      utxe : 1,
      urxe : 1,
      ubr : UBR_1MHZ_9600,
      umctl : UMCTL_1MHZ_9600,
      ssel : 0x02,
      pena : 0,
      pev : 0,
      spb : 0,
      clen : 1,
      listen : 0,
      mm : 0,
      ckpl : 0,
      urxse : 0,
      urxeie : 1,
      urxwie : 0
    }
  };
```

I parametri sono impostati secondo le esigenze di Parani per la comunicazione seriale, con riferimento a [5]. Nei campi ubr e umctl vendono impostati la frequenza del clock del Tmote, pari a 1 MHz, e il baud rate con riferimento alle impostazioni di fabbrica di Parani a 9600 bps [5].

Nell'implementazione mettiamo in evidenza due gruppi di funzioni, command e event, distinti. Le prime funzioni per l'invio dei comandi AT-COMMAND, per effettuare il setup della scheda, e le seconde per effettuare il forwarding dei messaggi verso lo smartphone. Questo perché, pur avendo la stessa modalità per l'invio dei messaggi attraverso il bus seriale, richiedono esecuzioni in risposta agli eventi differenti. Per quanto concerne l'invio in entrambi i casi vengo effettuate le stesse procedure, ovvero la configurazione del buffer di ricezione, l'abilitazione dei segnali di interrupt ed infine l'invio del messaggio, come mostrato rispettivamente nelle linee di codice che seguono.

```
call UartStream.receive(rxStrBuffer, rxBufferLength);
resultRXEI = call UartStream.enableReceiveInterrupt();
result = call UartStream.send(txStrBuffer,txBufferLength);
```

Una volta ottenuta la risposta verrà disabilitato l'interrupt, rilasciata la risorsa e segnalato al componente di livello più alto la ricezione di una risposta.

In fase di ricezione, come già spiegato, c'è una gestione differente degli eventi generati, che si tratti di risposte a comandi di setup di Parani o di forwarding verso lo smartphone. Nel caso in cui si stia effettuando il forwarding dei messaggi, si deve porre attenzione ad una particolarità della trasmissione bluetooth descritta nel manuale utente di Parani[5]. Nella trasmissione bluetooth viene descritto un tempo di delay, dovuto alla necessità fisica di Parani di inviare il messaggio ed alla qualità del link di comunicazione. Quindi questo delay può essere variabile e assumere un valore minimo di 30 ms in presenza di un link di comunicazione buono senza interferenze. Per questo motivo è stato necessario introdurre un timer che gestisca questo ritardo e non permetta, a componenti di livello superiore, l'immediato invio di un nuovo messaggio prima che quello precedente sia stato trasmesso. In caso contrario il nuovo messaggio andrebbe a sovrascrivere il buffer di invio nella scheda bluetooth.

Un ulteriore aspetto da sottolineare è relativo alle funzioni messe a disposizione dal TinyOs per l'invio dei byte nel bus di comunicazione UART, in particolare nel momento della ricezione. Il messaggio ricevuto attraverso il bus seriale viene salvato in un buffer impostato dall'utente, come abbiamo visto poc'anzi. Tuttavia deve essere impostata anche la lunghezza di tale buffer e di conseguenza tale lunghezza deve essere corrispondente al messaggio da ricevere. Questa rappresenta una grossa limitazione in quanto si deve sapere in anticipo quando è lungo, in termini di byte, il messaggio che ci aspettiamo di ricevere. Questo aspetto deve essere ben gestito altrimenti c'è la possibilità di perdere delle informazioni. Nel nostro caso le risposte sono state gestite nel modo più semplice considerando le risposte ricevute tutte di una lunghezza prestabilita, senza comunque perdere di efficienza nel programma.

3.3 L'applicazione per Android

L'applicazione per Android è pensata come un'interfaccia per l'utente che deve monitorare la rete di sensori, ed è composta da tre parti fondamentali sviluppate come Activity delle applicazione per Android. Oltre a testare le funzioni di comunicazione bluetooth con il Tmote, questa applicazione permette di impostare

Figura 3.2: Desktop cellulare android.

le coordinate x-y dei nodi della rete rispetto un punto di riferimento stabilito.

3.3.1 BlueTmote.java

La classe BlueTmote.java rappresenta la classe principale di inizializzazione del programma sul dispositivo Android. Si compone di due parti principali, una per configurare l'interfaccia grafica del programma e l'altra per configurare la componente per la gestione del Bluetooth. Per prima cosa vengono inizializzate le componenti grafiche della finestra principale, che è composta da una lista che presenta all'utente i mote scoperti nelle sue vicinanze, e da un menù contestuale (fig. 3.3(a)) che permette di eseguire alcuni semplici comandi come ad esempio la possibilità di effettuare il discover di altri dispositivi bluetooth.

La parte più interessate è rappresentata dalla gestione della componente bluetooth. All'avvio del programma viene controllato se il bluetooth è attivo o meno e in questo caso viene chiesto all'utente di attivarlo per proseguire l'esecuzione. Una volta attivato il dispositivo bluetooth sullo smartphone è possibile gestire le connesioni e trasmettere dei dati. Per questo scopo viene creato un oggetto della classe BluetoothChatService che si occupa della gestione del bluetooth. Al momento della creazione chiede gli venga passato un oggetto di tipo handle con lo scopo di notificare all'applicazione principale i cambiamenti di stato del bluetooth e l'eventuale arrivo di nuovi messaggi. Di seguito si osserva la struttura del

Handler e la diversa elaborazione dei messaggi ricevuti, e la notifica del cambio di stato.

```
private final Handler mHandler = new Handler() {
     @Override
     public void handleMessage(Message msg) {
         switch (msg.what) {
         case MESSAGE_STATE_CHANGE:
             switch (msg.arg1) {
             case BluetoothChatService.STATE_CONNECTED:
               ...
             case BluetoothChatService.STATE_CONNECTING:
               ...
             case BluetoothChatService.STATE_LISTEN:
               ...
             case BluetoothChatService.STATE_NONE:
               ...
             }
             break;
         case MESSAGE_DEVICE_NAME:
           ...
         case MESSAGE_TOAST:
           ...
         }
     }
...
```

All'Handler viene inoltre notificata la ricezione dei messaggi via bluetooth da parte del Tmote, come descritto dal frammento di codice che segue. Alla ricezione del messaggio proveniente dal Tmote viene estrapolato l'id del nodo che è stato scoperto e i valori di potenza di trasmissione, rispettivamente Rssi e Lqi. Per come è stata pensata l'architettura di comunicazione alla ricezione di ogni messaggio lo smartphone risponde con un messaggio. Tale messaggio può essere una semplice conferma, quindi un ok, oppure un messaggio più articolato che è composto dall'id del nodo e la sua posizione (x,y) stabilita dall'utente. Questa tecnica di invio dei messaggi viene attuata perché non c'è la possibilità da parte dello smartphone di

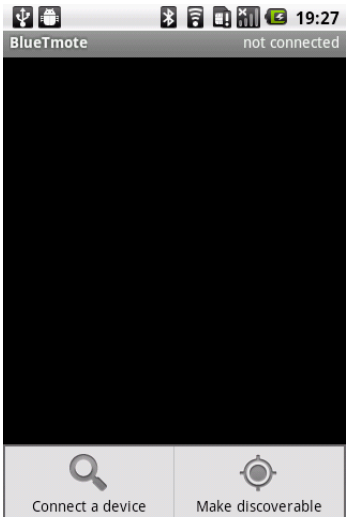
saper quando il Tmote è attivo sul canale Bluetooth, in questo modo appena lo sente attivo gli invia le informazioni che ha a disposizione.

```
case MESSAGE_READ:
  byte[] readBuf = (byte[]) msg.obj;
  String readMessage = new String(readBuf, 0, msg.arg1);
  //Elab. received string
  receivedMessage(readMessage);
  if(send_pos){
    ...
    String send_msg = mote_id + pos_x + pos_y;
    mChatService.write(send_msg.getBytes());
    send_pos = false;
  }else{
    String replay = "**OK********";
    byte[] send = replay.getBytes();
    mChatService.write(send);
  }
break;
```

Alla ricezione di ogni nuovo messaggio viene controllato l'id del nodo, e questo viene memorizzato in un array e visualizzato nell'interfaccia utente, dalla quale si possono interrogare le varie entry per leggerne i parametri e le posizioni dei nodi(fig. 3.4). Nella prossima sezione andremo a vedere l'activity per la visualizzazione dei parametri di ciascun nodo.

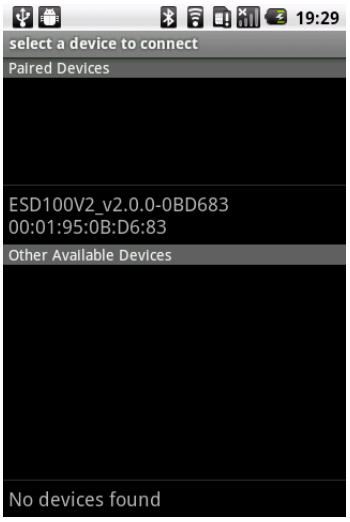
3.3.2 MoteParam.java

Questa classe java permette di visualizzare i parametri di un nodo della rete (fig. 3.4(b)) selezionato dalla lista dei nodi nell'activity principale. Di Interesse in questo programma è l'avvio e lo scambio di parametri tra le activity. Dalla parte dell'activity principale, BlueTmote.java, viene creato un oggetto Intent per lanciare l'esecuzione della nuova interfaccia della classe MoteParam. Prima della sua attivazione vengono memorizzati nell'Intent i valori dei parametri che questa nuova interfaccia deve visualizzare. L'avvio della nuova activity viene effettuato



(a) Menù contestuale

(b) Lista dispositivi bluetooth



(c) Ricerca altri dispositivi

Figura 3.3: Ricerca di dispositivi bluetooth.

attraverso la funzione startActivityForResult che stabilisce anche il messaggio di risposta una volta terminata la sua esecuzione.

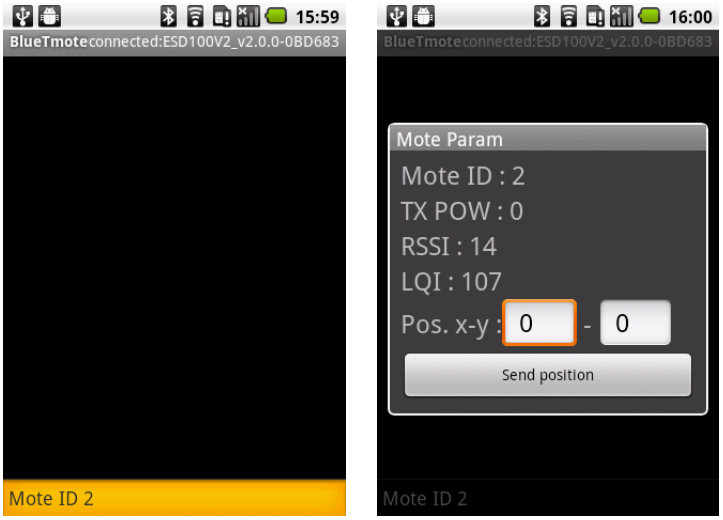
```
 public void onItemClick(AdapterView<?> parent,
                         View view, int position, long id) {
 Intent myIntent = new Intent(view.getContext(), MoteParam.class);
 int index = ((Mote)mConversationView
                         .getItemAtPosition(position)).getNodeId();
  myIntent.putExtra("mote_id",Integer.toString(index));
  myIntent.putExtra("mote_tx_pow",
                         Integer.toString(mote[index].getTxPow()));
  myIntent.putExtra("mote_rssi",
                         Integer.toString(mote[index].getRssi()));
  myIntent.putExtra("mote_lqi",
                         Integer.toString(mote[index].getLqi()));
  myIntent.putExtra("mote_pos_x",
                         Integer.toString(mote[index].getX()));
  myIntent.putExtra("mote_pos_y",
                         Integer.toString(mote[index].getY()));

  startActivityForResult(myIntent, REQUEST_POSITION_MOTE);
...
```

Una volta stabilita la posizione del nodo fisso premendo il pulsante per l'invio del messaggio l'activity corrente crea un nuovo oggetto Intent con i dati da inviare l'activity chiamante, ovvero quella principale, e a questo punto termina la sua esecuzione.

```
public void onClick(View v) {
 // Send a message using content of the edit text widget
 TextView pos_x = (TextView) findViewById(R.id.mote_pos_x);
 TextView pos_y = (TextView) findViewById(R.id.mote_pos_y);

 //send the position
 Intent intent = new Intent();
 intent.putExtra(EXTRA_SEND_POS_X,pos_x.getText().toString().trim());
```



(a) Selezione di un nodo.

(b) Dialog activity con i parametri del nodo selezionato

Figura 3.4: Interfaccia principale.

```
 intent.putExtra(EXTRA_SEND_POS_Y,pos_y.getText().toString().trim());
 intent.putExtra(EXTRA_MOTE_ID, moteID.getText().toString().trim());

 // Set result and finish this Activity
 setResult(Activity.RESULT_OK, intent);
 finish();
}
```

A questo punto una volta chiamata la funzione setResult, l'activity chiamante viene avvisata che il risultato è disponibili per essere elaborato, attraverso l'override della funzione seguente.

```
public void onActivityResult(int requestCode,
                             int resultCode, Intent data){
```

3.3.3 BluetoothChatService.java

Questa classe si occupa della gestione delle connessioni Bluetooth con altri dispositivi. È composta da un thread che rimane in ascolto per le connessioni in ingresso, un thread per stabilire la connessione con un'altro dispositivo bluetooth,

ed un ultimo thread che si occupa di gestire le trasmissioni, quando lo smartphone è già connesso con altri dispositivi Bluetooth. Presentiamo brevemente i thread di seguito:

- AcceptThread. Finché il dispositivo bluetooth non è connesso il codice controlla che ci siano delle richieste di connessione, in caso affermativo la procedure controlla in modo sincronizzato lo stato attuale del bluetooth individuando due casi sensibili da gestire. Il primo che il bluetooth sia nella condizione di poter accettare la connessione e di conseguenza avviare il metodo per la connessione. Il secondo caso che il bluetooth sia già connesso con un'altro dispositivo, dunque si ritiene che la nuova richiesta di connessione sia da ignorare e chiudere il socket.

  ```
  private class AcceptThread extends Thread { ... }
  ```

- ConnectThread. Una volta ricevuta la richiesta di connessione o creato il socket, viene attivato questo thread per collegare il socket di comunicazione. Se la connessione fallisce il socket viene chiuso e viene fatto ripartire in thread, descritto in precedenza, in attesa di una nuova connessione. Mentre se la connessione del socket ha successo il thread viene reinizializzato, il dispositivo si trova nello stato connected e parte il thread successivo.

  ```
  private class ConnectThread extends Thread { ... }
  ```

- ConnectedThread. Una volta connesso il socket si passa a gestire il flusso di dati proveniente dal dispositivo a cui si è connessi. Vengo dunque inizializzati i flussi di ingresso e uscita del socket e il thread legge il flusso in ingresso, se arrivano nuovi messaggi questi vengono notificati dall'handler alla finestra principale. Inoltre il thread fornisce un metodo non sincronizzato per l'invio dei messaggi.

  ```
  private class ConnectedThread extends Thread { ... }
  ```

Come mostrato in figura 3.3, una volta premuto il pulsante per la ricerca di nuovi dispositivi bluetooth appare una nuova schermata che distingue i dispositivi già accoppiati dai nuovi dispositivi scoperti. È possibile così selezionare il dispositivo desiderato ed effettuare il collegamento.

3.4 Problematiche da risolvere

Il programma così implementato funziona e risponde alle richieste attese, tuttavia presenta alcuni aspetti problematici che richiedono particolare attenzione. Questi aspetti toccano aspetti come la scalabilità e il fault tollerance del sistema. Si evidenziano tre problematiche principali:

- l'utilizzo del bus seriale da parte del Tmote;
- la risposta della rete ai ping del sistema centrale;
- l'invio dei messaggi via bluetooth da parte di Parani.

Il primo aspetto, relativo alla gestione del bus da parte del Tmote, è probabilmente l'aspetto più complesso da trattare e riguarda la difficoltà di gestire le comunicazioni. Come già osservato l'architettura del Tmote è sviluppata attorno alla condivisione del bus USART0 al quale sono collegati degli ingressi per l'acquisizione di dati da sensori esterni, un bus I2C, il bus UART che andiamo ad utilizzare e, molto importante, è collegato pure il bus di collegamento con il chip CC2420 per le comunicazioni con gli altri mote. In questo contesto il software deve poter collegare e impostare i parametri per le diverse comunicazioni al momento opportuno. Questo è reso possibile con l'utilizzo delle interfacce e componenti resi disponibili dal TinyOs. Nel nostro caso il problema si verifica quando l'utente, in possesso dello smartphone, volesse comunicare con il Tmote. In questa situazione ci si trova nella difficoltà di sapere se il bus è abilitato per la comunicazione seriale oppure no. Ricordiamo inoltre che in questo schema di collegamento una volta che il Parani è impostato nella modalità 3 di funzionamento tutti i messaggi che riceve vengono inoltrati al destinatario. Nel caso in cui l'utente invii dei messaggi mentre il bus non è abilitato questi andrebbero persi, senza aver la possibilità di recuperarli. È plausibile tuttavia pensare ad un sistema di ACK per verificare la disponibilità del bus. Ma anche in questo caso il messaggio di ACK inviato dallo smartphone al mote potrebbe andare perso se in bus non è abilitato. Dal lato dello smartphone andremmo a inviare degli ACK finché il bus non viene abilitato dal Tmote, ma nel momento in cui questo avviene significa che già il mote ha delle informazioni da inviare allo smartphone. Questo spiega la soluzione adottata in questo progetto, per la quale se l'utente dello smartphone ha dei messaggi da inviare al Tmote questi vengono inviati solo

quando si ricevono dei messaggi dallo stesso, perché in questo modo si è consapevoli che il Tmote sta utilizzando il bus ed è in grado di inviare e ricevere. Questa soluzione tuttavia risulta di scarsa efficienza perché limita le possibilità di operare dell'utente dello smartphone e potrebbe introdurre dei ritardi nelle comunicazioni con gli altri nodi fissi della rete, per esempio nel caso in cui l'utente si sia spostato e il nodo fisso della rete diventi irraggiungibile. Una possibile soluzione è una gestione di tipo time division del bus in cui il bus divenga disponibile per un certo intervallo di tempo, questo caso potrebbe essere utile solo nella fase di labeling dei nodi fissi. Facendo così il Tmote non sarebbe più in grado di ascoltare i messaggi provenienti dai nodi fissi della rete, perché per utilizzare il bus deve disabilitare l'utilizzo della radio.

Il secondo problema è strettamente connesso a quello appena descritto, in quanto la risposta dei nodi della rete al ping da parte del nodo collegato allo smartphone è legata alla capacità dello stesso a gestire il bus seriale. Infatti il Tmote riesce a comunicare in modo separato o attraverso il canale bluetooth o attraverso il canale radio. Nel caso in cui arrivino dei messaggi da parte dei nodi della rete durante la comunicazione bluetooth questi non potranno essere rilevati e di conseguenza potrebbero essere perduti. Il software è stato sviluppato in modo tale che quando al Tmote arrivi un messaggio da parte dei nodi della rete questo venga immediatamente inoltrato via bluetooth. Tuttavia questa tecnica presenta degli svantaggi, nel senso che se più nodi rispondono al ping nello stesso momento o in un lasso di tempo molto breve comunque si verifica una perdita di informazioni. Si potrebbe pensare a una soluzione che prevede il salvataggio delle risposte ai ping in una tabella e successivamente il suo invio allo smartphone. Anche in questo modo c'è da pensare e prevedere quale dimensione deve avere la tabella nel programma Nesc, in quanto non prevede l'utilizzo di allocazione dinamica della memoria.

In fine un'aspetto al quale porre attenzione è il sistema di comunicazione di Parani. Nei test di comunicazione tra Tmote e smartphone si è osservato un comportamento strano nella ricezione dei messaggi provenienti dal chip bluetooth. Il messaggio inviato da Parani verso lo smartphone in qualche caso viene ricevuto in più parti, generalmente due. Questa situazione è facilmente risolvibile ponendo un flag di fine messaggio e ricomponendolo al termine della ricezione. Tuttavia questo comportamento aleatorio non trova spiegazione nella documentazione del

dispositivo, pertanto si può solo ipotizzare che sia dovuta a comportamenti combinati da parte del Tmote e Parani. Più precisamente si pensa che il Tmote invii il messaggio verso Parani e che in qualche caso la trasmissione venga interrotta da altri interrupt da parte del sistema TinyOs provocando il parziale riempimento del buffer di Parani. Nel frattempo Parani riceve segnalazione che il buffer si sta riempiendo e comincia la trasmissione dei dati verso lo smartphone. Con questa ipotesi si pensa che le interruzioni nello stream possano provocare l'invio parziale del messaggio, che in realtà deve essere ancora terminato da parte del Tmote.

Capitolo 4

Localizzazione

In questo capitolo verranno presentati alcuni progetti di localizzazione presenti in letteratura. Tra le innumerevoli tecniche sviluppate negli ultimi anni si è scelto di descrivere solo le tecniche che più delle altre si sono distinte per un approccio originale e un elevato grado di accuratezza per risolvere il problema della localizzazione.

Al termine di questa breve presentazione verrà proposto una tecnica di localizzazione gready implementata sullo smartphone Android, come esempio per testare la fattibilità del progetto.

4.1 Classificazione delle tecniche di localizzazione

Le tecniche di localizzazione possono essere classificate in diversi modi, possono essere suddivise in base alle informazioni a priori per la localizzazione dei nodi, come ad esempio l'utilizzo o meno di nodi àncora, nodi di cui è nota la posizione e utilizzati per determinare la posizione di nodi rimanenti. Differenti classificazioni possono essere effettuate in base alle ipotesi a priori sulla propagazione nel mezzo e della grandezza fisica utilizzata, nel caso di onde elettromagnetiche si può supporre un decadimento quadratico della potenza di trasmissione in funzione della distanza, oppure si può apprendere il modello che meglio descrive la propagazione del campo direttamente dalle misure effettuate. Altre classificazioni possono essere fatte sulla base dell'architettura dell'algoritmo utilizzato.

Un'approccio molto diffuso per la classificazione delle tecniche di localizzazione è effettuato in base ai parametri utilizzati per ottenere la posizione dei nodi, come ad esempio la misura del tempo di volo (Time of Arrival, ToA), la differenza del tempo di volo (Time Difference of Arrival, TDoA, l'intervallo che intercorre tra la ricezione del segnale radio e un emettitore di ultrasuoni), l'angolo di arrivo (Angle of Arrival, AoA), la potenza del segnale ricevuto (Received Signal Strength Indicator, RSSI). I metodi sviluppati nelle prime tre tecniche di localizzazione richiedono tuttavia dispositivi hardware dedicati come antenne direzionali, emettitori di ultrasuoni, sistemi per la sincronizzazione dei nodi, magnetometri, infrarossi o videocamere. Questo tipo di sistemi sono tuttavia poco utilizzati perché sfruttano un hardware dispendioso e molto sensibile.

Sistemi basati sulla lettura del RSSI sono invece più diffusi perché hanno costi più contenuti in quanto ciascun nodi di una WSN monta un antenna per le comunicazioni dalla quale è possibile effettuare questa lettura. Tuttavia questi sistemi, in un ambiente indoor, soffrono di problemi di riflessione, diffrazione, interferenza che vanno ad alterare la lettura del RSSI rendendola poco affidabile.

Molti algoritmi che utilizzano RSSI possono essere raggruppati in due differenti classi:

- RSSI-map based, questi algoritmi generano una mappa del valore del RSSI per ogni nodo àncora, vengono in oltre incluse nella mappa la morfologia dell'ambiente compresi i muri. Una volta calcolata la mappa off-line, la stima della posizione viene effettuata on-line cercando il valore più vicino alla misura rilevata.

- RSS-channel model based, il vantaggio di questi algoritmi è quello di non richiedere nessuna informazione a priori dell'ambiente. Dunque grazie ad un modello del canale cercano di calcolare la distanza del nodo mobile dai nodi àncora e successivamente di triangolare la posizione.

4.2 Received Signal Strength Indicator, RSSI

Il chip radio CC2420 [4] presente sul Tmote Sky, è in grado di fornire, fra diversi indici, anche il valore di RSSI, Received Signal Strength Indicator, che significa letteralmente Indicatore di Forza del Segnale Ricevuto. L'RSSI è un numero che

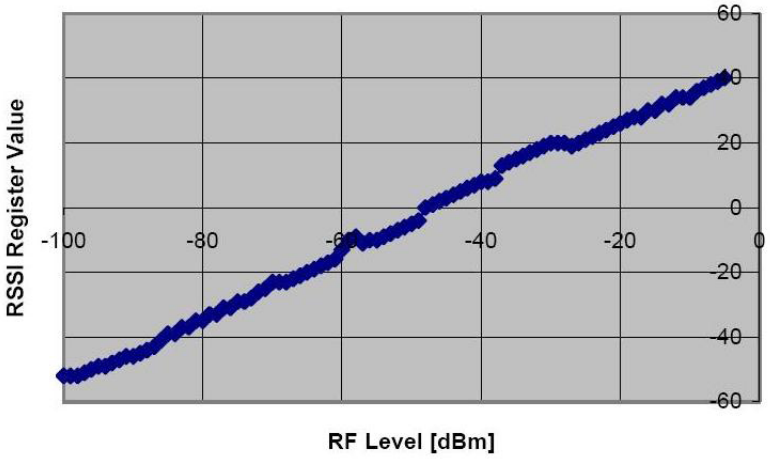


Figura 4.1: Andamento del RSSI rispetto alla potenza del campo elettromagnetico.

la radio ci fornisce in un registro di 8 bit in complemento a 2 (RSSI.RSSI VAL) che è legato alla potenza del campo elettromagnetico (RF level) ossia alla forza del segnale ascoltato, da cui il nome RSS. L'RSS può essere ricavato direttamente dal valore del registro RSSI.RSSI VAL mediante l'utilizzo della seguente equazione:

$$P_{RF} = RSS = RSSI\_VAL + RSSI\_OFFSET \quad [dBm] \tag{4.1}$$

dove RSSI OFFSET è un valore, trovato empiricamente durante lo sviluppo del sistema radio e dato quindi dal costruttore, di circa -45 dBm. L'andamento generale dell'RSSI VAL in funzione della potenza del segnale ricevuto è riportato in figura 4.1. Il valore di RSSI nel registro RSSI.RSSI VAL viene continuamente calcolato ed aggiornato dal chip radio ad ogni arrivo di un nuovo pacchetto di dati.

Legato al valore di RSSI anche il valore di Link Quality Indicator (LQI) è ritornato direttamente dal chip radio il quale lo genera nel blocco di sincronizzazione e di correlazione dei dati. Il valore di LQI rappresenta dunque la facilità col quale è stato possibile associare la sequenza di simboli ricevuti ad una parola di quattro bit e quindi, dipende in una certa misura dalla quantità di rumore presente nel canale nella banda in cui avviene la trasmissione.

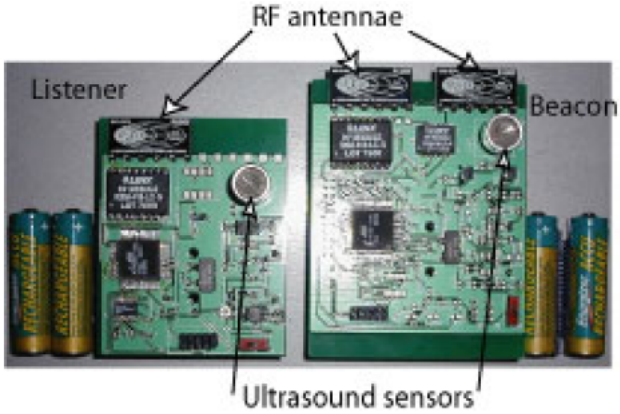


Figura 4.2: Esempio di nodi utilizzati dal sistema radar.

4.3 Esempi di sistemi di localizzazione

4.3.1 Cricket

Il sistema di localizzazione Cricket [16] [17], è nato dal progetto Oxygen del MIT. Scopo del progetto era la realizzazione di un sistema di localizzazione completo, a basso costo e con un elevato grado di robustezza in funzione dello spegnimento dei nodi àncora della rete.

La tecnica di localizzazione di un nodo mobile sfrutta la differenza tra il tempo del segnale elettromagnetico ed il segnale ultrasonico, con una precisione compresa nell'ordine del centimetro. Tuttavia una soluzione di questo tipo è poco utilizzata, perché i componenti utilizzati per generare il segnali ultrasonici sono molto sensibili alle vibrazioni meccaniche e soggetti a frequenti rotture. Un esempio di nodo e riportato in figura 4.2.

4.3.2 Radar

Il progetto RADAR [18] [19] è uno dei primi a non utilizzare nodi dedicati per realizzare un sistema di localizzazione. Si concentra sulla localizzazione di schede WiFi e l'utilizzo come nodi àncora degli Access Point (AP) della rete. In questo contesto le trasmissioni radio sono sensibilmente differenti da quelle rilevate in una WSN, perciò il confronto non è immediato.

Il sistema sfrutta la potenza del segnale ricevuto per ottenere informazioni utili alla localizzazione del nodo mobile. Dalle prove effettuare venne osservato

che il valore non dipendeva solo dalla distanza del nodo ma anche dall'orientazione dell'antenna. Il sistema utilizza l'equazione 4.2, che lega la potenza del campo elettromagnetico $P(d)$ alla distanza d e al numero di muri C compresi tra il nodo mobile e i nodi àncora.

$$P(d) = P(d_0) - 10n_p \log\Big(\frac{d}{d_0}\Big) - C \cdot WAF \tag{4.2}$$

Con $P(d_0)$ si indica il valore della potenza del segnale elettromagnetico rilevata ad una distanza d_0, n_p il fattore di decadimento del segnale elettromagnetico, mentre WAF indica l'attenuazione del segnale causata dal passaggio attraverso i muri.

4.3.3 MoteTrack

MoteTrack [20] [21] è un sistema di localizzazione decentralizzato sviluppato dall'Università di Harvard. Il sistema non utilizza alcun tipo di modello che leghi la potenza del segnale RSS ricevuto ad un qualche valore di distanza. Si tratta di un sistema di localizzazione decentralizzato che si basa sulla misura della potenza del segnale elettromagnetico prodotto dai nodi àncora. La posizione del nodo mobile viene calcolato sulla base di una mappa creata dalla raccolta di misure svolte a priori sul campo.

L'insieme dei valori di RSS rilevati da un nodo mobile, unitamente all'identificativo (ID) del nodo àncora che ha generato il campo elettromagnetico a cui si riferiscono i dati, è detta signature. L'insieme di una signature e delle coordinate spaziali in cui essa è stata raccolta è detto invece reference signature. L'insieme delle reference signature raccolte sono memorizzate fra i nodi della rete in modo da minimizzare l'occupazione di memoria per il singolo nodo e contemporaneamente massimizzare la ridondanza dei dati. Durante la fase di localizzazione, un nodo mobile M, dopo aver raccolto una propria signature, utilizza le reference signature memorizzate nei nodi àncora da esso rilevabili per stimare la propria posizione. La posizione del nodo è dunque determinata attraverso una media pesata delle distanze dalle coordinate contenute nelle reference signature utilizzate e, i pesi utilizzati nella media, sono proporzionali alle distanze fra la signature rilevata dal nodo M e quelle contenute nelle reference signature. La tecnica di localizzazione del progetto MoteTrack suppone dunque che due signature siano tanto più simili fra loro quanto più vicine sono le coordinate in cui queste sono

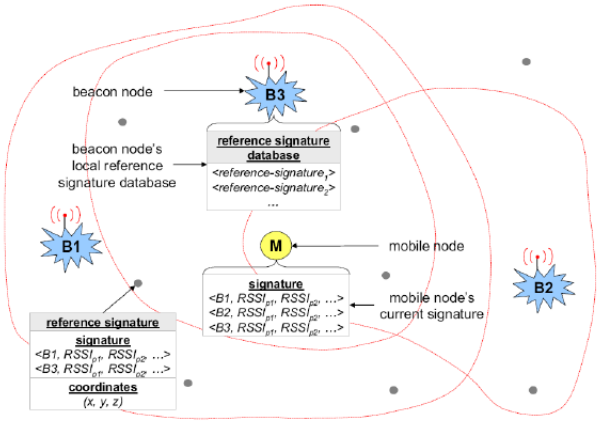


Figura 4.3: Esempio di raccolta di signature di un nodo mobile M. I nodi àncora sono indicati come nodi beacon.

state raccolte. In figura 4.3 è riportato un esempio di raccolta di una signature da parte di un nodo mobile M e l'invio di alcune reference signature da parte dei nodi àncora.

4.3.4 Interferometrico

Il sistema interferometrico utilizza come grandezza fisica per la localizzazione la fase di un segnale elettromagnetico di che si genera dall'interferenza di due segnali sinusoidali emessi da alcuni nodi àncora. Questa soluzione è proposta dall'Università di Vanderbilt con il nome Radio Interferometric Positioning System (RIPS) viene descritto in [22]. La localizzazione effettuata su vaste aree privi di ostacoli come campi da calcio ottengono errori inferiori a 3 cm.

L'idea dei ricercatori della Vanderbilt è stata dunque di utilizzare il fenomeno del battimento di due segnali sinusoidali ad alta frequenza per ottenere un segnale d'interferenza a bassa frequenza di cui fosse possibile misurare la fase anche con l'hardware a basso costo presente nei nodi. La localizzazione dei nodi della rete avviene quindi, come illustrato in figura 4.4, attraverso il confronto delle fasi misurate dai singoli nodi. Il fenomeno del battimento, attraverso cui è generata un'onda a bassa frequenza, partendo dall'interferenza di due sinusoidi ad alta frequenza.

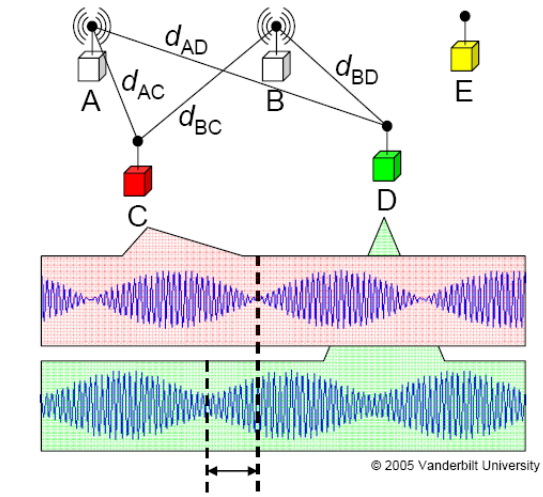


Figura 4.4: Principio di funzionamento del sistema di localizzazione interferometrico.

4.3.5 ARIANNA e TESEO

Recentemente sono stati proposti due nuovi algoritmi per la localizzazione presso l'Università di Padova denominati come ARIANNA e TESEO [23]. Il primo, ARIANNA, si sviluppa attorno alla costruzione di una mappa a priori dell'ambiente, mentre il secondo, TESEO, rientra nella categoria di algoritmi basati sul modello di campo elettromagnetico.

ARIANNA si basa sulla lettura del RSSI da parte del nodo mobile rispetto i nodi àncora, per creare una mappa a priori della distribuzione del RSSI nell'area di interesse. Il nodo mobile viene modellato come un processo Markoviano omogeneo del primo ordine, in modo tale da poter permettere un filtraggio Bayesiano per stimare e localizzare la posizione del nodo mobile.

TESEO adotta un modello del canale radio per legare il valore del RSS con la distanza del nodo mobile, espresso in (4.3).

$$P(d) = P(d_0) - 10n_p \log\left(\frac{d}{d_0}\right) + \chi_\sigma \tag{4.3}$$

Dove $P(d)$ è la potenza ricevuta (RSS) in [dBm], $P(d_0)$ è la potenza ricevuta a una determinata distanza d_0, n_p è il fattore di path-loss, mentre χ_σ è una variabile aleatoria gaussiana di media nulla e varianza σ^2.

L'algoritmo può essere costruito sia in modo centralizzato che decentralizzato per risolvere problemi di scalabilità del sistema. La localizzazione del modo mobile è composta, nella sua parte principale, dalla stima della posizione attraverso uno stimatore a massima verosimiglianza e una seconda parte di recovery, che entra in gioco quando la connessione viene interrotta, e adotta uno stimatore ai minimi quadrati.

Più precisamente ogni nodo àncora riceve un certo numero di messaggi in broadcast dal nodo mobile e se questi superano una certa soglia di potenza P_{th} e LQI_{th} vengono salvati. In base a questi valori ciascun nodo àncora $j = 2, ..., N$ calcola la potenza media $\breve{P}_{1,j}$ e dunque la distanza dal nodo mobile $\breve{d}_{1,j}$ attraverso lo stimatore a massima verosimiglianza. Si viene a creare una cella di nodi vicini al nodo mobile descritta come segue:

$$A_s = \{j > 1 | P_{1,j} > P_{th}, \breve{d}_{1,j} < Range, LQI_{1,j} > LQI_{th}\} \tag{4.4}$$

dove Range indica la distanza massima dal nodo mobile. Infine tutti nodi in (4.4) inviano la distanza calcolata al nodo mobile che effettua il calcolo della posizione.

4.4 Implementazione sullo smartphone

Presentiamo di seguito una semplice tecnica di localizzazione ideata per testare la fattibilità dell'utilizzo dello smartphone come strumento per il calcolo e la presentazione della posizione del nodo mobile. Il codice scritto viene presentato in appendice C.

La soluzione adottata si basa sul modello del campo (4.3) già adottato nell'articolo [23] e presentato nel paragrafo 4.3.5. Dove si fa notare che $P(d)$ è una variabile gaussiana con media $\bar{P}(d) = P(d_0) - 10n_p \log(d/d_0)$ e varianza σ^2 ed è possibile una sua riformulazione come segue:

$$P(d) = P_{TX} + A - 10n_p \log(d) + \chi_\sigma \tag{4.5}$$

dove P_{TX} è una potenza di trasmissione ben conosciuta (well-know transmission power) e A è un fattore di attenuazione. È possibile quindi calcolare la distanza $d_{1,j}$ tra nodo mobile (indicato con indice 1) e nodo fisso $j = 2, ..., n$:

$$d_{1,j} = e^{-\frac{\gamma}{2}} 10^{\left(\frac{A-P(d)}{10n_p}\right)} \tag{4.6}$$

dove $e^{\frac{\gamma}{2}}$ è il fattore che interviene nel calcolo della media della variabile aleatoria $d(P)$ [24].

$$\mathbb{E}[d(P)] = de^{\frac{\gamma}{2}}; \quad \gamma = \left(\frac{\sigma \ln 10}{10n_p}\right)^2 \tag{4.7}$$

Con queste informazioni possiamo presentare l'algoritmo, che esegue alla fine il calcolo della posizione del nodo mobile come la media delle coordinate x e y dei nodi àncora pesata sulle distanze degli stessi nodi àncora dal nodo mobile.

L'algoritmo segue i seguenti passi:

1. inizializzazione dei nodi àncora con le loro posizioni (x,y) rispetto ad un punto di riferimento stabilito a priori;

2. il nodo mobile, una volta connesso allo smartphone, manda dei messaggi di ping in broadcast ai nodi àncora, permettendo a quest'ultimi di ricevere il messaggio nello stesso istante;

3. i nodi àncora rilevano il valore di RSSI dal messaggio ricevuto e lo inoltrano in risposta al ping verso il nodo mobile con le seguenti informazioni:

seq_no	tx_pow	RSSI	LQI	x	y

   dove seq_no è il sequence number del ping inviato dal nodo mobile, tx_power è il livello di potenza utilizzato secondo la tabella 2.1, il valore LQI e la posizione x e y del nodo;

4. una volta ricevuto il pacchetto di risposta al ping e lette le informazioni in esso contenuto, il nodo mobile inoltra, utilizzando il protocollo bluetooth, tali informazioni allo smartphone, secondo quanto descritto nel capitolo 3.

5. nello smartphone i valori vengo salvati in un array, e si procede con il calcolo della posizione (x_1, y_1) del nodo mobile, da prima stimando la distanza dei nodi àncora $j = 2, ..., n$ secondo la (4.6) e successivamente calcolando la media pesata delle coordinate x_j e y_j come:

$$x_1 = \frac{\sum_j d_{1,j}^2 x_j}{\sum_j d_{1,j}^2} \quad y_1 = \frac{\sum_j d_{1,j}^2 y_j}{\sum_j d_{1,j}^2}; \tag{4.8}$$

6. al termine del calcolo la posizione ottenuta viene visualizzata sullo schermo dello smartphone. In figura 4.5 viene riportato un esempio di visualizzazione

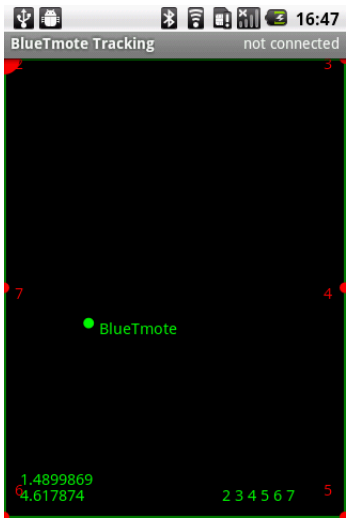


Figura 4.5: Presentazione della posizione del nodo mobile sullo smartphone.

del programma di localizzazione sullo smartphone rispetto ad uno dei test effettuati.

In tabella 4.1 e in figura 4.6 vengo riportati alcuni dei valori di posizione reale e posizione calcolata con il rispettivo errore di alcuni test condotti. Sono stati utilizzati sei nodi disposti lungo il perimetro di una stanza di 6x8 mq. L'algoritmo sfrutta i parametri già presentati in [23], ovvero $n_p = 2.12$, $A = -63.67$ e $\sigma = 7.57[dBm]$, mentre non viene imposta nessuna soglia e non viene fatta nessuna media sul valore di RSSI. Si osserva che tuttavia l'algoritmo non è molto preciso, questo perché risente in modo sostanziale dell'instabilità del valore di RSSI rilevato in fase di ricezione. Instabilità dovuta a riflessioni, diffrazioni e scattering che il segnale subisce in ambienti indoor, ma anche dall'errore sistematico nella lettura della misura. Doveroso osservare che possono interferire anche le comunicazioni attraverso il protocollo bluetooth e quelle dovute all'antenna dello smartphone. Inoltre da alcuni test condotti si è ottenuto un risultato differente a seconda della disposizione dei nodi àncora nell'ambiente. Ad esempio nel caso in cui i nodi àncora siano disposti solo lungo il perimetro, oppure che siano posizionati anche all'interno dell'area di interesse. Purtroppo non si è potuto analizzare più precisamente tale comportamento.

Posizione Reale [m]	Posizione calcolata [m]	Errore [m]
(1,6; 1,98)	$(3,13 \cdot 10^{-6}; 3,99)$	2,57
(3,2; 1,98)	(1,58; 3,93)	2,54
(4,8; 1,98)	(5,73; 3,52)	1,8
(4,8; 3,96)	(5,99; 4)	1,19
(3,2; 3,96)	(0,39; 2,14)	3,35
(1,6; 3,96)	(2,63; 3,99)	1,03
(1,6; 5,94)	(1,29; 7,13)	1,23
(3,2; 5,94)	(2,52; 6,3)	0,77
(4,8; 5,94)	(5,88; 7,46)	1,86

Tabella 4.1: Posizione calcolata rispetto alla posizione reale con rispettivo errore.

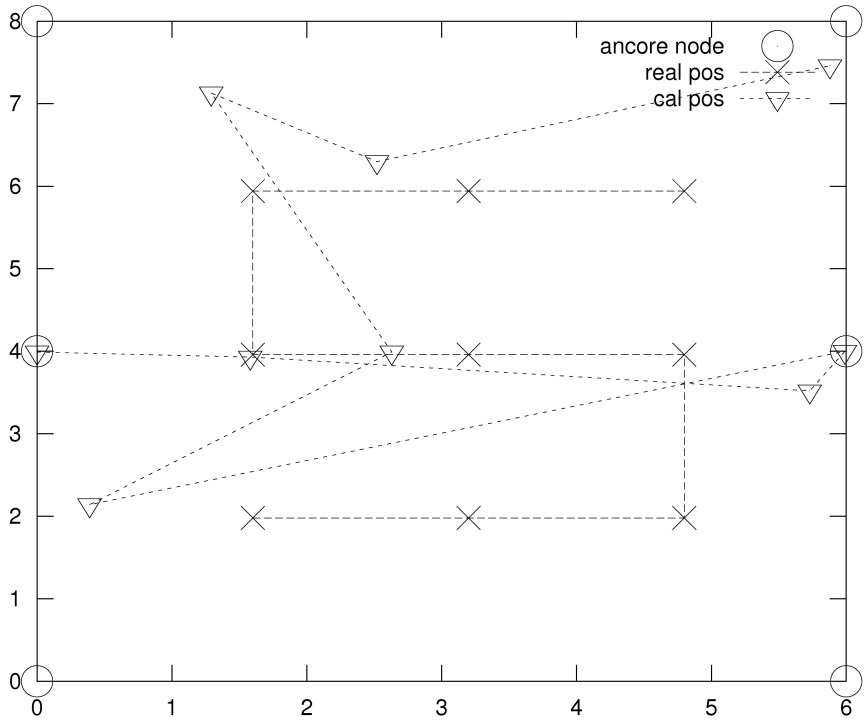


Figura 4.6: Esempio di di localizzazione relativo ai dati riportati in tabella 4.1.

Capitolo 5

Conclusioni

Nella tesi si è presentato un sistema per la navigazione indoor per palmare tramite una rete di sensori wireless. Il progetto è stato sviluppato in due parti, la prima relativa l'interfacciamento del palmare ad un nodo della rete attraverso una connessione bluetooth, mentre la seconda riguardante la realizzazione di un semplice sistema di localizzazione del nodo mobile collegato al palmare.

La prima parte è giustificata dal fatto che ancora sul mercato non sono presenti palmari con integrato il protocollo di comunicazione IEEE 802.15.4, è stato necessario quindi collegare un nodo della rete ad una scheda bluetooth per interfacciare la rete al palmare. Nella prima parte si sono dovute affrontare diverse difficoltà come ad esempio la mancanza di documentazione relativa alle caratteristiche della porta seriale del Tmote-sky, che non rispetta lo standard di idle high sul pin di trasmissione cosa che invece viene mantenuta dalla scheda bluetooth Parani. Il collegamento effettuato risulta soddisfacente, rimangono tuttavia alcuni aspetti su cui porre attenzione, come la condivisione del bus seriale da parte del Tmote che può portare a perdita di messaggi, in quanto l'utilizzo del bus per le comunicazioni bluetooth esclude la ricezione di messaggi dal resto della rete e viceversa. Mentre dal lato dello smartphone la ricezione dei messaggi risulta discontinua, infatti in certi casi arrivano per intero in altri spezzati in due o più parti.

Tuttavia vi siano alcuni problemi nelle comunicazioni, il sistema così come è stato implementato permette la ricezione delle informazioni necessarie per effettuare l'operazione di localizzazione descritta nella seconda parte della tesi. Dopo una breve presentazione dei parametri RSSI e LQI e alcuni esempi di sistemi di

localizzazione presenti in letteratura viene descritta una tecnica intuitiva per il calcolo della posizione del nodo mobile. Tecnica basata sulla media pesata delle coordinate dei nodi àncora rispetto alle distanze degli stessi dal nodo mobile. In questa parte conclusiva a completamento del progetto è stato presentato l'algoritmo implementato per la localizzazione e i risultati ottenuti. Questa tecnica presenta due vantaggi fondamentali, ovvero non richiede informazioni a priori dell'ambiente e riduce drasticamente la mole di calcoli da effettuare

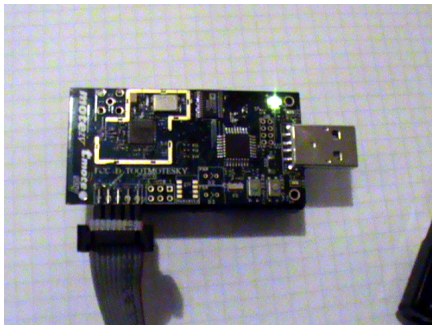
Dai test effettuati si osserva che i valori di posizione ottenuti con questa tecnica risentono della variabilità del RSSI dovuto ai fenomeni di riflessione, diffrazione e scattering. Un miglioramento potrebbe essere introdotto correggendo il valore di RSSI tramite algoritmi di consensus come proposto nella tesi [25]. I Risultati tuttavia sono da considerarsi indicativi e utili per stabilire la fattibilità del'utilizzo di un algoritmo di localizzazione su di un palmare dotato di risorse limitate. Aspetto quest'ultimo che porterà in futuro alla realizzazione di un sistema di localizzazione di dimensioni e con caratteristiche adatte alla commercializzazione. Si fa notare infatti che fino ad oggi questo tipo di sistemi di localizzazione non sono ancora stati commercializzati anche se le reti di sensori sono ampiamente utilizzate già da diversi anni nel monitoraggio ambientale, industriale e nell'ambito della domotica.

In conclusione il progetto svolto deve essere migliorato sotto il punto di vista delle comunicazioni, soprattutto nell'uso del bus e della comunicazione bluetooth, mentre l'algoritmo di localizzazione proposto è ancora poco preciso e richiede un'ulteriore studio.

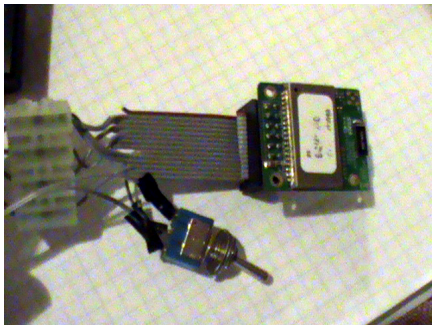
Appendice A

BlueTmote

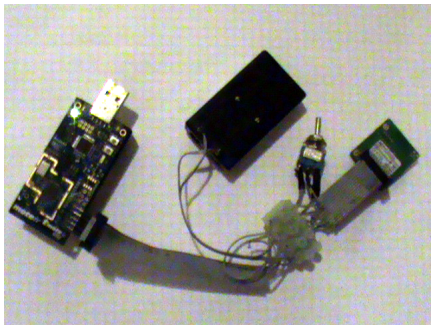
Di seguito in figura A.1 vengono mostrate alcune foto del collegamento tra Tmote-Sky(fig.A.1(a)) e Parani(fig.A.1(b)) a formare quello che abbiamo chiamato all'interno di questa tesi come BlueTmote (fig.A.1(c)).



(a) Tmote-Sky



(b) Parani



(c) BlueTmote

Figura A.1: Foto del collegamento ottenuto.

Appendice B

Comunicazione seriale

Proponiamo di seguito il programma sviluppato per la comunicazione seriale tra Tmote-sky e Parani scritto in Nesc per TinyOs.

BlueTmoteAppC.nc

```
#define TEST_ARBITER_RESOURCE   "Test.Arbiter.Resource"

configuration BlueTmoteAppC {}
implementation
{
  components BlueTmoteC as App, LedsC, MainC;
  components SerialBluetoothC;
  components new TimerMilliC() as BootTimer;
  components new TimerMilliC() as MilliTimer;

  App.Boot -> MainC.Boot;
  App.SerialBluetooth -> SerialBluetoothC;
  App.Leds -> LedsC;
  App.BootTimer -> BootTimer;
  App.MilliTimer -> MilliTimer;

  App.BluetoothControl -> SerialBluetoothC.StdControl;

  // for communication; sending
```

```
  components ActiveMessageC;
  components new AMSenderC(AM_RADIOMSG);
  App.Packet -> AMSenderC;
  App.AMPacket -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.AMControl -> ActiveMessageC;

  // for receiving
  components new AMReceiverC(AM_RADIOMSG);
  App.Receive -> AMReceiverC;

  // Resource arbiter
  components new FcfsArbiterC(TEST_ARBITER_RESOURCE) as Arbiter;
  enum {
    RESRADIO_ID = unique(TEST_ARBITER_RESOURCE),
    RESBLUE_ID = unique(TEST_ARBITER_RESOURCE),
  };
  App.ResRadio -> Arbiter.Resource[RESRADIO_ID];
  App.ResBlue -> Arbiter.Resource[RESBLUE_ID];

}
```

BlueTmoteC.nc

```
#include "Timer.h"
#include "SerialBluetooth.h"
#include "printf.h"
module BlueTmoteC
{
  uses
  {
    interface Leds; //led0 = red, led1 = green, led2 = blu
    interface Boot;
    interface SerialBluetooth;
    interface Timer<TMilli> as BootTimer;
    interface Timer<TMilli> as MilliTimer;
```

```
    // LCD screen
    interface StdControl as BluetoothControl;

    // for communication: sending
    interface Packet;
    interface AMPacket;
    interface AMSend;
    interface SplitControl as AMControl;
    // for receiving
    interface Receive;

    // Resource arbiter
    interface Resource as ResRadio;
    interface Resource as ResBlue;
  }
}
implementation
{
  uint8_t counter = 1;
  bool busy = FALSE;
  message_t pkt;
  uint16_t seqno = 0;
  uint8_t send_node_id;
  uint8_t send_x;
  uint8_t send_y;
  bool send_pos = FALSE;
  uint8_t countRx = 0;
  uint8_t rxRssi[6];
  uint16_t sumRssi = 0;
  uint16_t totRssi = 0;

  event void Boot.booted()
  {
```

```
    TOSH_MAKE_UTXD0_OUTPUT();
    TOSH_SET_UTXD0_PIN();
    call ResBlue.request();
    call BootTimer.startOneShot(BOOT_SHOT_TIME);
  }
/***
 * resources
 ***/
  event void ResBlue.granted()
  {
    call BluetoothControl.start();
    call AMControl.stop();
  }

  event void ResRadio.granted()
  {
    call BluetoothControl.stop();
    call AMControl.start();
  }

  event void AMControl.stopDone(error_t err) {
  }

  event void AMControl.startDone(error_t err) {
    if(err==SUCCESS) {
    }
    else {
      call AMControl.start();
    }
  }

/***
 * bluetooth device setup, send and receive done event
 ***/
```

```
event void BootTimer.fired()
{
  error_t error = SUCCESS;
  if(counter == 1){
    error = call SerialBluetooth.sendCommand(AT);
    printf("***---***\n post sendcmd : %s\n",AT);
    printfflush();
  }
  else if(counter == 2){
    error = call SerialBluetooth.sendCommand(BTMODE3);
    printf("***---***\n post sendcmd : %s\n",BTMODE3);
    printfflush();
  }
  else if(counter == 3){
    error = call SerialBluetooth.sendCommand(ATZ);
    printf("***---***\n post sendcmd : %s\n",ATZ);
    printfflush();
  }else if(counter == 4){
    error = call SerialBluetooth.setAttendConn();
    printf("***---***\n post sendcmd : attend Connection\n");
    printfflush();
  }
  counter++;
  if(error == FAIL)
    call Leds.led0On();
}

event void SerialBluetooth.sendCommandDone(error_t error)
{
  if(error == FAIL){
    call Leds.led0On();
    call Leds.led1Off();
    call ResBlue.release();
    call BluetoothControl.stop();
```

```
    }
    else{
      call Leds.led0Off();
      call Leds.led1On();
    }
  }

  event void SerialBluetooth.receiveCommandDone(error_t error)
  {
    if(error == FAIL){
      if(counter == 3){ // in this case the bluetooth is just on mode 3
        call Leds.led0Off();
        call Leds.led1On();
        call BootTimer.startOneShot(BOOT_SHOT_TIME);
      }else{
        call Leds.led0On();
        call Leds.led1Off();
        call ResBlue.release();
        call BluetoothControl.stop();
      }
    }
    else{
      call Leds.led1On();
      if(counter < 5){
        call BootTimer.startOneShot(BOOT_SHOT_TIME);
      }else{
        call Leds.led2On();
      }
    }
  }

  event void SerialBluetooth.connectionDone(error_t error){
    call ResRadio.request();
    call ResBlue.release();
```

```
  call MilliTimer.startPeriodic(3000);
}

event void SerialBluetooth.forwardingDone(error_t error){
  if (error == SUCCESS){
    call ResRadio.request();
    call ResBlue.release();
  }else{
    call MilliTimer.stop();
    call SerialBluetooth.setAttendConn();
  }
}

event void SerialBluetooth.sendPosToMote(char *str){
  char s[4];
  int i;
  RadioBluetoothMsg* rbpkt;
  for(i = 0; i < 4; i++)
    s[i] = str[i];
  send_node_id = atoi(s);
  for(i = 0; i < 4; i++)
    s[i] = str[i+4];
  send_x = atoi(s);
  for(i = 0; i < 4; i++)
    s[i] = str[i+8];
  send_y = atoi(s);
  rbpkt = (RadioBluetoothMsg*)(call Packet.getPayload(&pkt, NULL));
  rbpkt->seqno = seqno;
  rbpkt->im_pos = TRUE;
  rbpkt->x = send_x;
  rbpkt->y = send_y;
  send_pos = TRUE;
}
event void MilliTimer.fired()
```

```
  {
    if(send_pos)
      if(call AMSend.send(send_node_id, &pkt,
                          sizeof(RadioBluetoothMsg)) == SUCCESS){
     send_pos = FALSE;
      }
    if(!busy) {
      RadioBluetoothMsg* rbpkt = (RadioBluetoothMsg*)
                              (call Packet.getPayload(&pkt, NULL));
      rbpkt->seqno = seqno;
      rbpkt->im_pos = FALSE;
      rbpkt->x = 0;
      rbpkt->y = 0;
      if(call AMSend.send(AM_BROADCAST_ADDR, &pkt,
                          sizeof(RadioBluetoothMsg)) == SUCCESS) {
        busy = TRUE;
        seqno = seqno + 1;
      }
    }
  }

/***
 * Send and receive done radio event
 ***/
  event void AMSend.sendDone(message_t* msg, error_t error) {
    if(error==SUCCESS) {
      printf("AM Send Done: success seqno = %i\n",seqno-1);
      printfflush();
    }
    else {
      printf("AM Send Done: FAIL \n");
      printfflush();
    }
    if(&pkt==msg) {
```

```
    busy = FALSE;
  }
}
event message_t* Receive.receive(message_t* msg, void* payload,
                                                    uint8_t len) {
  char str[30] = "";
  char str1[3];
  uint16_t rx_seqno = 0;
  uint16_t nodeid = 0;
  uint8_t tx_pow = 0;
  int8_t rssi = 0;
  uint8_t lqi = 0;
  bool is_pos = FALSE;
  uint16_t x = 0;
  uint16_t y = 0;

  cc2420_header_t* header;
  call ResBlue.request();
  call ResRadio.release();

  if(len==sizeof(RadioBluetoothMsg)) {
    RadioBluetoothMsg* rbpkt = (RadioBluetoothMsg*)payload;
    rx_seqno = rbpkt->seqno;
    tx_pow = rbpkt->tx_power;
    rssi = rbpkt->rssi;
    lqi = rbpkt->lqi;
    is_pos = rbpkt->im_pos;
    x  = rbpkt->x;
    y = rbpkt->y;
  }

  header = (cc2420_header_t*)msg->header;
  nodeid = header->src;
```

```
    itoa(rx_seqno, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(nodeid, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(tx_pow, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(rssi, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(lqi, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(is_pos, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(x, str1, 10);
    strcat(str,str1);  strcat(str,"/");
    itoa(y, str1, 10);
    strcat(str,str1);
    call SerialBluetooth.forwardMsg(str);
    call Leds.led2Toggle();
    return msg;
  }
}
```

SerialBluetooth.nc

```
interface SerialBluetooth
{
  command bool isForwarding();

  command error_t setAttendConn();

  /***
   * Send AT command to Bluetooth device
   ***/
  command error_t sendCommand(char *str);

  /***
```

```
   * Send AT command to Bluetooth device
   ***/
  command error_t forwardMsg(char *str);

  /***
   * Receive the response on send command
   ***/
  event void sendCommandDone(error_t error);

  /***
   * Receive the response on receive command
   ***/
  event void receiveCommandDone(error_t error);

  /***
   * Receive the response on established connection
   ***/
  event void connectionDone(error_t error);

  /***
   * Receive the response on forwarding
   ***/
  event void forwardingDone(error_t error);

  /***
   * Receive the position from cellular and forward to a mote
   ***/
  event void sendPosToMote(char *str);
}
```

SerialBluetoothC.nc

```
configuration SerialBluetoothC
{
  provides interface SerialBluetooth;
  provides interface StdControl;
```

```
}
implementation
{
  components SerialBluetoothP;
  components new Msp430Uart0C() as UartC;

  SerialBluetoothP.Resource -> UartC.Resource;
  SerialBluetoothP.UartStream -> UartC.UartStream;
  SerialBluetoothP.Msp430UartConfigure <- UartC.Msp430UartConfigure;
  SerialBluetooth = SerialBluetoothP;
  StdControl = SerialBluetoothP.StdControl;

  components new TimerMilliC() as MilliTimer;
  SerialBluetoothP.DelayTimer -> MilliTimer;
}
```

SerialBluetoothP.nc

```
#include "SerialBluetooth.h"
#include "printf.h"

module SerialBluetoothP
{
  uses
  {
    interface Resource;
    interface UartStream;
    interface Timer<TMilli> as DelayTimer;
  }
  provides
  {
    interface Msp430UartConfigure;
    interface SerialBluetooth;
    interface StdControl;
  }
}
```

```
implementation
{

  msp430_uart_union_config_t msp430_uart_tmote_config = {
    {
      utxe : 1,
      urxe : 1,
      ubr : UBR_1MHZ_9600,
      umctl : UMCTL_1MHZ_9600,
      ssel : 0x02,
      pena : 0,
      pev : 0,
      spb : 0,
      clen : 1,
      listen : 0,
      mm : 0,
      ckpl : 0,
      urxse : 0,
      urxeie : 1,
      urxwie : 0,
      utxe : 1,
      urxe : 1
    }
  };

  uint16_t txBufferLength = 0;
  char txStrBuffer[BUFFER_SIZE];

  uint8_t rxBufferLength = 6;
  char rxStrBuffer[BUFFER_SIZE];

  bool attendConnection = FALSE; //TRUE = attend bluetooth connection
  bool forwarding = FALSE; //TRUE = it's in forwarding state
  bool busy = FALSE; //TRUE = bluetooth is sending the msg attend 30 ms
```

```
task void sendDoneSuccessTask()
{
  signal SerialBluetooth.sendCommandDone(SUCCESS);

  if(DBG_INT){
    printf("UartStream send : SUCCESS ...\n");
    printfflush();
  }
}

task void sendDoneFailTask()
{
  signal SerialBluetooth.sendCommandDone(FAIL);

  call UartStream.disableReceiveInterrupt();

  if(DBG_INT){
    printf("UartStream send : FAIL ... \n");
    printfflush();
  }
}

task void receiveDoneTask(){
  if(DBG){
    int i;
    for (i=0; i<rxBufferLength; i++){
      printf(" %c",rxStrBuffer[i]);
      printfflush();
    }
    printf("\n");
    printfflush();
  }
```

```
  //receive "OK"
  if(rxStrBuffer[2] == OK[0] && rxStrBuffer[3] == OK[1])
    signal SerialBluetooth.receiveCommandDone(SUCCESS);
  else
    //receive "ERR"
    if(rxStrBuffer[2] == ER[0] && rxStrBuffer[3] == ER[1])
      signal SerialBluetooth.receiveCommandDone(FAIL);

  call UartStream.disableReceiveInterrupt();
}

void setAttendConnection(bool state){
  atomic attendConnection = state;
}

void setForwarding(bool state){
  atomic forwarding = state;
}

command bool SerialBluetooth.isForwarding(){
  return forwarding;
}

task void connectionDoneTask(){
  if(DBG){
    int i;
    printf("Connection done : ");
    for (i=0; i<rxBufferLength; i++){
      printf(" %c",rxStrBuffer[i]);
      printfflush();
    }
    printf("\n");
    printfflush();
  }
```

```
    //receive "CONN"
    if(rxStrBuffer[2] == CONN[0] && rxStrBuffer[3] == CONN[1])
      setForwarding(TRUE);
    else
      //receive "DISC"
      if(rxStrBuffer[2] == DISC[0] && rxStrBuffer[3] == DISC[1]){
        setForwarding(FALSE);
        setAttendConnection(FALSE);
      }

    call UartStream.disableReceiveInterrupt();

    signal SerialBluetooth.connectionDone(SUCCESS);
  }

  task void forwardingDoneTask(){
    if(DBG){
      int i;
      printf("Forwarding done : ");
      for (i=0; i<rxBufferLength; i++){
        printf(" %c",rxStrBuffer[i]);
        printfflush();
      }
      printf("\n");
      printfflush();
    }

    //receive "OK"
    if(rxStrBuffer[2] == OK[0] && rxStrBuffer[3] == OK[1]){
      signal SerialBluetooth.forwardingDone(SUCCESS);
    }else
      //receive "CONN"
      if(rxStrBuffer[2] == CONN[0] && rxStrBuffer[3] == CONN[1]){
```

```
        setForwarding(TRUE);
        setAttendConnection(TRUE);
//        signal SerialBluetooth.forwardingDone(SUCCESS);
signal SerialBluetooth.connectionDone(SUCCESS);
      }else
        //receive "DISC"
        if(rxStrBuffer[2] == DISC[0] && rxStrBuffer[3] == DISC[1]){
          setForwarding(FALSE);
          setAttendConnection(TRUE);
          signal SerialBluetooth.forwardingDone(FAIL);
        }else{
          //the possible message is the position of a mote
          signal SerialBluetooth.forwardingDone(SUCCESS);
  signal SerialBluetooth.sendPosToMote(rxStrBuffer);
        }
        call UartStream.disableReceiveInterrupt();

  }

  task void send(){
    int i;
    error_t result;
    error_t resultRXEI;
    call UartStream.receive(rxStrBuffer, rxBufferLength);
    resultRXEI = call UartStream.enableReceiveInterrupt();
    result = call UartStream.send(txStrBuffer,txBufferLength);

    if(DBG){
      for (i=0; i<txBufferLength; i++){
        printf(" %c",txStrBuffer[i]);
        printfflush();}
      printf("\n");
      printfflush();}
    if(DBG_INT){
```

```
    if(resultRXEI == SUCCESS)
      printf("UartStream ERI : SUCCESS ...\n");
    else
      printf("UartStream ERI : FAIL ...\n");
    printfflush();
  }

  if(result == FAIL)
  {
    post sendDoneFailTask();
  }
}

command error_t StdControl.start(){
  return call Resource.immediateRequest();
}

command error_t StdControl.stop(){
  call Resource.release();
  return SUCCESS;
}

command error_t SerialBluetooth.setAttendConn(){
  setAttendConnection(TRUE);
  rxBufferLength = 6;
  call UartStream.receive(rxStrBuffer, rxBufferLength);
  call UartStream.enableReceiveInterrupt();
  return SUCCESS;
}

command error_t SerialBluetooth.sendCommand(char *str)
{
  txBufferLength = strlen(str);
  if(txBufferLength > BUFFER_SIZE-1)
```

```
    return FAIL;
  memcpy(txStrBuffer,str,txBufferLength);
  txStrBuffer[txBufferLength] = 0x0D; // add carrige return to command
  txBufferLength = txBufferLength +1;
  rxBufferLength = 6;
  post send();
  return SUCCESS;
}

command error_t SerialBluetooth.forwardMsg(char *str)
{
  if(!busy){
    busy = TRUE;
    txBufferLength = strlen(str);
    if(txBufferLength > BUFFER_SIZE-1)
      return FAIL;
    memcpy(txStrBuffer,str,txBufferLength);
    rxBufferLength = 12;
    post send();
    return SUCCESS;
  }else
    signal SerialBluetooth.forwardingDone(FAIL);
    return FAIL;
}

event void Resource.granted()
{

}

// respect the min delay of bluetooth transmition
event void DelayTimer.fired()
{
  busy = FALSE;
```

```
}

async event void UartStream.sendDone(uint8_t *buf, uint16_t len,
                                                    error_t error)
{
  if(!forwarding){
    if(error == SUCCESS)
      post sendDoneSuccessTask();
    else
      post sendDoneFailTask();
  }else
    call DelayTimer.startOneShot(DELAY_TIME);
}

async command msp430_uart_union_config_t* Msp430UartConfigure.getConfig()
{
  return &msp430_uart_tmote_config;
}

async event void UartStream.receivedByte(uint8_t byte)
{
}

async event void UartStream.receiveDone(uint8_t* buf, uint16_t len,
                                                      error_t error)
{
  if(!attendConnection)
    post receiveDoneTask();
  else
    if(!forwarding)
      post connectionDoneTask();
    else
      post forwardingDoneTask();
}
```

```
}
```

SerialBluetooth.h

```
#ifndef SERIAL_BLUETOOTH_H
#define SERIAL_BLUETOOTH_H
enum
{
  BUFFER_SIZE = 256,
};

typedef nx_struct RadioBluetoothMsg {
  nx_uint16_t seqno;
  nx_uint8_t tx_power;
  nx_uint8_t rssi;
  nx_uint8_t lqi;
  nx_bool im_pos;
  nx_uint16_t x;
  nx_uint16_t y;
} RadioBluetoothMsg;

enum {
  AM_RADIOMSG = 6,
  TIMER_RADIO_PING = 1000
};

uint16_t BOOT_SHOT_TIME = 250;
uint16_t DELAY_TIME = 32;

bool DBG_INT = 0; // 0:disable 1:enable debug msg for interrupt to pc
bool DBG     = 1; // 0:disable 1:enable debug msg to pc

//Test comunication over UART to Bluetooth
char *AT = "AT";
//Change Bluetooth chipset mode read for connaction
char *BTMODE3 = "AT+BTMODE,3";
```

```
//Reset software
char *ATZ = "ATZ";

char *OK = "OK"; //result exec comand OK
char *ER = "ER"; //result exec comand ERROR
char *CONN = "CONN"; //result exec comand CONNECT
char *DISC = "DISC"; //result exec comand DISCONNECT

#endif
```

Appendice C

BlueTmote Tracking

Di seguito viene proposto il codice sviluppato per l'applicazione di localizzazione scritto in Java per lo smartphone Android.

Solver.java

```
/**
 * La classe Solver.java offre il metodo per il calcolo della posizione
 * del nodo mobile.
 */
package com.navlab.android.BlueTmoteTracking;

import android.graphics.Point;
import android.graphics.PointF;
import android.util.Log;

/**
 * @author Stefano Dazzo
 */
public class Solver extends Thread{
// Debugging
private static final String TAG = "BlueTmote";
private static final boolean D = true;

private Mote[] ancoreNodes;
private RoomView display;
```

```
//parametri utilizzati dal sistema Teseo
public static final double sigma = 7.57;
public static final double A = -63.67;
public static final double np = 2.12;
public double expPesoDistanza = -2;

public double gamma;
public double c;
public double exp;

public PointF position;

private int seqno = -1;

public Solver(RoomView view){
  display = view;
  gamma = Math.pow((sigma*Math.log(10))/(10*np), 2);
  c = Math.pow(Math.E, (gamma/2));
}

public void run(){
  while(true){
            position = tracking();
            display.mobileNode(position);
            // Delay
            try {
               Thread.sleep(2000);
            } catch (InterruptedException e) { }
  }
}

public void cancel(){}
```

```
public void setAncoreNode(Mote[] mote){
  synchronized(this){
    ancoreNodes = mote;
  }
}
double x = 0;
double y = 0;
public synchronized PointF tracking(Mote[] mote){
  //tecnica pesata
  ancoreNodes = mote;
  double sommaPesi = 0;       //pesata su 1/d^1.5
  double sommaPesataX = 0;
  double sommaPesataY = 0;
  for(int i = 0; i < ancoreNodes.length; i++){
    if(ancoreNodes[i] != null){
      double exp = (A - ancoreNodes[i].getRssi()) / 10 * np;
      double distanza = Math.pow(10,exp)/c;
      double pesiDistanze = Math.pow(distanza, expPesoDistanza);
      sommaPesi += pesiDistanze;               //1/d^expPesoDistanza
      sommaPesataX += pesiDistanze * (ancoreNodes[i].getX() );
      sommaPesataY += pesiDistanze * (ancoreNodes[i].getY() );
    }
  }
  x = sommaPesataX / sommaPesi;
  y = sommaPesataY / sommaPesi;
  if(D) Log.d(TAG, x+" "+y);
  return new PointF((float)x,(float)y);
}
}
```

Bibliografia

[1] Reto Meier, *Professional Android 2 application developement*, p 426, 427, 430, 433-434

[2] Marabese Daniele, Tesi di laurea: *Analisi delle principali piattaforme mobile ai fini del loro impiego per la gestione dell'assistenza clienti di una ditta informatica*, pagine 29-31, 34-37, 116-119, 145-152.

[3] Moteiv Corporation, *Tmote sky*, `http://www.cs.uvm.edu/~crobinso/mote/tmote-sky-quickstart-110.pdf`, Rev. B, April 2005, Document 7430-0021-06.

[4] Chipcon AS Products, *CC2420 Datasheet*, www.chipcon.com.

[5] Sena Technologies Inc., *Parani-ESD100V2/110V2 USER GUIDE*, ver.2, 19-11-2009.

[6] Documentazione Smartphone. `http://www.lg.com/it/mobile/mobile-phones/LG-smartphone-android-LG-Optimus-One.jsp`. LG Electronics Italia S.p.A.

[7] Documentazione Android. `http://developer.android.com/guide/basics/what-is-android.html`.

[8] David Gay, David Culler, and Philip Levis. *Nesc language reference manual*, settembre 2002.

[9] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, David Culler. *The nesC Language:A Holistic Approach to Networked Embedded Systems*. giugno 2003.

[10] Crossbow Technology. *TinyOS Getting Started Guide, Rev. A*, October 2003 document 7430-0022-03. tmote-sky-datasheet

[11] Documentazione TinyOS, `http://docs.tinyos.net/tinywiki/index.php`.

[12] Documentazione API Bluetooth per Android, `http://developer.android.com/guide/topics/wireless/bluetooth.html`.

[13] Dakai Zhu, Ali Tosun. *RF Communication for LEGO/Handy Board with Tmote.* Department of Computer Science University of Texas at San Antonio. 2008.

[14] TinyOS Enhancement Proposals TEP 108: Resource Arbitration, `http://www.tinyos.net/tinyos-2.x/doc/html/tep108.html`

[15] TinyOS Enhancement Proposals TEP 113: Serial Communication, `http://www.tinyos.net/tinyos-2.x/doc/html/tep113.html`.

[16] A. Smith, H. Balakrishnan, M. Goraczko, N. Priyantha. *Tracking Moving Devices with the Cricket Location System.* Technical report, MIT Computer Science and Artificial Intelligence Laboratory, The Stata Center, 32 Vassar Street, Cambridge, MA 02139, 2004.

[17] N. B. Priyantha, A. Chakraborty, H. Balakrishnan. *The Cricket Location-Support System.* 6th ACM International Conference on Mobile Computing and Networking (ACM MOBICOM), August 2000.

[18] P. Bahl, V. N. Padamanabhan. *RADAR: An In-Building RF-based User Location and Tracking System.* INFOCOM, pages 775-784, 2000.

[19] P. Bahl, V. N. Padamanabhan, A. Balachandran. *Enhancements to the RADAR User Location and Tracking System.* Technical report, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, February 2000.

[20] K. Lorincz, M. Welsh. *MoteTrack: A Robust, Decentralized Approach to RFBased Location Tracking.* Division of Engineering and Applied Sciences, Harvard University.

[21] K. Lorincz, M. Welsh. *A Robust, Decentralized Approach to RF Based Location Tracking.* Technical report, Harvard University, 2004.

[22] M. Maroti, B. Kusy, G. Balogh, P. Volgyesi, A. Nadas, K. Molnar, S. Dora, A. Ledeczi. *Radio Interferometric Positioning*. Technical report, Institute for Software Integrated Systems Vanderbilt University, November 2005.

[23] M. Bertinato, G. Ortolan, F. Maran, R. Marcon, A. Marcassa, F. Zanella, P. Zambotto, L. Schenato, A. Cenedese. *RF Localization and tracking of mobile nodes in Wireless Sensors Networks: Architectures, Algorithms and Experiments*. DEI, Department of Information Engineering, University of Padua, Italy.

[24] Luca Parolini. *Metodi di Localizzazione per Reti di Sensori Wireless*. Tesi di Laurea, Università degli Studi di Padova 2006.

[25] Simone Cieno. *Calibrazione di sensori distribuiti tramite algoritmi di consensus*. Tesi di Laurea, Università degli Studi di Padova 2007.

Ringraziamenti

Nell'ultima facciata di questa tesi ringrazio il Professore Luca Schenato che mi ha dato la possibilità di lavorare a questo progetto e poter giungere al traguardo della laurea, lo ringrazio per la disponibilità e la fiducia.

Ringrazio anche i proprietari del locale in cui ho potuto effettuare gli esperimenti e le riprese per testare il progetto che ho realizzato, perché ho potuto disporre di una stanza sufficientemente grande.

Ringrazio i miei genitori e la mia famiglia che mi ha dato i mezzi per affrontare questo cammino verso la laurea. Mezzi non solo economici ma anche morali, nei momenti di sconforto erano sempre presenti.

Ringrazio infine i miei più cari amici sempre vicini nei momenti difficili ma anche nei momenti di gioia e festa, perché qualche volta bisogna pur sgombrare la mente e aprire lo sguardo a nuovi orizzonti.

A tutte quelle persone che hanno avuto un peso nella mia carriera universitaria ma che ora sono lontane o irraggiungibili, insomma un grazie di cuore a tutti.