Università degli Studi di Milano

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Triennale in Informatica

Sviluppo, implementazione e simulazione di un algoritmo distribuito di autolocalizzazione per reti di sensori wireless

Relatore: Prof. Federico Pedersini

Correlatore: Dott. Anna Morpurgo

Tesi di laurea di:

Alessandro Reina

Matricola n. 656509

Anno Accademico 2005/2006

A chi mi è stato vicino senza essere presente

Indice

Introduzione

Il progresso tecnologico avvenuto nell'ambito della miniaturizzazione elettronica e i conseguenti numerosi miglioramenti nei sistemi micro elettromeccanici (MEMS), nelle comunicazioni *wireless* e nei sistemi digitali a basso consumo energetico hanno portato allo sviluppo delle reti di sensori, cioè reti di piccoli dispositivi di calcolo equipaggiati di un sensore (della grandezza fisica che si vuole rilevare), un chip per la comunicazione via radio, un'unità di elaborazione dei dati ed un sistema di alimentazione autonomo. Ambiti di applicazione delle reti di sensori sono molteplici: militari, ambientali, legati alla salute e ambienti intelligenti. La maggior parte delle applicazioni richiedono la conoscenza della posizione dei nodi. Diversi approcci sono stati proposti in letteratura per risolvere il problema della localizzazione dei nodi: tramite l'uso di nodi àncora, cioè di nodi la cui posizione è nota, senza àncora, concorrenti e incrementali. L'algoritmo proposto in questa tesi è un algoritmo distribuito in cui ciascun nodo si autolocalizza esclusivamente sulla base della stima della distanza dai propri vicini, senza l'ausilio di nodi àncora. Più precisamente l'algoritmo risolve il problema in quattro fasi: nella prima fase i nodi trasmettono messaggi broadcast, ciascun nodo identifica i propri vicini e fa una prima stima delle distanze da questi; nella seconda fase vengono determinate le posizioni di tre nodi iniziali, che fungeranno poi da nodi àncora; nella terza fase i nodi non an-

cora localizzati sfruttano le informazioni dei vicini localizzati per fare a loro volta una prima stima della propria posizione; nella quarta fase si procede con un'ottimizzazione in cui si immagina una molla tra ogni coppia di vicini che spinge e tira i nodi al fine di trovare una situazione di equilibrio che minimizza l'energia tra i nodi. L'algoritmo è stato poi implementato come applicazione *TinyOs* per nodi di sensori *mote MicaZ*. Per valutare l'algoritmo è stato sviluppato un ambiente che integra in un unico strumento i sistemi *Matlab, Cygwin, Python, TinyOs* che interoperano per l'esecuzione di simulazioni ripetute e la raccolta dei risultati ai fini di un'analisi delle prestazioni. La tesi è organizzata come segue: nel primo capitolo vengono introdotte le reti di sensori. Nel secondo capitolo si illustrano gli algoritmi di localizzazione noti in letteratura. Nel terzo capitolo si procede alla motivazione, definizione e descrizione formale dell'algoritmo proposto. Nel quarto capitolo vengono esposte le scelte progettuali per l'implementazione dell'algoritmo e viene descritto l'ambiente sviluppato per la simulazione e valutazione del medesimo. Nel quinto capitolo si procede nell'analisi delle prestazioni dell'algoritmo mediante simulazione. Infine nel sesto capitolo vengono fatte alcune proposte di proseguimento del lavoro. Una breve introduzione al sistema operativo TinyOS e al linguaggio di programmazione *nesC* si trova negli appendici.

Capitolo 1

Reti di sensori

Le sempre crescenti esigenze in termini di elaborazione e trasmissione di dati ed i progressi tecnologici hanno portato allo sviluppo di reti di sensori [6], o *Wireless Sensor Network* (WSN), cioè di reti di piccoli dispositivi di calcolo autonomi dotati di sensori per il monitoraggio di diversi fenomeni fisici, come pressione, suono, vibrazioni, inquinamento, temperatura, umidità, luminosità e così via; situazioni di interesse, ad esempio in campo medico, per la cura degli ambienti agricoli e forestali, per il trasporto, in campo militare e altro ancora. L'evoluzione dell'elettronica digitale ha permesso lo sviluppo di nodi di dispositivi dotati di sensori, a basso costo e a basso consumo di piccole dimensioni e capaci di comunicare a brevi distanze. Tali dispositivi possono coprire l'ambiente da monitorare per lunghi periodi di tempo consentendo analisi dettagliate e sviluppo di molteplici applicazioni. Attuali argomenti di ricerca nell'ambito delle reti di sensori sono lo studio di come limitare il consumo dell'energia e aumentare così l'autonomia dei nodi, di come raccogliere, elaborare e memorizzare i dati efficientemente, cercando di rappresentarli in modo che l'occupazione in memoria sia minore possibile, e lo studio di protocolli di instradamento e algoritmi di localizzazione.

1.1 Nodi di sensori

I nodi di sensori si possono immaginare come dei piccoli computer, estremamente semplici in termini delle loro interfacce e dei loro componenti. Di solito sono equipaggiati di un microprocessore, di un dispositivo di memorizzazione dei dati, di sensori, di un convertitore analogico/digitale (ADC), di un *data transceiver* ovvero un dispositivo costituito da trasmettitore e ricevitore ed infine di una sorgente di alimentazione autonoma (generalmente batterie). La loro dimensione può variare dalla dimensione di una scatola da scarpe fino alla grandezza di un granellino di polvere. Il costo di un sensore è variabile in un intervallo di un centinaio di euro a pochi centesimi a seconda della dimensione e complessità richiesta del nodo. I vincoli di dimensione e costo corrispondono a vincoli sulle risorse, come: energia, memoria, velocità di elaborazione e larghezza di banda di comunicazione. Vediamo ora, più in dettaglio, gli elementi principali che costituiscono un nodo di sensori.

1.1.1 Sensori

Un sensore ha il compito di trasformare una grandezza fisica in un segnale elettrico da elaborare. Ogni nodo può integrare uno o più sensori diversi che acquisiscono grandezze fisiche differenti. Il consumo di energia, di cui occorre sempre tenere conto, dipende dal tipo di sensore e dall'uso che se ne fa. Ad esempio il convertitore di segnale analogico/digitale aumenta i consumi di energia in proporzione all'aumento della frequenza di campionamento.

1.1.2 Unità di elaborazione

Ciascun nodo è dotato di una CPU (*Central Processing Unit*) che oltre a permettere l'elaborazione dei dati acquisiti dal sensore prima di trasmetterli,

può essere programmata per sviluppare applicazioni più sofisticate. Ad esempio, per mezzo della CPU è possibile sviluppare applicazioni capaci di disattivare opportunamente dispositivi, come il chip radio, quando non vi è la necessità di comunicare, o altri dispositivi.

1.1.3 Dispositivo di comunicazione

Le reti di sensori sono dotate di un chip radio che permette la comunicazione senza fili. Per contenere il consumo energetico nelle reti di sensori si limita il raggio di trasmissione ad alcune decine di metri. Questa decisione è legittimata dal fatto che rispetto alle reti locali wireless, le reti di sensori wireless sono molto più dense, di conseguenza è possibile coprire lunghi percorsi instradando l'informazione *hop by hop* attraverso i nodi. Anche dopo questi accorgimenti, la comunicazione wireless dell'informazione presenta una delle principali operazioni di consumo dell'energia, da tener conto anche nello sviluppo del software.

1.2 Descrizione del Mote

Smart Dust, polvere intelligente, è un progetto di Kristofer Pister [24] del 2001, che pone le radici per una nuova tecnologia che studia le possibilità d'integrazione di sensori, processori e sistemi di comunicazione in un unico millimetrico dispositivo. *Smart Dust*, basandosi su tre concetti fondamentali di miniaturizzazione, integrazione e basso consumo energetico, ha portato allo sviluppo di piccoli nodi di sensori, chiamati *mote*, che soddisfano queste caratteristiche. I mote utilizzano sistemi micro-elettro-meccanici, MEMS, che sono l'integrazione di elementi meccanici, sensori, attuatori e componenti elettronici su un unico substrato comune di silicio. In figura 1.1 possiamo osservare l'architettura del *Tmote Sky* prodotto dalla *moteiv*, e in figura 1.2

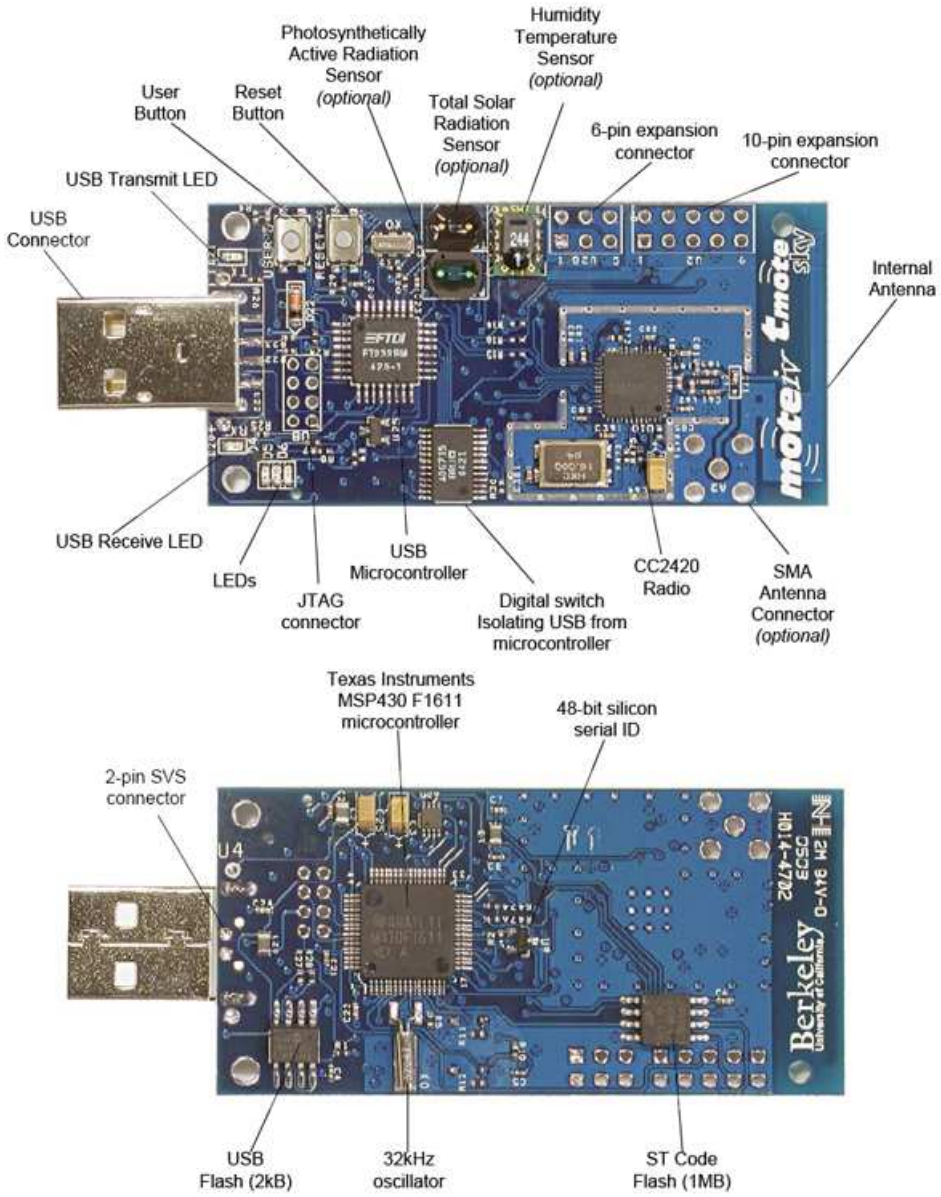


Figura 1.1: Tmote Sky caratteristiche fronte e retro

il suo schema a blocchi con il microcontrollore che gestisce sensori, chipset radio, memoria flash e il componente JTAG, mediante il quale è possibile l'interfacciamento con hardware esterno, ad esempio un computer. Le figure 1.5, 1.6, 1.7, 1.8, 1.9 illustrano alcuni modelli di mote disponibili in commercio. Infatti in molti casi è necessaria la comunicazione tra un *mote*, che

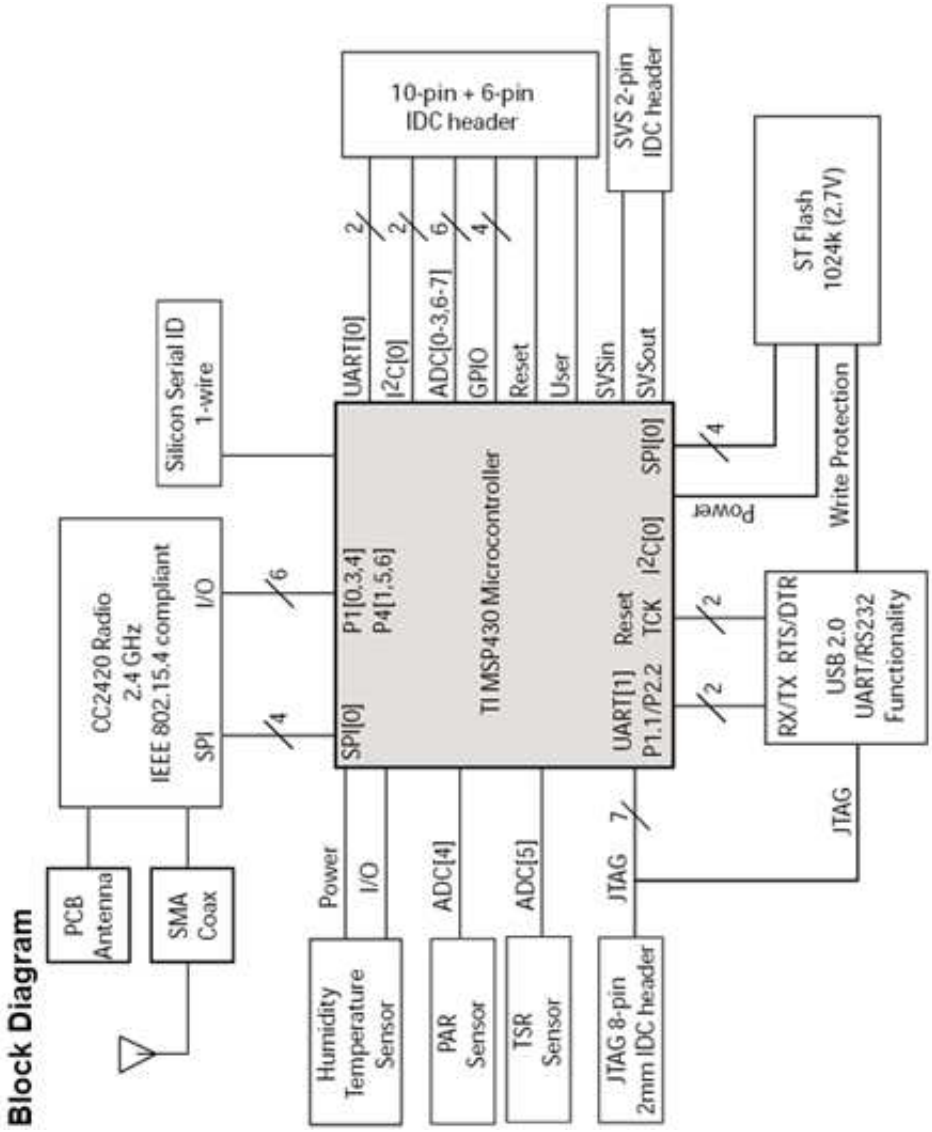
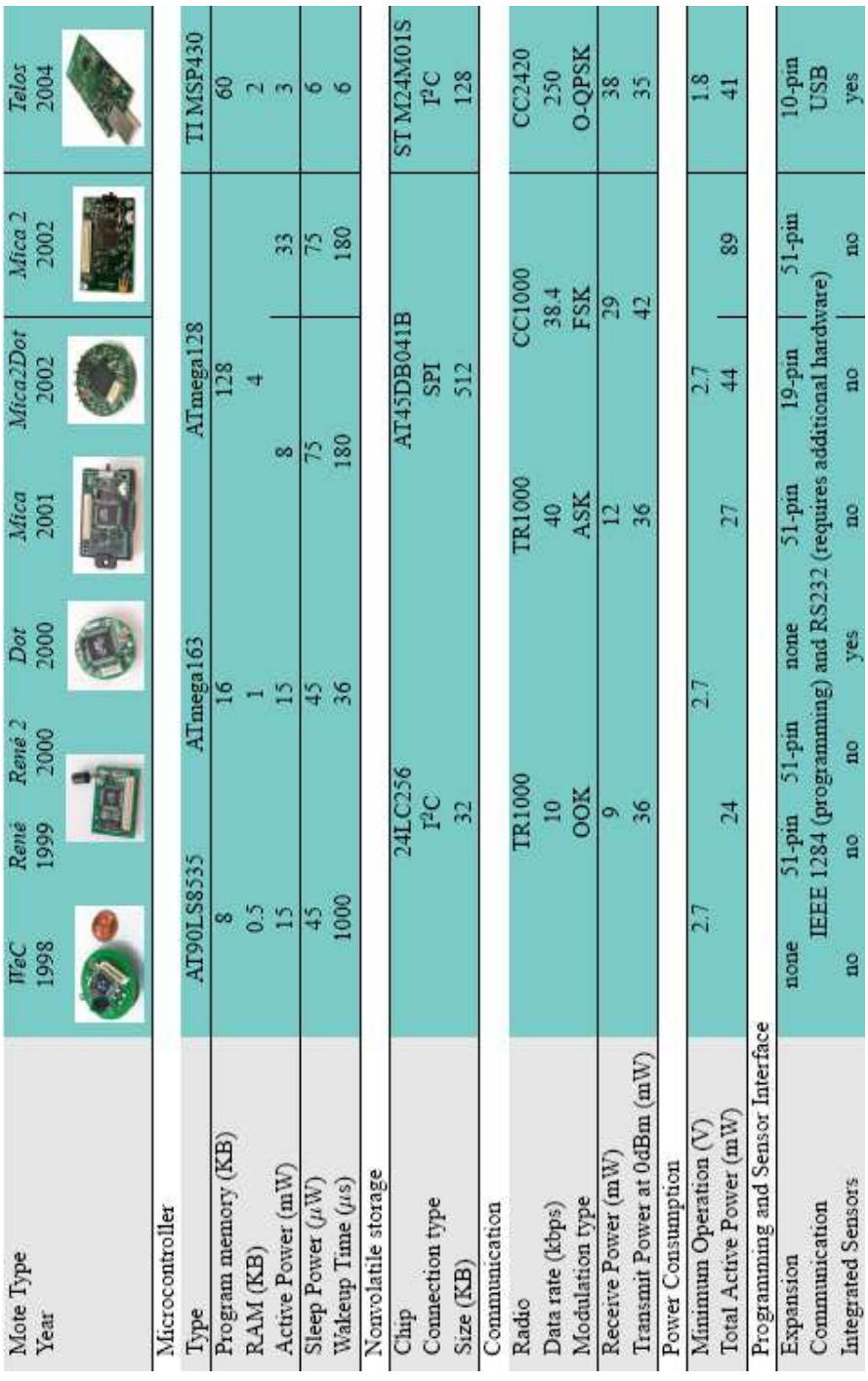


Figura 1.2: Schema a blocchi del Tmote Sky

funge da *base station*, ed un computer per poter memorizzare i dati rilevati dalla rete di sensori. L'evoluzione tecnologica di questi piccoli dispositivi è mostrata in figura 1.3.



Mote Type	WeC	René	René 2	Dot	Mica	Mica2Dot	Mica 2	Telos
Year	1998	1999	2000	2000	2001	2002	2002	2004
Microcontroller								
Type	AT90LS8535	ATmega163			ATmega128			TI MSP430
Program memory (KB)	8	16			128			60
RAM (KB)	0.5	1			4			2
Active Power (mW)	15	15		8		33		3
Sleep Power (μW)	45	45		75		75		6
Wakeup Time (μs)	1000	36		180		180		6
Nonvolatile storage								
Chip	24LC256			AT45DB041B				ST M24M01S
Connection type	I²C			SPI				I²C
Size (KB)	32			512				128
Communication								
Radio	TR1000			TR1000		CC1000		CC2420
Data rate (kbps)	10			40		38.4		250
Modulation type	OOK			ASK		FSK		O-QPSK
Receive Power (mW)	9			12		29		38
Transmit Power at 0dBm (mW)	36			36		42		35
Power Consumption								
Minimum Operation (V)	2.7		2.7			2.7		1.8
Total Active Power (mW)	24			27		44	89	41
Programming and Sensor Interface								
Expansion	none	51-pin	51-pin	none	51-pin	19-pin	51-pin	10-pin
Communication	IEEE 1284 (programming) and RS232 (requires additional hardware)							USB
Integrated Sensors	no	no	no	yes	no	no	no	yes

Figura 1.3: Progresso tecnologico dei mote [9]

1.3 Architettura della rete di sensori

Una rete di sensori è costituita da un largo numero di dispositivi autonomi, detti nodi sensori, che vengono dislocati all'interno o vicini ad un'area da monitorare o fissati su oggetti di cui si vuole tener traccia (veicoli, animali o persone). Le reti possono essere organizzate in modo centralizzato o distribuito: nelle reti centralizzate i nodi comunicano con un solo nodo, detto *sink*, che elabora le informazioni al posto dei nodi; nelle reti distribuite i nodi hanno la capacità di elaborare le informazioni in maniera indipendente. Siccome le reti di sensori sono state pensate per essere utilizzate anche in terreni inaccessibili oppure in aree di catastrofi, la posizione dei nodi spesso non può essere predeterminata, ad esempio se si pensa di lanciarli da un areo. Per questo motivo la rete deve essere sufficientemente densa per garantire la copertura dell'area da monitorare. In altre parole, ciò significa che i protocolli e gli algoritmi per le reti di sensori devono permettere alla rete di organizzarsi autonomamente. La realizzazione di applicazioni di reti di sensori richiede tecniche di reti wireless ad hoc, cioè reti composte solo da stazioni o dispositivi wireless che comunicano in modo diretto senza l'uso di *access point* o qualunque tipo di connessione di rete via cavo. Anche se diversi protocolli e algoritmi sono già stati proposti per le reti tradizionali ad hoc wireless, questi non si adattano alle qualità e requisiti delle applicazioni per reti di sensori. Per illustrare questo problema le differenze [8] tra le reti di sensori e le reti ad hoc vengono delineate qui di seguito:

- il numero di nodi di sensori in una rete di sensore può essere di diversi ordini di grandezza maggiore dei nodi in una rete ad hoc
- i nodi di sensori sono disposti in modo denso
- i nodi di sensori sono propensi ai fallimenti
- la topologia delle reti di sensori cambia frequentemente

- i nodi di sensori usano principalmente una comunicazione broadcast mentre molte reti ad hoc sono basate su una comunicazione punto a punto
- i nodi di sensori hanno limitate energie, capacità computazionale e memoria
- i nodi di sensori possono non avere un identificatore globale

Uno dei principale fattori critici nelle reti di sensori è appunto il consumo dell'energia. I nodi di sensori hanno una limitata, generalmente non sostituibile, sorgente di energia, dunque, mentre le reti tradizionali ad hoc hanno lo scopo di raggiungere un alta qualità di servizio (QoS), i protocolli delle reti di sensori devono prima di tutto tener conto della limitata energia a disposizione.

1.4 Applicazioni delle reti di sensori

Con le reti di sensori si può monitorare una grande varietà di condizioni ambientali:

- temperatura
- umidità
- movimento
- luce
- pressione
- struttura del terreno
- livello del rumore

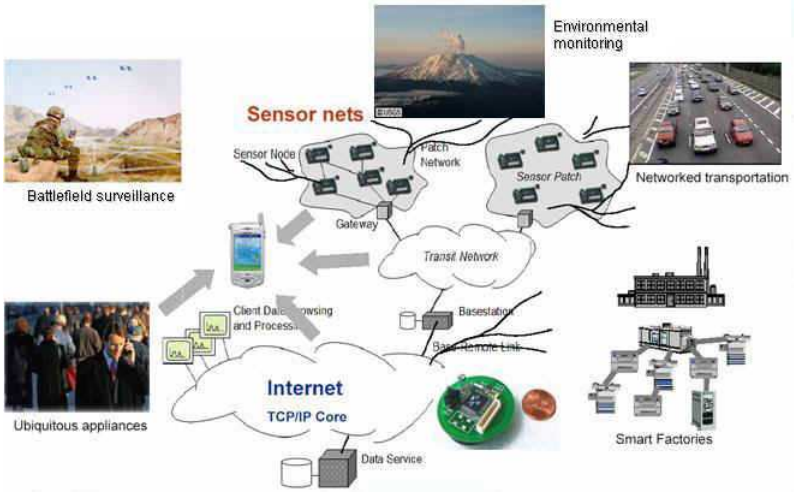


Figura 1.4: Architettura di reti di sensori (WenZhan Song - Sensorweb Research Laboratory)

- presenza o assenza di diversi tipi di oggetti
- sollecitazioni meccaniche
- velocità, direzione e dimensione di un oggetto
- ...

Possiamo suddividere le applicazioni dei nodi di sensori in:

- applicazioni militari
- applicazioni ambientali
- applicazioni legate alla salute
- applicazioni per ambienti intelligenti

Esempi di acquisizione e reperimento di informazioni monitorate da reti di sensori in tempo reale sono mostrati in figura 1.4.

1.4.1 Applicazioni militari

Le reti di sensori sono state utilizzate per la prima volta in ambito militare dove alcune delle applicazioni sono monitorare le forze alleate, l'equipaggiamento, le munizioni e i puntamenti, sorvegliare il campo di battaglia e la dislocazione delle forze nemiche per conoscere la stima dei danni della battaglia, quindi scoprire e fare la ricognizione di attacchi nucleari e chimici.

Monitorare le forze alleate, equipaggiamento e munizioni

I comandanti possono costantemente monitorare lo stato delle forze alleate, le condizioni e la disponibilità degli equipaggiamenti e munizioni in un campo di battaglia mediante l'uso delle reti di sensori. Ogni truppa, veicolo, equipaggiamento e munizione può essere dotato di un piccolo sensore che riferisce lo stato. I dati rilevati sono poi riportati ad un nodo centrale, detto *sink*, e spedite ai comandanti.

Sorveglianza del campo di battaglia

Parti del campo di battaglia, incroci nelle strade, fossi, possono creare delle situazioni di attacco che è meglio controllare in anticipo, coprendole con dei nodi di sensori che continuano a monitorare lo stato delle forze nemiche.

Dislocazione delle forze nemiche

Le reti di sensori possono essere installate in terreni critici per ottenere informazioni tempestive sullo stato delle truppe nemiche prima che si possa essere intercettati.

Puntamenti

Le reti di sensori possono essere incorporate all'interno dei sistemi di guida di munizioni intelligenti.

Stima dei danni della battaglia

Appena prima di un attacco, le reti di sensori possono essere installate nell'area obiettivo per raccogliere le informazioni sulla stima dei danni.

Scoperta e ricognizione di attacchi nucleari e chimici

Nelle guerre chimiche o nucleari, è importante la tempestiva informazione sugli agenti presenti nell'area controllata. In questo modo è possibile informare in anticipo attraverso le reti di sensori che rilevano tali agenti, le forze alleate che prenderanno dei provvedimenti prima che avvenga una tragedia.

1.4.2 Applicazioni ambientali

Alcune delle applicazioni delle reti di sensori nell'ambito ambientale includono il rilevamento dei movimenti di uccelli, di piccoli animali, e di insetti, il monitoraggio di particolari condizioni ambientali (terreno, mare, atmosfera, irrigazioni) la rilevazione di componenti chimici e biologici (agricoltura di precisione) ed infine la rilevazioni di incendi e allagamenti.

Rilevazione di incendi nelle foreste

I nodi di sensori possono essere strategicamente, casualmente e densamente dislocati in una foresta, in modo che la rete possa comunicare l'esatta origine dell'incendio prima che l'incendio si estenda in maniera incontrollabile.

Agricoltura di precisione

Alcuni dei benefici dell'uso delle reti di sensori nell'agricoltura di precisione sono la capacità di monitorare il livello di pesticidi nell'acqua potabile, il livello di erosione del terreno e il livello dell'inquinamento in tempo reale.

1.4.3 Applicazioni legate alla salute

Svariate applicazioni delle reti di sensori legate alla salute sono: fornire interfacce per disabili, monitorare pazienti, eseguire valutazioni diagnostiche, amministrare farmaci in ospedale, telemonitorare dati fisiologici umani, rilevare e monitorare dottori e pazienti all'interno dell'ospedale.

Telemonitoraggio dei dati fisiologici umani

Dati fisiologici possono essere raccolti da una rete di sensori su un lungo periodo di tempo per essere usati al fine di ricerche mediche. Il paziente viene dotato di un piccolo e leggerissimo nodo di sensori applicato al suo corpo. Il nodo ha uno specifico compito, ad esempio, rilevare la frequenza cardiaca o misurare la pressione nel sangue. Questi piccoli nodi di sensori permettono al paziente grande libertà nei movimenti e concedono ai dottori la possibilità di identificare i sintomi in modo tempestivo.

1.4.4 Applicazioni per ambienti intelligenti

Home automation

Il progresso tecnologico ha permesso di integrare sensori anche in ambiente domestico dove i nodi possono comunicare tra di loro e con reti esterne, ad esempio internet. Questo permetterebbe all'utente di gestire la sua casa, in modo estremamente semplice, localmente e in modo remoto ovunque lui sia. É possibile la predisposizione di una serie di servizi personalizzati differenti a seconda della persona presente nell'ambiente.

Controllo ambientale negli uffici

L'aria condizionata e il sistema di riscaldamento di molti palazzi sono centralizzati. La temperatura in una stanza può variare di pochi gradi, da un

lato della stanza può essere troppo alta e dall'altro troppo bassa perché c'è solo un termostato nella stanza e perché il flusso d'aria dal sistema centrale non è distribuito in modo uguale in tutti i punti della stanza. Un sistema distributo di rete di sensori può essere installato per controllare il flusso dell'aria e la temperatura in diverse zone della stanza.

Museo interattivo

Nel prossimo futuro, i bambini potranno interagire con gli oggetti di un museo e imparare sempre di più su ciò che stanno osservando. Questi oggetti saranno in grado di interagire con l'utente in base al tatto e alla ricognizione del segnale vocale.

Gestione del magazzino

É possibile dotare ogni oggetto di un sensore in modo da poter tracciare la sua locazione in qualunque momento da qualunque utente. Infatti potrebbe essere necessario, da parte del personale di una ditta, controllare lo stato delle scorte, oppure per l'utente finale sapere con precisione dove si trova l'oggetto che ha ordinato.

Figura 1.5: MPR410CB

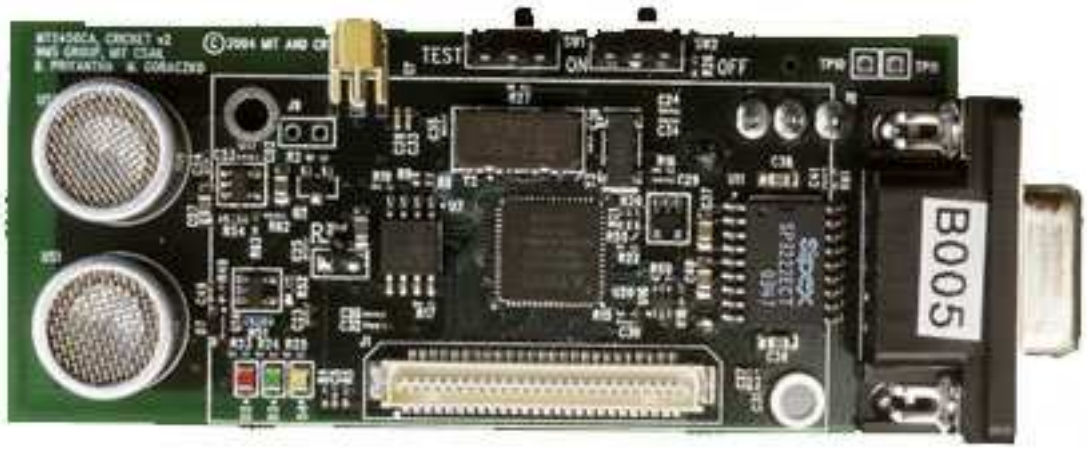


Figura 1.6: MCS410 Cricket Mote

Figura 1.7: MPR2400 MICAz

Figura 1.8: TPR2400CA-TelosB

Figura 1.9: Stargate

Capitolo 2

Panoramica sugli algoritmi di localizzazione

Diverse applicazioni che utilizzano reti di sensori richiedono che ogni nodo appartenente alla rete sia localizzato fisicamente rispetto ad un comune sistema di coordinate. Ad esempio un algoritmo di instradamento, per decidere quale sia il percorso più breve per la trasmissione di un pacchetto, potrebbe sfruttare informazioni sulla localizzazione dei nodi. Spesso le reti di sensori sono composte da migliaia di nodi, talvolta posizionati in ambienti difficili da raggiungere, dove non è sempre possibile l'installazione manuale, e ciò rende impossibile collocare ciascun nodo in una posizione stabilita. Si potrebbero localizzare i nodi di una rete attraverso una misura manuale, ma ciò comporterebbe una soluzione non scalabile. Oppure, si potrebbero dotare di sensori GPS i nodi, ma anche se questa soluzione è scalabile e non soggetta ad errori, oltre ad essere proponibile solo in spazi aperti, introduce svantaggi, quali il maggiore consumo energetico, il costo e la dimensione del dispositivo. Per far fronte a questi problemi esistono diverse classi di algoritmi di localizzazione per reti di sensori, dove ogni nodo comunica con i propri vicini stimando la distanza da ciascuno di essi fino ad arrivare ad una

stato di convergenza della rete, in cui ogni nodo ha determinato la propria posizione.

2.1 Caratteristiche degli algoritmi

Data la natura *ad hoc** delle reti di sensori wireless, è necessario che la rete sia in grado di configurarsi automaticamente. Gli algoritmi utilizzati per la localizzazione, devono inoltre essere in grado di funzionare anche quando si verificano perdite di connettività tra nodi e di stimare la posizione corretta anche in presenza di rumori o errori, e devono essere sviluppati ottimizzando il consumo di energia. La localizzazione dei sensori della rete può avvenire in modo distribuito oppure centralizzato. Nel caso distribuito ogni nodo si localizza in modo autonomo attraverso lo scambio di informazioni con i propri vicini. Ciò permette in genere una maggior velocità di localizzazione, riduzione del consumo dell'energia, perché le informazioni sono scambiate solamente tra vicini. Nel caso centralizzato si definisce un nodo, detto *sink*, solitamente con prestazioni maggiori degli altri nodi della rete, ed una fonte energetica superiore, che gli permette di elaborare in modo veloce le informazioni che tutti i nodi della rete gli inviano. Questo implica nella rete un numero maggiore di messaggi scambiati rispetto al caso distribuito, in quanto non tutti i nodi sono connessi direttamente al nodo sink, e di conseguenza sono costretti a scambiarsi le informazioni tra di loro per raggiungerlo. Aumenta così il consumo energetico e la latenza. Un altro fattore molto importante è la sicurezza. Negli algoritmi distribuiti le informazioni sono scambiate solamente con i vicini ed è il nodo stesso ad elaborare le proprie informazioni, mentre negli algoritmi centralizzati le

*Le reti *ad hoc* sono composte solo da stazioni o dispositivi wireless che comunicano in modo diretto senza l'uso di *access point* o qualunque tipo di connessione di rete via cavo

informazioni possono percorrere tutta la rete per raggiungere il nodo sink andando a discapito della riservatezza dei dati.

2.2 Classificazione degli algoritmi

Si possono classificare gli algoritmi di localizzazione in algoritmi che si basano su àncore o no, e in algoritmi concorrenti o incrementali.

Algoritmi con àncora assumono che un certo numero di nodi conoscano la loro posizione, grazie a misure manuali, modulo GPS o altri metodi di localizzazione. La localizzazione consisterà quindi nel ricavare, a paritre dai nodi *àncora*, la posizione dei restanti nodi di rete.

Algoritmi senza àncora al contrario dei precedenti, non dispongono di nodi localizzati a priori. L'unica informazione disponibile è in genere una stima più o meno accurata della distanza tra nodi vicini. Tali algoritmi usano quindi l'informazione sulla distanza locale per cercare di determinare la posizione dei nodi. Ogni sistema di coordinate non è unico, e può essere integrato in un altro sistema di coordinate in infiniti modi diversi, dipendenti da rotazioni, traslazioni e piegamenti.

Algoritmi incrementali inseriscono nell'insieme dei nodi localizzati un nodo alla volta, dopo averlo localizzato, come mostrato in figura 2.1. Di solito partono da tre o quattro nodi con coordinate assegnate. Quindi, si aggiunge un nuovo nodo all'insieme, calcolando le sue coordinate in base alla distanza dai precedenti nodi localizzati. Svantaggio di questi tipi di algoritmo sono la propagazione dell'errore di misura e la facilità di trovare soluzioni localmente ottime ma non globalmente.

Algoritmi concorrenti i nodi raffinano la stima della loro posizione in modo concorrente, cioè eseguono la loro attività in maniera contemporanea, come illustrato in figura 2.2. Questi algoritmi riescono in genere ad evitare il problema del minimo locale, perché continuano a minimizzare l'errore locale dappertutto, contrastando così l'accumulo di errore globale.

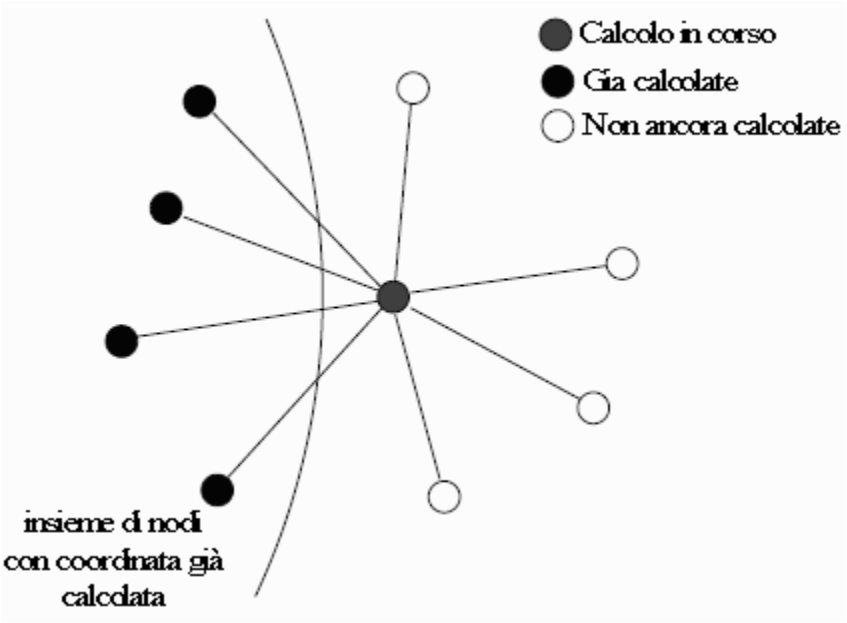


Figura 2.1: Nodi coinvolti in una tipica ottimizzazione incrementale [21]

2.3 Tecniche di localizzazione in letteratura

Volendo considerare le tecniche di localizzazione più note in letteratura, è opportuno segnalare le seguenti:

- Doherty, Pister e El Ghaoui [11], hanno studiato un algoritmo basato sulle ancore che utilizza solo vincoli di connessione tra i nodi àncora. L'algoritmo rappresenta le connessioni come un insieme convesso di vincoli di posizione, e usa un algoritmo centralizzato di programma-

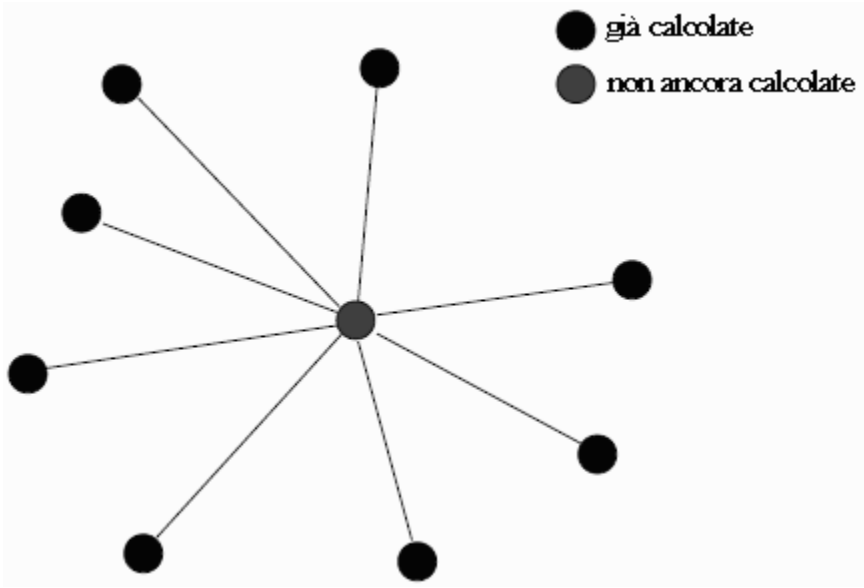


Figura 2.2: Nodi coinvolti in una tipica ottimizzazione concorrente [21]

zione lineare per risolvere la posizione del nodo. L'algoritmo non è facile da rendere scalabile e non è robusto agli errori di misura.

- Bulusu, Heidemann e Estrin [18], descrivono uno schema senza GPS che usa la connettività fra i nodi per ottenere una buona robustezza. Le coordinate dei nodi sono ottenute calcolando la media delle coordinate dei nodi àncora vicini. Questo è un algoritmo concorrente che non usa ottimizzazioni.

- Cricket [19] è un algoritmo studiato da Priyantha, Chakraborty e Balakrishnan per ambienti interni. Ogni nodo non localizzato si localizza usando un *listener* che raccoglie ed analizza le informazioni dei nodi àncora sparsi nell'ambiente. Il listener usa la differenza (TDoA) tra il tempo di arrivo del primo bit di un'informazione RF e il segnale ad ultrasuoni per determinare la distanza dal nodo àncora.

- L'algoritmo AHLoS [2] utilizza una tecnica iterativa che cerca di minimizzare l'errore quadratico fra le distanze misurate e quelle stimate

al fine di determinare la posizione corretta. Quando un nodo è stato localizzato, funge da nodo àncora per i nodi vicini non àncora localizzati. Questo metodo, chiamato *collaborative multilateration*, permette ai nodi che non hanno sufficienti nodi àncora vicini, di localizzarsi non appena uno dei propri vicini si è localizzato.

- L'algoritmo ABC [4] è un algoritmo incrementale che non usa ancore. ABC prima seleziona tre nodi nello stesso raggio di segnale e assegna loro le coordinate che soddisfano le loro interdistanze, quindi, in modo incrementale, calcola le coordinate degli altri nodi usando la distanza dai tre nodi con le coordinate già calcolate. Questo modo di procedere comporta la propagazione dell'errore, conseguenza del metodo iterativo.

- Terrain [4] è un algoritmo basato su ancore, costruito su ABC. Ogni nodo àncora esegue l'algoritmo ABC. Usando le coordinate determinate da ABC, ogni nodo calcola la distanza dai tre nodi àncora più vicini. Ogni nodo compie una ottimizzazione concorrente usando la distanza dai nodi àncora e le loro coordinate.

- Savarese, Rabaey e Langendoen, descrivono l'algoritmo Hop-Terrain [5]. Hop-Terrain è un algoritmo concorrente a due fasi che utilizza nodi àncora. Nella prima fase i nodi àncora propagano informazioni a tutta la rete. Tali informazioni sono utilizzate dai nodi non localizzati, per stimare la propria posizione iniziale. La seconda fase è di raffinamento, ovvero i nodi migliorano la stima della posizione corrente basandosi sulla stima della posizione precedente. Questo algoritmo è più robusto dell'algoritmo Terrain, in quanto è indipendente dal tipo di ranging.

- Niculescu e Nath [17], hanno proposto un algoritmo distribuito basato su nodi àncora che utilizza AoA (*Angle of Arrival*), con il quale è

possibile determinare l'angolo fra un nodo non localizzato e almeno tre nodi àncora.

- Howard, Mataric e Sukhatme [3] hanno studiato un algoritmo di localizzazione che utilizza il rilassamento basato sulle molle (*spring-base*). Nel loro sistema, i robot sono equipaggiati da un sensore di movimento che permette di misurare i cambi di posizione. La soluzione al problema di localizzazione si trova immaginando di costruire una rete di robot con una data posizione, connessi dalle molle, che si scambiano informazioni. Viene quindi eseguita in modo distribuito una procedura di rilassamento che porta alla minimizzazione dell'energia globale del sistema e quindi alla soluzione.

Capitolo 3

Algoritmo SNAFDLA

3.1 Motivazioni per SNAFDLA

L'algoritmo SNAFDLA (*Sensor Network Anchor Free Distributed Localization Algorithm*), proposto in questa tesi, è un algoritmo di localizzazione completamente distribuito, scalabile, robusto al rumore sul canale per reti di sensori piane. L'algoritmo sviluppato si classifica come un algoritmo senza àncore e concorrente.

3.2 Definizione del problema

Dato un insieme di nodi, con posizioni sconosciute, e un meccanismo con il quale ciascun nodo può stimare la distanza dai suoi vicini, si vuole determinare la posizione di ciascun nodo rispetto ad un sistema di riferimento comune, mediante la comunicazione locale da nodo a nodo. Ciò significa che non è necessario l'utilizzo di un protocollo di instradamento e la conoscenza del grafo di connessione tra nodi globale, in quanto ciascun nodo scambia messaggi solo con i propri vicini. Ciascun nodo stima la distanza dai suoi vicini come una trasformazione del segnale RSSI. Chiameremo tale distanza,

distanza misurata. Prima di procedere alla descrizione dell'algoritmo occorre introdurre la terminologia e le notazioni utilizzate. Una rete di sensori sia descritta da un grafo $G = (N, E)$ dove N è l'insieme dei nodi ed E l'insieme degli archi. Due nodi i e $j \in N$ si dicono vicini se esiste un arco $e_{(i,j)} \in E$ che li collega. Denotiamo con $d_{(i,j)}$ la distanza stimata dall'algoritmo tra il nodo i e il nodo j, mentre la distanza misurata sull'RSSI dal nodo i al nodo j sia identificata con $r_{(i,j)}$. Data una collezione di N nodi, e le distanze misurate di ogni nodo dai suoi vicini, l'obiettivo è produrre un insieme di punti $p_i = (x_i, y_i)$ coerenti con tutte le distanze misurate, cioè, dati i punti p_i e p_j, la distanza stimata tra i nodi i e j deve essere $\|p_i - p_j\| = d_{(i,j)}$ per ogni $e_{(i,j)} \in G$. Per alcuni grafi può accadere che una o più posizioni non siano uniche a parità di rotazione, traslazione e riflessione. Come mostrato in figura 3.1 [21] un grafo è rigido se non può flettersi preservando le distanze tra i nodi. Talvolta può succedere che anche se il grafo è rigido, potrebbe essere soggetto a piegamenti. Ad esempio, se ci sono due triangoli che hanno un lato in comune, un triangolo può essere riflesso dall'altro lato senza che nessuna distanza cambi. Questo tipo di grafo viene chiamato rigido, ma non globalmente rigido. Il grafo globalmente rigido preserva le distanze e non può essere piegato. Una garanzia della gloable rigidità del grafo della rete si può ottenere imponendo che ogni nodo della rete abbia almeno tre vicini non allineati. Per questo motivo, in questa trattazione, così come in altre in letteratura [21, 2, 11, 18, 4, 5], si assume che questo vincolo venga rispettato.

3.3 Descrizione di SNAFDLA

L'algoritmo SNAFDLA è un algoritmo distribuito senza ancore e concorrente. Il principio di funzionamento dell'algoritmo è il seguente: ogni nodo inizialmente trasmette in broadcast dei beacon in modo da permettere ai

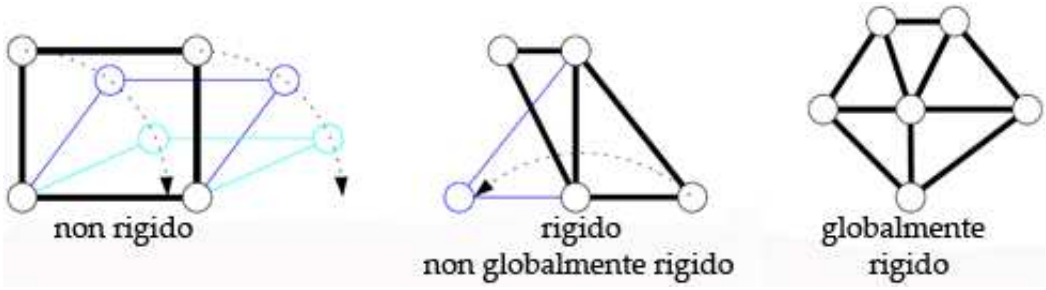


Figura 3.1: Esempio di grafo non rigido, non globalmente rigido, globalmente rigido [21]

nodi di vicini di stimare la distanza. Dopodiché si autolocalizzano tre nodi che fungeranno da àncora per i nodi non localizzati. I nodi vicini che hanno sufficienti nodi àncora possono stimare la propria posizione utilizzando le infomazioni sulla posizione dei nodi àncora e sulla distanza da essi. Si può immaginare il grafo di connessione dove i lati sono molle e quando la distanza che si misura, da un nodo ai suo vicini, è maggiore della distanza stimata, allora la molla tira i due nodi più lontani, viceversa se la distanza misurata è minore della distanza stimata la molla spinge i due nodi più vicini. Questo processo iterato più volte, per ciascun nodo, comporta un raffinamento della propria posizione che termina solo quando le molle, che agiscono sul nodo e i suoi vicini, sono in equilibrio. Quando tutti i nodi saranno in una situazione di equilibrio l'algoritmo termina. Consideriamo ora in dettaglio ciascuna fase dell'algoritmo.

3.3.1 1° Fase: Determinazione della connettività

In questa prima fase ogni nodo i trasmette in broadcast un *beacon*, cioè un messaggio che permette al ricevente di identificare il mittente. Ciascun nodo j che riceve il messaggio identifica il nodo i come vicino e mediante il valore del segnale RSSI ne determina la distanza misurata $r_{i,j}$.

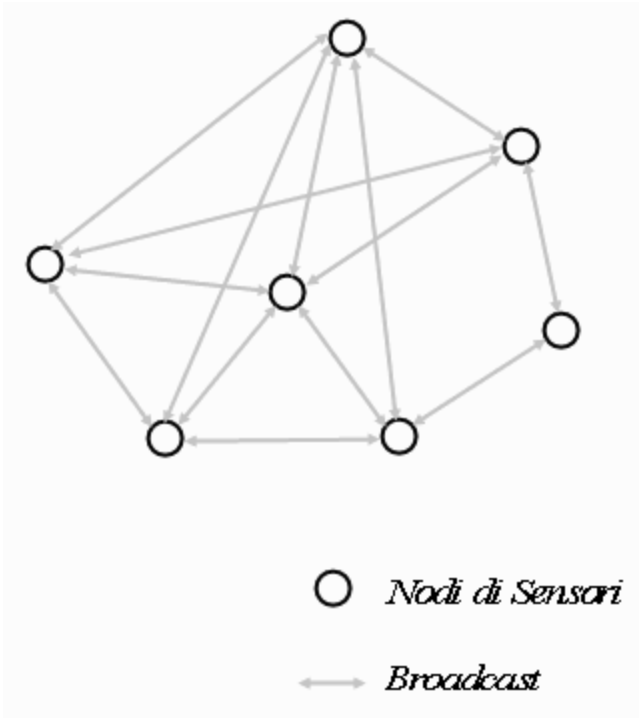


Figura 3.2: Determinazione della connettività con l'invio di messaggi broadcast

3.3.2 2° Fase: Localizzazione del triangolo iniziale

Ciò che si vuole raggiungere mediante l'esecuzione di questa fase è la determinazione delle coordinate iniziali di tre nodi della rete. Si seleziona un nodo i che inizializza le proprie coordiate, secondo il piano bidimensionale euclideo, a (0,0), ovvero l'origine degli assi. Successivamente il nodo i seleziona una nodo a, tra i suoi vicini, e gli trasmette la propria lista dei vicini. Il nodo a si identifica sul piano come ($d_{i,a}$,0), quindi legge la lista ricevuta dal nodo i e ricerca nella propria lista dei vicini un nodo b vicino in comune con il nodo i. Successivamente il nodo a trasmette l'identificatore del nodo b, al nodo i. Il nodo i determina le coordinate del nodo b utilizzando la legge dei coseni. Siano α, β, γ i tre angoli interni di un triangolo e i, a, b le lunghezze dei suoi lati. Se tre di queste misure sono conosciute, si possono

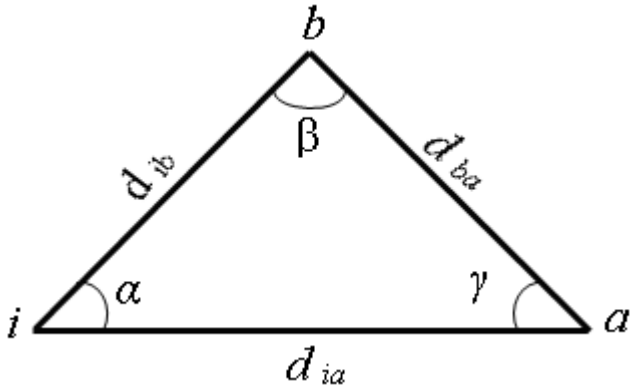


Figura 3.3: Geometria di localizzazione dei primi 3 nodi della rete

determinare le tre incognite. Utilizzando la legge dei coseni si ha:

$$d^2_{(a,b)} = d^2_{(i,a)} + d^2_{(i,b)} - 2d_{(i,a)}d_{(i,b)}cos(\alpha)$$

da cui

$$cos(\alpha) = \frac{d^2_{(i,b)} + d^2_{(i,a)} - d^2_{(a,b)}}{2 \cdot d_{(i,a)} \cdot d_{(i,b)}}$$

$$x_b = \frac{d^2_{(i,b)} + d^2_{(i,a)} - d^2_{(a,b)}}{2 \cdot d_{(i,a)}}$$

$$y_b = d_{(i,b)} \cdot \sqrt{1 - \Big(cos(\alpha)\Big)^2}.$$

Va osservato che il valore di y_b può essere determinato a meno del segno. La scelta arbitraria del segno positivo per y_b può avere come conseguenza che la soluzione che si ottiene sia *specularmente riflessa* rispetto a quella vera. Questo non viene però considerato un problema, perché l'eventuale riflessione può essere facilmente eliminata a posteriori.

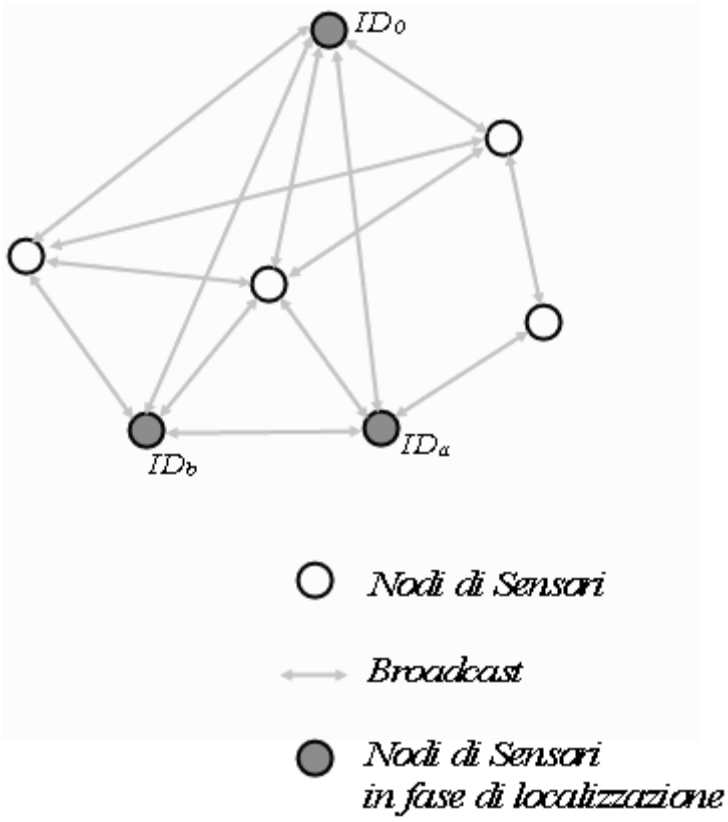


Figura 3.4: Localizzazione del triangolo iniziale

3.3.3 3° Fase: Localizzazione iniziale dei nodi

La fase precedente, rivolta a determinare la posizione iniziale di tre nodi, permette, di far evolvere la localizzazione a tutti i nodi nel grafo. Infatti ogni nodo può procedere alla autoinizializzazione della propria posizione quando possiede almeno tre vicini localizzati. Più formalmente, per ciascun nodo che ha almeno tre vicini localizzati, si procede alla costruzione e risoluzione di un sistema in cui vengono impostate le distanze da ciascuno dei vicini. In funzione del numero di vicini il sistema costruito può essere *determinato* o *sovradeterminato*, ovvero con più equazioni che incognite, che in tal caso sarà risolto mediante la soluzione nel senso dei *minimi quadrati*. Per la costruzione del sistema si utilizza l'informazione sulla distanza dal nodo i,

del quale si vuole stimare la posizione, al nodo $j, \forall j \in C_i$, dove C è l'insieme dei vicini localizzati del nodo i. Sia $d_{(i,k)}$ la distanza tra due punti, i e k:

$$d_{(i,k)} = \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2}$$

Costruiamo il sistema lineare sfruttando tale equazione che possiamo ri-

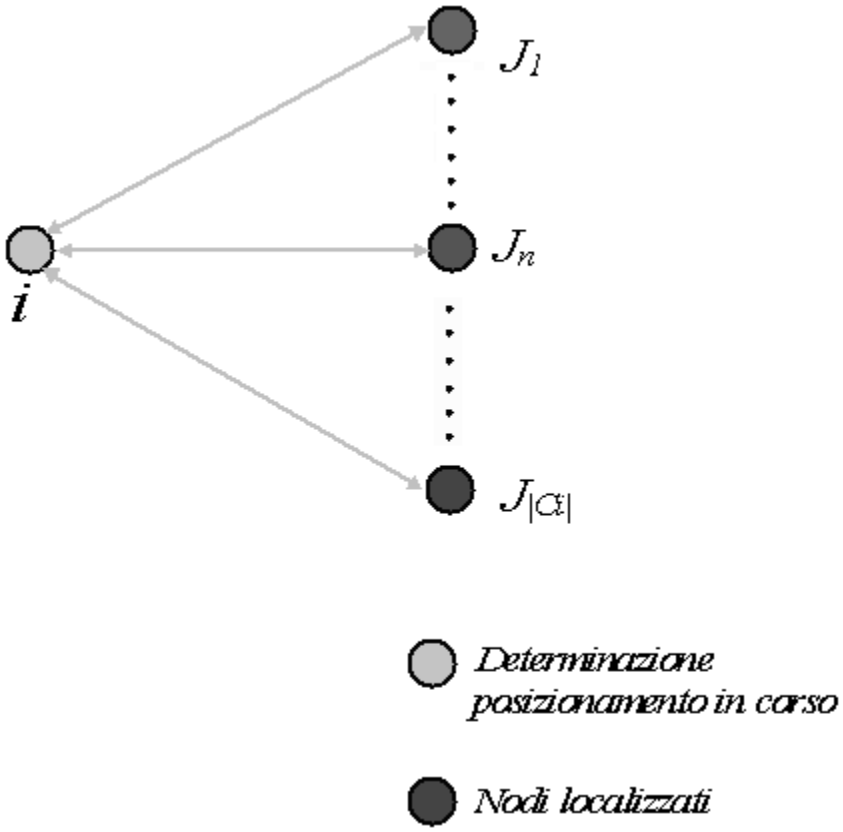


Figura 3.5: Informazione delle distanze dai vicini

scrivere, fissato un nodo $k \in C_i$, nel seguente modo:

$$d^2_{(i,k)} - x_k^2 - y_k^2 = x_i^2 - 2x_i x_k + y_i^2 - 2y_i y_k \tag{3.1}$$

Consideriamo due di queste equazioni, per due vicini, k e j:

$$d^2_{(i,k)} - x_k^2 - y_k^2 = x_i^2 - 2x_i x_k + y_i^2 - 2y_i y_k$$
$$d^2_{(i,j)} - x_j^2 - y_j^2 = x_i^2 - 2x_i x_j + y_i^2 - 2y_i y_j$$

Sottraendo membro a membro le equazioni, si ottiene

$$\frac{d^2_{(i,k)} - x_k^2 - y_k^2 - d^2_{(i,j)} + x_j^2 + y_j^2}{2} = x_i(-x_k + x_j) + y_i(-y_k + y_j)$$

Effettuando una ridenominazione otteniamo

$$\begin{aligned} b_j &= \frac{d^2_{(i,k)} - x_k^2 - y_k^2 - d^2_{(i,j)} + x_j^2 + y_j^2}{2} \\ a_{(j,1)} &= -x_k + x_j \\ a_{(j,2)} &= -y_k + y_j \\ b_j &= x_i a_{(j,1)} + y_i a_{(j,2)} \end{aligned}$$

Operando in questo modo per tutti i vicini del nodo i, otteniamo un sistema $S(x_i, y_i)$ di $|C_i| - 1$ equazioni.

$$S(x_i, y_i) = \Big\{ b_j = x_i a_{(j,1)} + y_i a_{(j,2)} \quad j = 2, ..., |C_i|$$

Possiamo vedere il sistema lineare in forma matriciale: chiamiamo A la matrice dei parametri $a_{(j,1)}, a_{(j,2)}$ dove $j = 2, ..., |C_i|$, b la matrice dei parametri b_j dove $j = 2, ..., |C_i|$ e con x indichiamo il vettore rappresentante le coordinate del nodo i, ovvero $x = (x_i, y_i)$. Possiamo così riscrivere con una simbologia più compatta il nostro sistema lineare come $Ax = b$ di $|C_i| - 1$ equazioni in 2 incognite.

$$Ax = b = \begin{pmatrix} a_{(2,1)} & a_{(2,2)} \\ a_{(3,1)} & a_{(3,2)} \\ \vdots & \vdots \\ a_{(|C_i|,1)} & a_{(|C_i|,2)} \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} b_2 \\ b_3 \\ \vdots \\ b_{|C_i|} \end{pmatrix} \tag{3.2}$$

Se il nodo i possiede solamente tre vicini localizzati, si determina un sistema lineare a 2 equazioni in 2 incognite, mentre se il nodo i possiede più di tre vicini localizzati, si ottiene un sistema lineare sovradeterminato. Il

seguente procedimento viene eseguito una sola volta per ciascun nodo. Una volta stimata la propria posizione, ciascun nodo la trasmette in broadcast.

L'algoritmo Anchor Free Localization [21], a differenza di SNAFDLA, elegge cinque nodi di riferimento, quattro dei quali sono ai margini estremi del grafo e uno circa al centro del grafo. Questo dovrebbe permettere di determinare un grafo iniziale senza piegamenti. Come risulta evidente, questo processo è estremamente dispendioso in termini di tempo, ed inoltre, comporta l'esecuzione di un algoritmo che ricerca il cammino di lunghezza massima per determinare gli estremi del grafo.

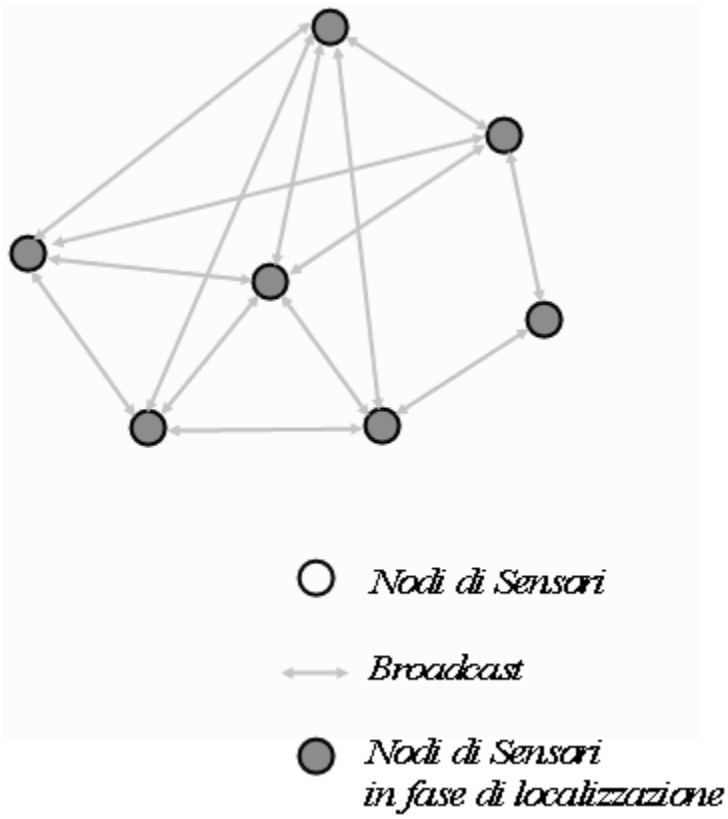


Figura 3.6: Localizzazione iniziale dei nodi

Localizzazione con tre vicini

Il sistema lineare determinato che si ottiene quando il nodo i ha solamente tre vicini localizzati, è il seguente:

$$\begin{aligned} Ax &= b \\ &= \begin{pmatrix} a_{(2,1)} & a_{(2,2)} \\ a_{(3,1)} & a_{(3,2)} \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} b_2 \\ b_3 \end{pmatrix} \end{aligned} \tag{3.3}$$

Per ottenere la soluzione del sistema (3.3) possiamo procedere al calcolo del determinante della matrice A e quindi alla soluzione del sistema mediante il *metodo di Cramer*:

$$detA = \begin{vmatrix} a_{(2,1)} & a_{(2,2)} \\ a_{(3,1)} & a_{(j_3,2)} \end{vmatrix} = a_{(2,1)}a_{(3,2)} - a_{(2,2)}a_{(3,1)}$$

$$\begin{aligned} x_i &= \frac{b_2 a_{(3,2)} - b_3 a_{(2,2)}}{detA} \\ y_i &= \frac{b_3 a_{(2,1)} - b_2 a_{(3,1)}}{detA} \end{aligned}$$

Localizzazione con più di tre vicini

Per $|C_i| > 3$, si hanno più equazioni che incognite. Siamo nel caso di un sistema lineare sovradeterminato; quindi una soluzione esatta, che soddisfa cioè tutte le equazioni, generalmente non esiste. Tale sistema può essere risolto nel senso dei minimi quadrati.

Soluzione nel senso dei minimi quadrati Date le triplette $(b_j, a_{(j,1)}, a_{(j,2)}), j = 2, ..., |C_i|$, si determinano i coefficienti (x_i, y_i) di una funzione

$$f(a_{(j,1)}, a_{(j,2)}) = x_i a_{(j,1)} + y_i a_{(j,2)}$$

Per cui:

$$\sum_{j=2}^{|C_i|} \epsilon_j^2 = \sum_{j=2}^{|C_i|} \Big[b_j - f(a_{(j,1)}, a_{(j,2)}) \Big]^2$$

sia minima. Gli ϵ_j sono chiamati residui. L'errore ϵ è dato da:

$$\epsilon(x) = \|Ax - b\|^2 \tag{3.4}$$

Trovare la soluzione del sistema (3.4) significa trovare quel vettore x tale che:

$$\epsilon(x) \leq \|Az - b\|^2, \forall z \in R^2 \tag{3.5}$$

Ovvero si ricerca quel vettore Ax che sia di minima distanza da b tra tutti i vettori Az. Siccome la norma euclidea utilizza la nozione di prodotto scalare e di angolo tra i vettori, si dimostra che, se vogliamo che $\|Ax - b\|^2$ sia minima, allora ricordando il concetto di ortogonalità tra vettori, ovvero $u \perp v \Leftrightarrow u^T v = 0$, è necessario che $Ax - b$ sia ortogonale ad A. Quindi possiamo scrivere:

$$\begin{aligned} A^T(Ax - b) &= 0 \\ A^T Ax &= A^T b \\ x &= (A^T A)^{-1} A^T b \end{aligned} \tag{3.6}$$

Effettuando la ridenominazione seguente $A^+ = (A^T A)^{-1} A^T$ otteniamo la matrice pseudoinversa di A chiamata anche pseudoinversa di *Moore-Penrose*. Il sistema ottenuto in (3.6) si risolve come esposto in precedenza.

3.3.4 4° Fase: Raffinamento iterativo della posizione dei nodi

Alla fine della fase precedente, ogni nodo della rete ha fatto una prima stima delle proprie coordinate. Se si è in presenza di errore le posizioni calcolate non sono consistenti con le distanze di tutto il grafo. Per questo è necessario a questo punto un raffinamento della localizzazione. Immaginiamo ogni lato $e_{(i,j)}$ nel grafo come una molla tra due masse, con una lunghezza a

riposo uguale alla distanza misurata tra i due nodi i e j. Siano (x_i, y_i) e (x_j, y_j) le coordinate stimate dei nodi i e j rispettivamente. Ricordiamo che chiamiamo distanza stimata tra i nodi i e j la distanza tra (x_i, y_i) e (x_j, y_j). Se la distanza stimata tra due nodi è minore della loro vera distanza (misurata), la molla induce una forza che respinge i due nodi, mentre, se la distanza stimata è maggiore della vera distanza, la molla attira i due nodi una verso l'altro. Questo schema viene chiamato *mass-spring model*. L'ottimizzazione segue essenzialmente il seguente metodo iterativo per ciascun nodo: ad ogni passo, i nodi si muovono in direzione della forza risultante determinata come la somma delle forza dai propri vicini. Ogni nodo termina la sua ottimizzazione quando la forza risultante che agisce su di lui è minore di un valore fissato.

Mass-spring optimization

In ogni momento, ogni nodo i possiede una stima corrente della propria posizione p_i. Periodicamente ciascun nodo i trasmette in broadcast la sua posizione, in modo che i vicini possano aggiornare l'informazione relativa alla posizione del nodo trasmettitore i. In questo modo, ogni nodo ha la stima aggiornata della propria posizione e di quella di tutti i suoi vicini. Usando queste informazioni, il nodo i calcola una distanza stimata $d_{i,j}$ da ogni vicino j. Inoltre il nodo i conosce la distanza misurata $r_{i,j}$ da ciascun vicino j.

Sia $\overrightarrow{v}_{i,j}$ il vettore unitario in direzione da p_i a p_j. La forza $\overrightarrow{F}_{i,j}$ indotta su i dalla molla tra i e j è data da

$$\overrightarrow{F}_{i,j} = \overrightarrow{v}_{i,j}(d_{i,j} - r_{i,j})$$

La forza risultante che agisce sul nodo i è la somma delle forze indotte dai

vicini di i

$$\overrightarrow{F}_i = \sum_j \overrightarrow{F}_{i,j}$$

L'energia $E_{i,j}$ presente tra i nodi i e j causata dalla differenza tra la distanza stimata e la distanza misurata è data dal quadrato della forza $\overrightarrow{F}_{i,j}$ e di conseguenza l'energia totale del nodo i è data da:

$$\begin{aligned} E_i &= \sum_j E_{i,j} \\ &= \sum_j (d_{i,j} - r_{i,j})^2 \end{aligned}$$

L'energia E_i di ciascun nodo i si riduce quando il nodo si muove di una piccola quantità in direzione della forza risultante $\overrightarrow{F}_i$. La condizione di diminuzione dell'energia viene considerata calcolando l'energia nella nuova posizione e controllando che sia minore rispetto alla posizione precedente. La condizione di non cadere in un minimo locale non è semplice da garantire perché dipende prettamente da una corretta inizializzazione del grafo. La quantità di cui un nodo deve muoversi è stata valutata sperimentalmente come $\frac{|\overrightarrow{F}_i|}{2 \cdot |C_i|}$. La convergenza viene raggiunta quando il valore di aggiornamento è inferiore ad un ϵ fissato a priori. Ovviamente più il valore di ϵ è piccolo, più i tempi di esecuzione dell'algoritmo aumentano perché si ricerca una misura della posizione sempre più precisa.

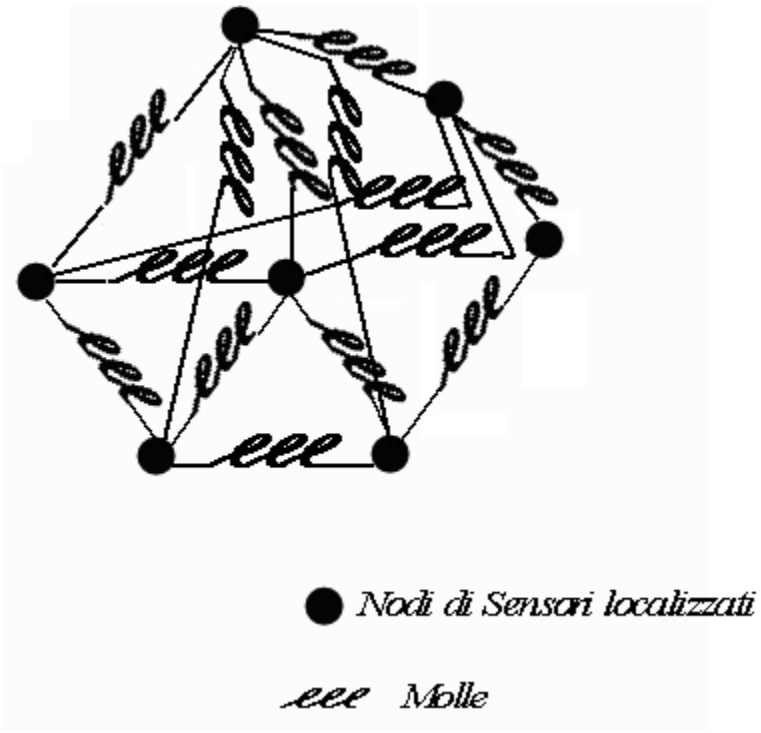


Figura 3.7: Raffinamento della posizione

Capitolo 4

Progettazione dell'algoritmo

Come descritto nel precedente capitolo, l'algoritmo di localizzazione è pensato per un sistema distribuito. La sua implementazione deve essere quindi condotta su una piattaforma hardware distribuita, possibilmente fra quelle impiegate nei nodi delle reti di sensori. Per questo l'algoritmo è stato codificato in linguaggio nesC, che è il linguaggio utilizzato per programmare i mote. nesC è basato sul C, ma con l'aggiunta di estensioni opportune per la gestione di vincoli di tempo reale e di concorrenza tra processi.

Va inoltre detto che l'ambiente di programmazione dei mote prevede l'utilizzo di nesC congiuntamente all'impiego di un particolare sistema operativo con caratteristiche real-time, espressamente studiato per tali piattaforme: TinyOS. Di fatto, in questo ambiente di programmazione nesC/TinyOs, il prodotto finale di compilazione è un unico codice eseguibile che contiene sia l'applicazione utente che la parte di sistema operativo necessaria al supporto dell'applicazione stessa. Una dettagliata descrizione del linguaggio è fornita nell'appendice C, mentre TinyOS è descritto nelle appendici B e A.

Per effettuare le simulazioni del sistema distribuito è stato adottato TOSSIM, un ambiente espressamente sviluppato per simulare sistemi di-

stribuiti che eseguono applicazioni nesC/TinyOS. TOSSIM, di fatto, genera tante istanze di esecuzione del codice quanti sono i nodi da simulare e li esegue in modo apparentemente concorrente, simulando lo scorrere del tempo.

Infine, per poter controllare ed utilizzare agevolmente il sistema di simulazione della rete, è stata sviluppata un'applicazione in Matlab, che fornisce all'utente un'interfaccia visuale intuitiva mediante la quale effettuare la programmazione degli esperimenti di simulazione e la gestione dell'intero sistema.

Il presente capitolo è quindi dedicato alla descrizione delle diverse fasi di implementazione ed analisi del sistema: l'implementazione dell'algoritmo di localizzazione su una rete di nodi di tipo Motes, la programmazione dell'ambiente TOSSIM (effettuata in linguaggio Phyton) per la gestione degli esperimenti di simulazione, lo sviluppo dell'infrastruttura del sistema e dell'interfaccia utente in Matlab per l'automatizzazione delle procedure di simulazione.

4.1 Sviluppo

L'algoritmo di autolocalizzazione illustrato al capitolo 3 viene implementato come applicazione TOS, scritta con il linguaggio *nesC*, da simulare mediante il simulatore TOSSIM. L'algoritmo è stato sviluppato seguendo il modello distribuito, dove ciascun nodo, che ora chiameremo mote, in funzione dei messaggi che scambia con i suoi vicini aggiorna il suo stato per raggiungere l'obiettivo di determinare la propria posizione rispetto ad un sistema di coordinate cartesiane. Per ovviare alla mancanza degli *acknowledge* nell'algoritmo, si utilizza la multiplazione a divisione del tempo (*Time Division Multiplexing*) ovvero una tecnica di condivisione del canale di comunicazione secondo la quale ogni mote può trasmettere in modo

esclusivo sul canale per un certo lasso di tempo chiamato *time slot*. Per non appesantire la spiegazione usiamo il termine componenti per indicare moduli (chiamati `NomeC`) configurazioni (chiamate `NomeAppC`) e interfacce (chiamate `Nome`). In figura 4.1 è mostrato il componente principale che gestisce l'evoluzione dei vari stati che assume il mote. L'algoritmo è stato sviluppato a moduli, dove ogni modulo rispetta un'interfaccia comune chiamata `Stato`. In questo modo è possibile cambiare una qualunque fase dell'algoritmo, e con pochi accorgimenti, ricollegarla alle altre. La prima fase dell'algoritmo, chiamata `PassoSetupAppC`, procede alla costruzione, per ciascun mote, di una tabella contenente le informazioni sulle distanze dai propri vicini. Nella simulazione questo primo passo si differenzia dalla realtà; utilizza infatti delle topologie di grafo generate esternamente. Data la struttura modulare dell'applicazione, è quindi semplice togliere tale modulo, che occorre alla simulazione, e ricollegare un nuovo modulo che utilizza solamente la potenza del segnale ricevuto, RSSI, per poter calcolare la distanza dal mote vicino e costruirsi dinamicamente la tabella dei vicini. Nella descrizione seguente si può parlare, senza perdere di generalità, di distanza come una trasformazione del valore RSSI. Una volta terminata la fase di inizializzazione, il mote ID_0 passa alla seconda fase dell'algoritmo, tutti gli altri passeranno alla terza fase dell'algoritmo, ovvero quella relativa alla ricerca della soluzione del sistema lineare. Il componente `Passo2AppC`, rappresentato in figura 4.2, implementa la seconda fase dell'algoritmo dove vengono determinate le coordinate iniziali di tre punti. Ricordiamo che il mote ID_0 inizializza le proprie coordinate a $(0, 0)$, seleziona poi il primo mote a lui vicino, che identifichiamo con ID_b, consultando la sua tabella dei vicini, imposta le coordinate di ID_b a $(d_{(0,b)}, 0)$, e gliele trasmette. Il mote ID_b aggiorna le proprie coordinate e invia la propria tabella dei vicini al mote ID_0. Il mote ID_0 ricercherà nella tabella dei vicini del mote ID_b un mote in comune con la propria tabella dei vicini. Una volta identifica-

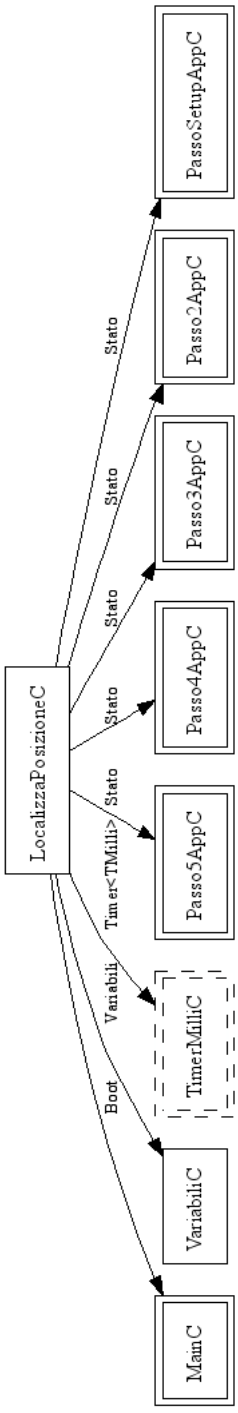


Figura 4.1: Schema di comunicazione tra i componenti (*configuration*)

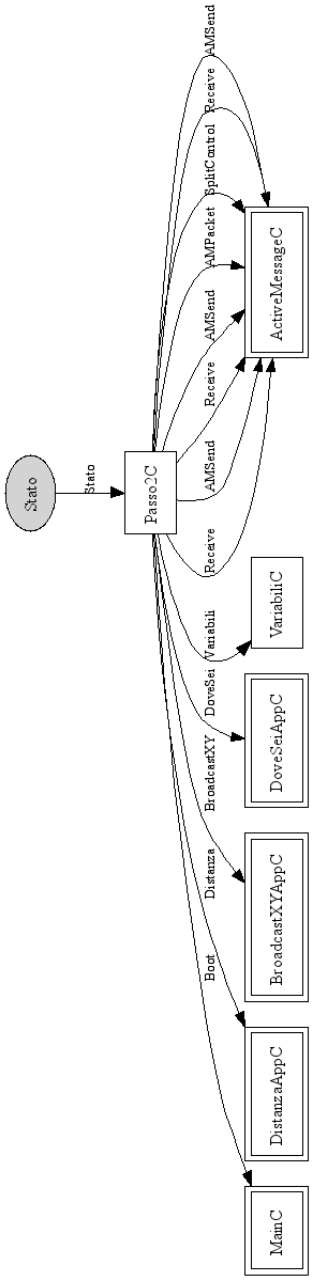


Figura 4.2: configuration Passo2AppC

to il mote in comune, che identifichiamo con ID_c, il mote ID_0 determina, con la legge dei coseni, le coordinate di ID_c, gliele trasmette e successivamente aggiorna la propria variabile di stato alla fase successiva dell'algoritmo. Ogni qualvolta un mote aggiorna le proprie coordinate, le trasmette in broadcast servendosi del componente `BroadcastAppC`. Provvediamo ora a dare una visione più approfondita dei componenti principali che entrano in gioco nella seconda fase. Il componente che gestisce i tipi di messaggio è `ActiveMessageC` con il quale possiamo impostare un identificatore per ciascun messaggio che si invia. In questo modo ogni volta che si riceve un nuovo messaggio, viene eseguita la funzione associata a tale identificatore di messaggio*. Il componente `VariabiliC` fornisce la gestione della tabella dei vicini, di informazioni legate allo stato del mote e allo stato della trasmissione dei messaggi. In questo modo, ogni fase dell'algoritmo può riferirsi a tale componente per ottenere le informazioni aggiornate in modo dinamico e prendere conoscenza se gli è concesso trasmettere un messaggio sul canale. Un componente fondamentale è `DistanzaAppC`. Tale componente aggiorna nella tabella dei vicini, la distanza che un mote ha dal proprio vicino, una volta ricevuto un messaggio. Tale componente, ogni volta che un mote deve aggiornare la distanza dal mote vicino, esegue la media rispetto ai valori di distanza precedenti per quel mote, apportando in questo modo una migliore stima della distanza. Questo procedimento di raffinamento della stima della distanza viene meglio precisato nella sezione 4.1.1. Terminata la fase di inizializzazione delle coordinate di tre mote e la ricezione da parte dei mote vicini delle posizioni aggiornate dei mote localizzati, tutti i mote con almeno tre vicini localizzati possono eseguire la terza fase dell'algoritmo, realizzata dal componente `Passo3AppC`, mentre i mote con un numero inferiore di vicini rimarrà in attessa di ricevere aggiornamenti di localizzazione

*Per un approfondimento sulla gestione dei messaggi e *Active Message* è possibile consultare la sezione B.3.4

dai mote vicini, che a seguito dell'esecuzione della terza fase otterranno. Il componente `Passo3AppC`, rappresentato in figura 4.3, utilizza il componente `MatriciAppC`, che si occupa di creare e risolvere il sistema lineare, sia esso determinato che sovradeterminato. Una volta risolto il sistema lineare il

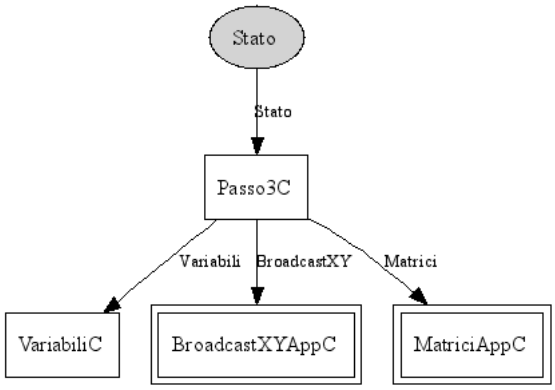


Figura 4.3: configuration Passo3AppC

componente imposta il suo stato alla fase successiva, ovvero la fase in cui, attraverso la *mass-spring optimization* si ricerca la posizione di convergenza del mote. Possiamo immaginare che il mote venga tirato come una molla dai suoi vicini, e ricerca con successivi aggiornamenti della sua posizione, che si basano sulla forza totale con cui i vicini lo tirano, una posizione di equilibrio in cui l'aggiornamento della posizione è inferiore ad un valore, ϵ, fissato. Il componente `Passo4AppC` è raffigurato in figura 4.4. Ogni volta che un mote aggiorna le proprie coordinate, come già menzionato sopra, provvede a comunicarlo via broadcast ai suoi vicini. Una volta che un mote si è autolocalizzato è necessario che tale mote continui a trasmettere la propria coordinata via broadcast, perché può essere che esistano dei vicini ancora non localizzati che necessitano di raffinare la stima della distanza dal mote per potersi autolocalizzare. Questa fase in cui si trasmettono le proprie coordinate una volta determinate, viene implementata dal componente `Passo5AppC`.

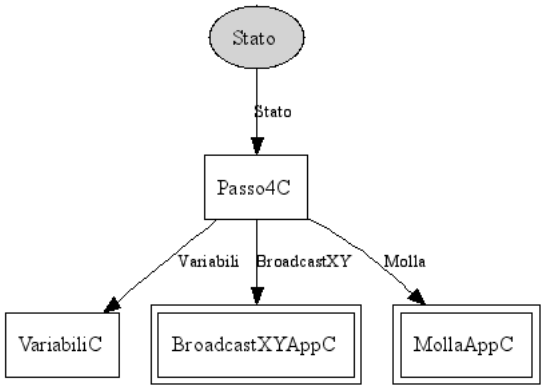


Figura 4.4: configuration Passo4AppC

4.1.1 Misurazione della distanza

La distanza misurata da ciascun mote quando riceve un nuovo messaggio, è composta dalla distanza reale più un rumore additivo generato con dei valori casuali distribuiti in modo uniforme. Più precisamente il rumore, ρ è pari alla moltiplicazione di un parametro, γ fissato a priori, per un numero, ϵ, generato casualmente con distribuzione uniforme definita tra $[-1, 1]$. La variabile casuale uniforme, ϵ, può assumere un qualsiasi valore nell'intervallo con uguale probabilità. Ci è quindi possibile decidere, a seconda della simulazione che vogliare fare, quale sia l'intervallo in cui i valori di rumore possono essere generati.

$$\rho = \gamma\epsilon \text{ con } \epsilon \in R, \epsilon \in [-1, 1]$$

La distanza misurata sarà quindi

$$\begin{aligned} d &= x + \rho \\ &= x + \gamma\epsilon \end{aligned}$$

La distanza stimata viene raffinata ogni volta che si riceve una nuova distanza misurata, ed è pari alla media di tutte le distanze misurate. Tale stima si perfeziona all'aumentare del numero di messaggi ricevuti. Questo

miglioramento può essere dimostrato come di seguito illustrato. Sia D la variabile casuale della distanza e il suo valore atteso pari a:

$$E(D) = x + \gamma \cdot E(\epsilon)$$

Il valore atteso, e varianza di una variabile casuale uniforme $X \sim U(a, b)$ sono:

$$E(X) = \frac{b+a}{2}$$
$$Var(X) = \frac{(b-a)^2}{12}$$

Nel nostro caso, il valore atteso della variabile casuale uniforme $\epsilon \sim [-1, 1]$ è:

$$E(\epsilon) = \frac{1-1}{2}$$
$$= 0$$

Abbiamo cosi dimostrato che il rumore ha media 0, di conseguenza non introduce una distorsione. Quindi il valore atteso della variabile casuale distanze sarà:

$$E(D) = x + \gamma \cdot E(\epsilon)$$
$$= x + 0$$
$$= x$$

ovvero la misura esatta della distanza. La varianza della distanza invece è:

$$Var(D) = \gamma^2 \cdot Var(\epsilon)$$

Calcolando la varianza di ϵ si ottiene:

$$Var(\epsilon) = \frac{(-1-1)^2}{12} = \frac{1}{3}$$

Quindi:

$$\begin{aligned} Var(D) &= \gamma^2 \cdot Var(\epsilon) \\ &= \gamma^2 \cdot \frac{1}{3} \end{aligned}$$

Fino adesso abbiamo parlato della variabile casuale distanza, ora vediamo cosa succede quando andiamo a calcolare valore atteso e varianza della distanza stimata. Sia D_n la variabile casuale stimata, ovvero la media delle distanze misurate, dove n rappresenta il numero di messaggi ricevuti.

$$D_n = \frac{1}{n} \sum_{i=1}^{n} x + \gamma \epsilon_i$$

Il valore suo valore atteso:

$$E(D_n) = \frac{1}{n} \sum_{i=1}^{n} x + \gamma \cdot \Big(E(\epsilon_i) \Big)$$

Dato che $E(\epsilon)$ abbiamo dimostrato essere 0 si ottiene:

$$\begin{aligned} E(D_n) &= \frac{1}{n} \sum_{i=1}^{n} x \\ &= \frac{n}{n} \cdot x = x \end{aligned}$$

Nuovamente il valore atteso della variabile casuale stima della distanza è la distanza corretta. Siccome le variabili casuali ϵ_i sono indipendenti e

identicamente distribuite possiamo dire che la varianza è uguale a:

$$\begin{aligned} Var(D_n) &= Var\left(\frac{1}{n}\sum_{i=1}^{n} x + \gamma\epsilon_i\right) \\ &= \frac{1}{n^2} Var\left(\sum_{i=1}^{n} x + \gamma\epsilon_i\right) \\ &\overset{iid}{=} \frac{1}{n^2}\sum_{i=1}^{n} \gamma^2 \cdot Var(\epsilon_i) \\ &= \frac{1}{n^2}\sum_{i=1}^{n} \gamma^2 \cdot \frac{1}{3} \\ &= \frac{n}{n^2} \cdot \gamma^2 \cdot \frac{1}{3} \\ &= \frac{\gamma^2 \cdot \frac{1}{3}}{n} \end{aligned}$$

Questo è proprio ciò che avevamo già intuito, ovvero all'aumentare dei messaggi ricevuti la stima migliora in quanto la varianza diminuisce.

$$\lim_{n\to\infty} \frac{\gamma^2 \cdot \frac{1}{3}}{n} = 0$$

4.2 Simulazione

L'ambiente di simulazione utilizzato si chiama TOSSIM. TOSSIM è un simulatore di eventi in modo discreto che simula applicazioni TinyOS. Funziona sostituendo alcuni componenti con componenti sviluppati apposta per la simulazione. Quando è in esecuzione preleva eventi dalla coda degli eventi e li esegue. TOSSIM può simulare a livello di comunicazione di pacchetti (*packet-level communication*) oppure a basso livello come chip radio per un ulteriore precisione dell'esecuzione del codice. A seconda del livello di simulazione scelto, gli eventi simulati potranno essere interrupt hardware oppure eventi di sistema. TOSSIM è una libreria per pezzo della quale si può scrivere la simulazione e la esegue. Il programma di simulazione può essere

scritto in Python oppure in C++. TOSSIM, come descritto nella sezione 4.1, non supporta misure di potenza del segnale. Per effettuare le nostre simulazioni, è stato modificato il simulatore affinchè si possa utilizzare un modello di trasmissione sul canale libero da errori. In questo modo abbiamo potuto valutare l'algoritmo secondo un nostro modello di rumore, descritto nella sezione 4.1.1. Abbiamo sviluppato un programma di simulazione che acquisisce, da un file esterno, la topologia della rete in modo che ciascun nodo sappia con chi può trasmettere.

4.3 Interoperabilità tra diversi sistemi

Molteplici e differenti sono i sistemi che devono operare insieme in modo dinamico per poter eseguire e valutare l'algoritmo proposto. Possiamo schematizzare in figura 4.5 come i diversi sistemi che andremo a descrivere interoperano tra loro. L'algoritmo è stato sviluppato per nodi di sensore mote che utilizzano TinyOs come sistema operativo. Tale algoritmo lo si vuole simulare in TOSSIM, un sistema di simulazione per mote che utilizza Python al fine di permettere all'utente di sviluppare script che permettono di interagire con l'appplicazione che si vuole simulare. TOSSIM così come la strumentazione necessaria per la compilazione e l'esecuzione di applicazione TinyOS necessità di un ambiente UNIX-like, nel nostro caso è stato scelto Cygwin. TOSSIM richiede la topologia della rete di sensori per poter simulare in modo distribuito l'algoritmo. Il nostro obiettivo è poter valutare l'algoritmo in diverse situazioni che vengono create in base al valore di certi parametri come il valore di picco dell'errore che agisce sulla distanza stimata, il raggio di comunicazione dei nodi e il numero di mote che costituiscono la nostra rete. I passi necessari per ottenere i risultati sono i seguenti:

- impostazione dei parametri

- generazione di una topologia di rete
- modifica dei sorgenti dell'algoritmo per configurarlo secondo i nostri parametri
- compilazione dei sorgenti
- generazione dello script python per la simulazione e configurazione di TOSSIM
- esecuzione della simulazione
- salvataggio dei dati acquisiti
- elaborazione dei dati
- valutazione dei dati

La maggior parte di questi passaggi al fine di valutare le prestazioni dell'algoritmo rendono l'esecuzione del test soggetto ad errori che possono essere dal malfunzionamento per un errata configurazione oppure per l'errata elaborazione dei dati. Si è voluto perciò rendere completamente automatico questo processo sviluppando un'applicazione Matlab che permette di scegliere i parametri di configurazione, il tipo di test che si vuole eseguire e quindi un'automatica elaborazione dei dati acquisiti interoperando tra i diversi sistemi. La figura 4.5 rappresenta l'interoperabilità tra i sistemi ad alto livello. Immaginiamo Matlab che, da un lato fornisce un'interfaccia utente visuale per il controllo del sistema e, dall'altro, comunica con Cygwin per poter simulare l'applicazione in TOSSIM il quale simula il sistema operativo TinyOS per il quale l'algoritmo è stato sviluppato. Ovviamente il canale di comunicazione è di input/output tra i livello perché come si richiede di eseguire un'operazione si ha la necessità di raccogliere i dati di output. In figura 4.6 viene rappresentata invece la complessa interoperabilità tra sistemi in modo più dettagliato.

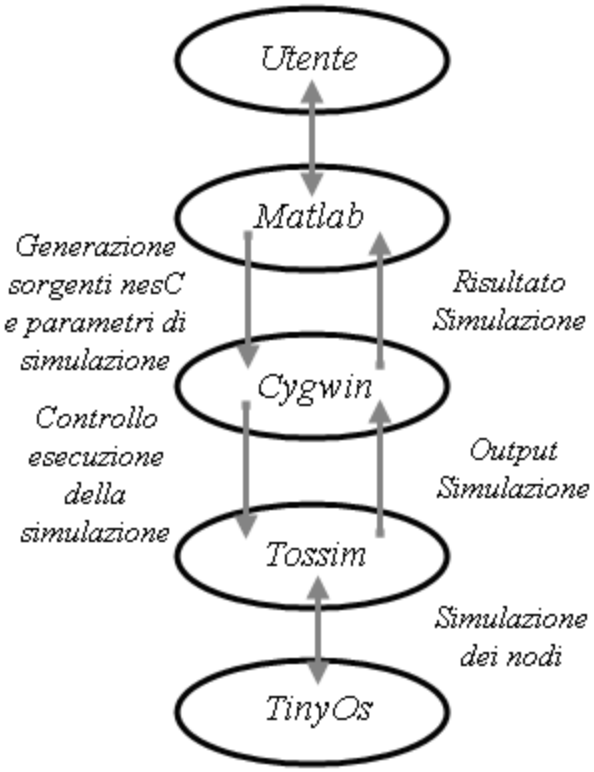


Figura 4.5: Interoperabilità tra sistemi - Alto livello

4.4 Testing

É stato necessario lo sviluppo di un ambiente per la generazione, esecuzione e sviluppo di test. I sistemi che devono interoperare tra loro sono molteplici e differenti e per questo è necessario rendere automatico l'ambiente di testing perché il tempo richiesto da operazioni manuali sarebbe estremamente lungo e soggetto ad errori data la vastità delle variabili in gioco che devono essere gestite. Per valutare il nostro algoritmo, dobbiamo conoscere la topologia di connessione dei mote in modo da poterlo confrontare con quella ottenuta dall'algoritmo. Per eseguire diversi test occorre poter variare diversi parametri per ciascun test che andremo a scegliere. Per questa ragione viene utilizzato *Matlab* un linguaggio per calcolo numerico, manipolazione e elaborazione di matrici. Mediante Matlab si genera in modo casuale un

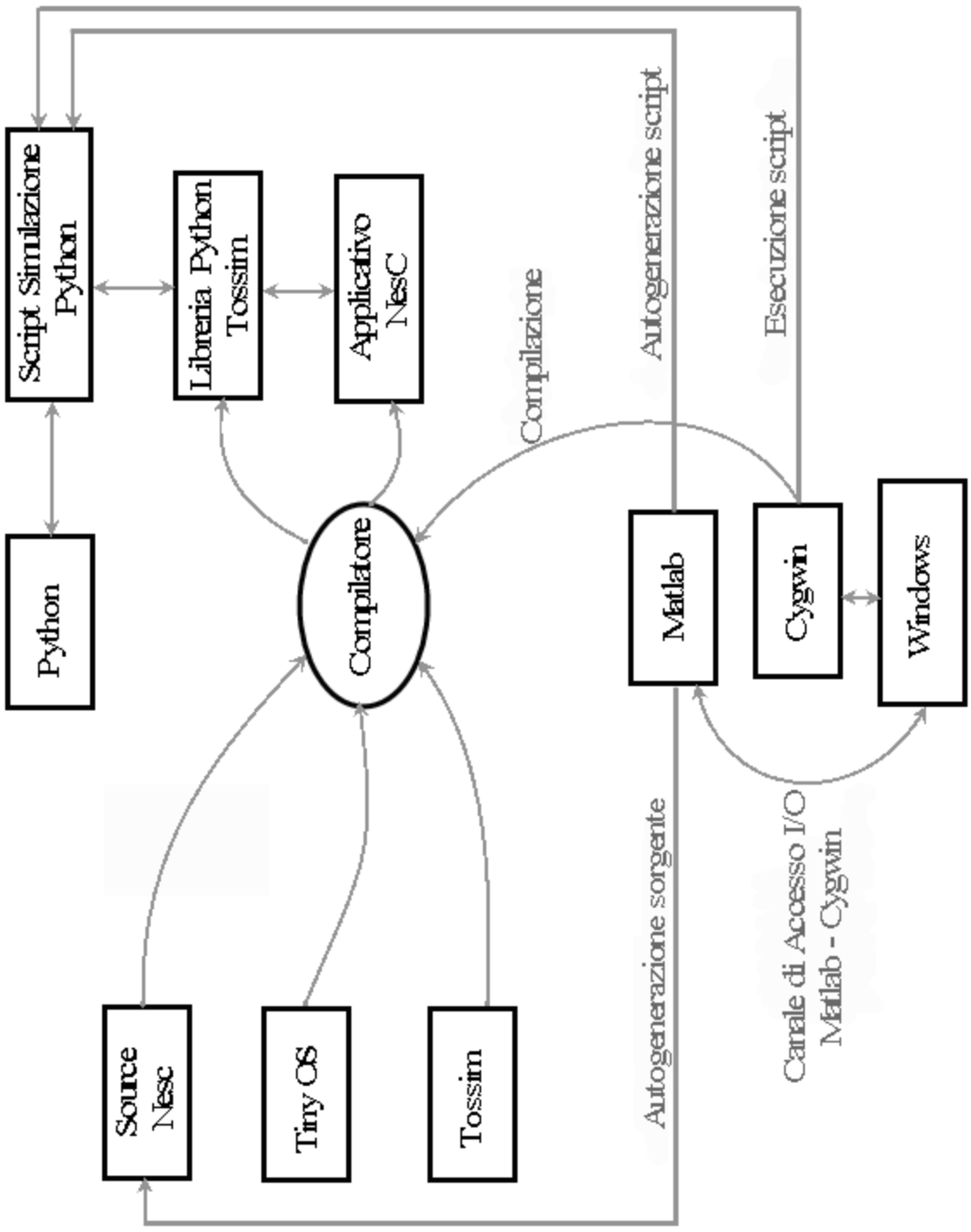


Figura 4.6:

Interoperabilità tra sistemi - Basso livello

grafo di n nodi con coordinate comprese tra $[0, 1]$. Le informazioni del grafo generate da Matlab vengono utilizzate per creare automaticamente dei file compatibili *nesC* che verranno successivamente compilati per TOS in un sistema *Cygwin*, sul quale verrà eseguita la simulazione dell'algoritmo. É stata inoltre sviluppata una GUI in Python per la visualizzazione grafica dell'evoluzione dell'algoritmo. In questo modo è possibile tener traccia visivamente della mutazione del grafo. Una volta terminata la simulazione, che può essere eseguita sia mediante GUI che in background rispetto all'applicazione principale in Matlab, i dati risultanti della simulazione vengono salvati da un programma in Python collegato con il programma di simulazione in Python per TOSSIM. Tali dati vengono successivamente importati nell'ambiente Matlab per essere valutati con strumenti di analisi numerica. In Matlab è stata sviluppata una GUI che implementa diversi tipi di test con parametri definibili dall'utente. Ogni test prevede ad eseguire il percorso logico di creazione, integrazione e quindi l'ottenimento dei risultati descritto precedentemente. Un'altro aspetto importante è la trasformazione dei dati risultanti dall'esecuzione dell'algoritmo per poterli confrontare con gli originali. Il grafo originale, generato con Matlab, pone un mote nell'origine e genera in modo casuale le coordinate degli altri mote. Il grafo risultante potrebbe avere subito una riflessione, traslazione e rotazione rispetto al grafo originale. É quindi necessario ricercarne i valori che rendono minima la somma delle interdistanze del grafo risultante dall'algoritmo rispetto a quello originale. Per verificare se c'è stata traslazione, rotazione oppure riflessione si procede nel modo qui di seguito descritto. Denoteremo le coordinate del nodo i indifferentemente con $< x_i, y_i >$ o con $V_{Matlab}(ID_i)$ e denoteremo con $< x_i^M, y_i^M >$ o $V_{Mote}(ID_i)$ le coordinate del nodo i determinate dall'algoritmo SNAFDLA.

Traslazione: sapendo che il grafo generato da Matlab pone il mote con ID_0 nell'origine degli assi, si calcola la traslazione che vi è con le coordiante del mote ID_0 trovate da SNAFDLA nel modo seguente. Per determinare la traslazione dei vettori coordinate risultanti dall'algoritmo rispetto ai vettori coordinate originali è sufficiente calcolare il vettore traslazione

$$\begin{aligned} V_T &=< tx, ty > \\ &= V_{Matlab}(ID_0) - V_{Mote}(ID_0) \\ &=< 0,0 > - < x_0^M, y_0^M > \\ &= - < x_0^M, y_0^M > \end{aligned}$$

e sommarlo a ciascun vettore coordinata $V_{Mote}(ID_i)$ con i appartenente all'insieme dei nodi.

$$\widehat{V_{Mote}}(ID_i) = V_{Mote}(ID_i) + V_T$$

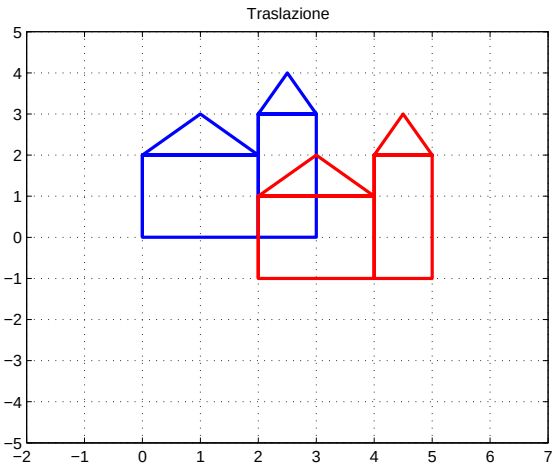


Figura 4.7: Esempio traslazione

Riflessione: si considerino tre mote, ID_i, ID_j, ID_k, e i vettori $V_{Matlab}(ID_i, ID_j)$, $V_{Matlab}(ID_i, ID_k)$, $V_{Mote}(ID_i, ID_j)$ e $V_{Mote}(ID_i, ID_k)$ $\forall i, j$ dove $V_{Matlab}(ID_i, ID_j) = V_{Matlab}(ID_j) - V_{Matlab}(ID_i)$ e analogamente $V_{Mote}(ID_i, ID_j) = V_{Mote}(ID_j) - V_{Mote}(ID_i)$. Calcoliamo il prodotto vettoriale

$$\widehat{V_{Matlab}} = V_{Matlab}(ID_i, ID_j) \times V_{Matlab}(ID_i, ID_k)$$
$$\widehat{V_{Mote}} = V_{Mote}(ID_i, ID_j) \times V_{Mote}(ID_i, ID_k)$$

Se $\widehat{V_{Matlab}} \cdot \widehat{V_{Mote}}$ è minore di 0 significa che il grafo risultante da SNAFDLA ha subito una riflessione rispetto all'originale. Per poter eliminare la riflessione dal grafo risultante, bisogna semplicemente moltiplicare per -1 x_i^M o y_i^M $\forall i$ appartenente all'insieme dei nodi.

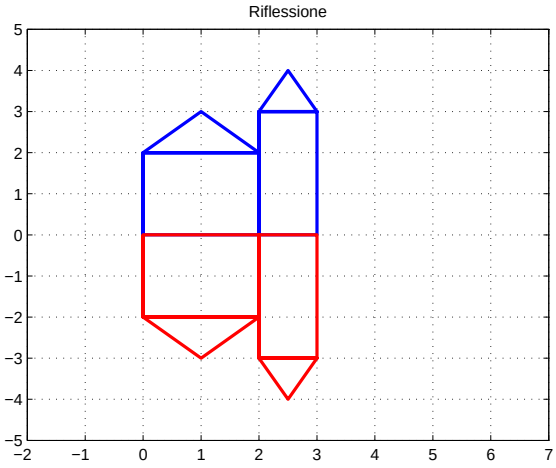


Figura 4.8: Esempio di riflessione

Rotazione: si considerino due mote, ID_i, ID_j. Si calcoli l'arco tangente dei vettori $V_{Matlab}(ID_i, ID_j)$ e $V_{Mote}(ID_i, ID_j)$ tenendo in considerazione il segno di entrambi gli argomenti per determinare il quadrante del risultato. L'angolo di rotazione del grafo risolto da SNAFDLA rispetto a quello generato da Matlab è pari a

$$\alpha = \theta_{Matlab} - \theta_{Mote}$$

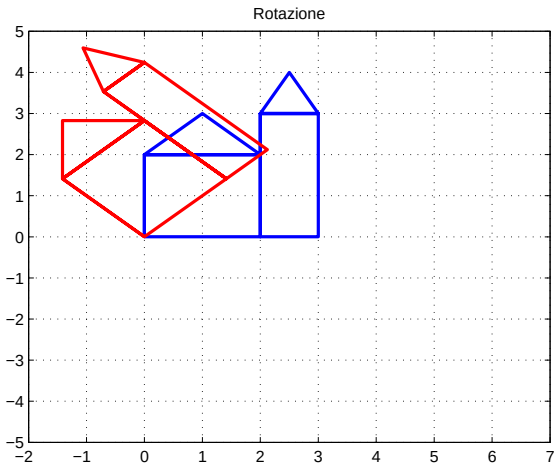


Figura 4.9: Esempio di rotazione

Calcolato α è possibile ruotare il grafo risultante dall'esecuzione di SNAFDLA, trasformando ciascun vettore coordinata nel modo seguente

$$\widehat{V_{Mote}}(ID_i) = \begin{pmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

$\forall i$ appartenente all'insieme dei nodi.

La figura 4.10 rappresenta una composizione delle trasformazioni precedentemente descritte.

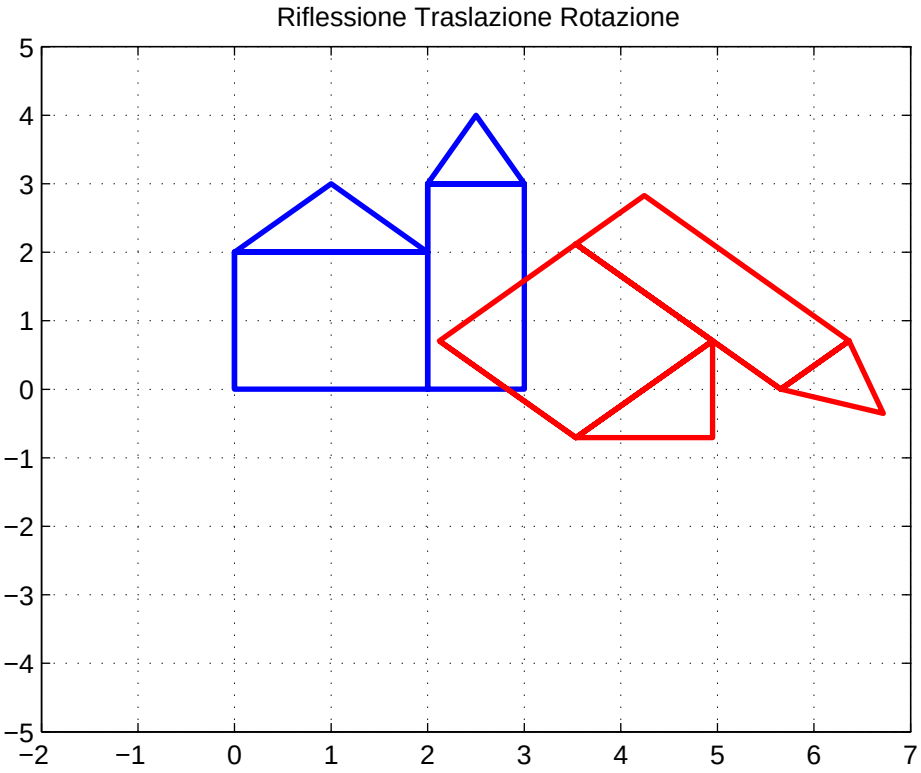


Figura 4.10: Esempio di Riflessione Traslazione Rotazione

Capitolo 5

Analisi

In questo capitolo si presentano gli esperimenti di simulazione effettuati allo scopo di valutare le prestazioni dell'algoritmo SNAFDLA. Le prove e le valutazioni sono state condotte con l'ausilio dell'ambiente di testing sviluppato appositamente.

5.1 Descrizione della procedura di analisi

Gli algoritmi di localizzazione sono valutati generalmente confrontando la soluzione delle posizioni stimate dall'algoritmo con quelle dei nodi appartenenti al grafo originale. Dette *interdistanze* le distanze tra la posizione stimata del nodo e quella originale, l'ambiente di testing ci permette di calcolare tali interdistanze ed effettuare prove ripetute della simulazione dell'algoritmo su una certa topologia di grafo, ogni volta ripetute con nuove realizzazioni dell'errore sulle distanze. Gli elementi e i parametri che caratterizzano ciascuna sessione di prove ripetute sono i seguenti:

- **Topologia del grafo**: le coordinate bidimensionali dei nodi del grafo sono generate casualmente nell'intervallo $[0, 1]$ con distribuzione uni-

forme. Questo implica che il grafo sia contenuto in un quadrato di lato 1 e che la massima distanza tra nodi sia pari a $\sqrt{2}$

- **Numero di nodi**: rappresenta la quantità di nodi che compongono il grafo

- **Range**: indica il raggio della circonferenza entro la quale il nodo può comunicare con i suoi vicini. Questo parametro è utilizzato nella generazione della topologia del grafo come valore di distanza massima entro quale i nodi possono comunicare.

- **Valore di picco dell'errore**: è il fattore moltiplicativo γ che viene applicato a variabili casuali generate uniformemente nell'intervallo $[-1, 1]$ per generare le realizzazioni del rumore additivo sulla distanza, che quindi è compresa nell'intervallo $[-\gamma, \gamma]$

L'ambiente di testing, mostrato in figura 5.1, una volta configurati i parametri sopra descritti, rende automatica la generazione di file sorgenti nesC necessari all'algoritmo per impostarsi in base ai valori inseriti, la compilazione e l'esecuzione della simulazione della prova. Questa procedura viene reiterata per il numero di prove che si vogliono effettuare.

5.2 Valutazione della robustezza e dell'errore sulle distanze

L'obiettivo di questo esperimento è valutare il limite di *robustezza* dell'algoritmo, cioè il valore massimo di errore sulle distanze oltre il quale l'algoritmo non è più affidabile. L'esperimento consiste nell'effettuare prove ripetute, fissato il numero di nodi e il range, e facendo variare il valore di picco dell'errore.

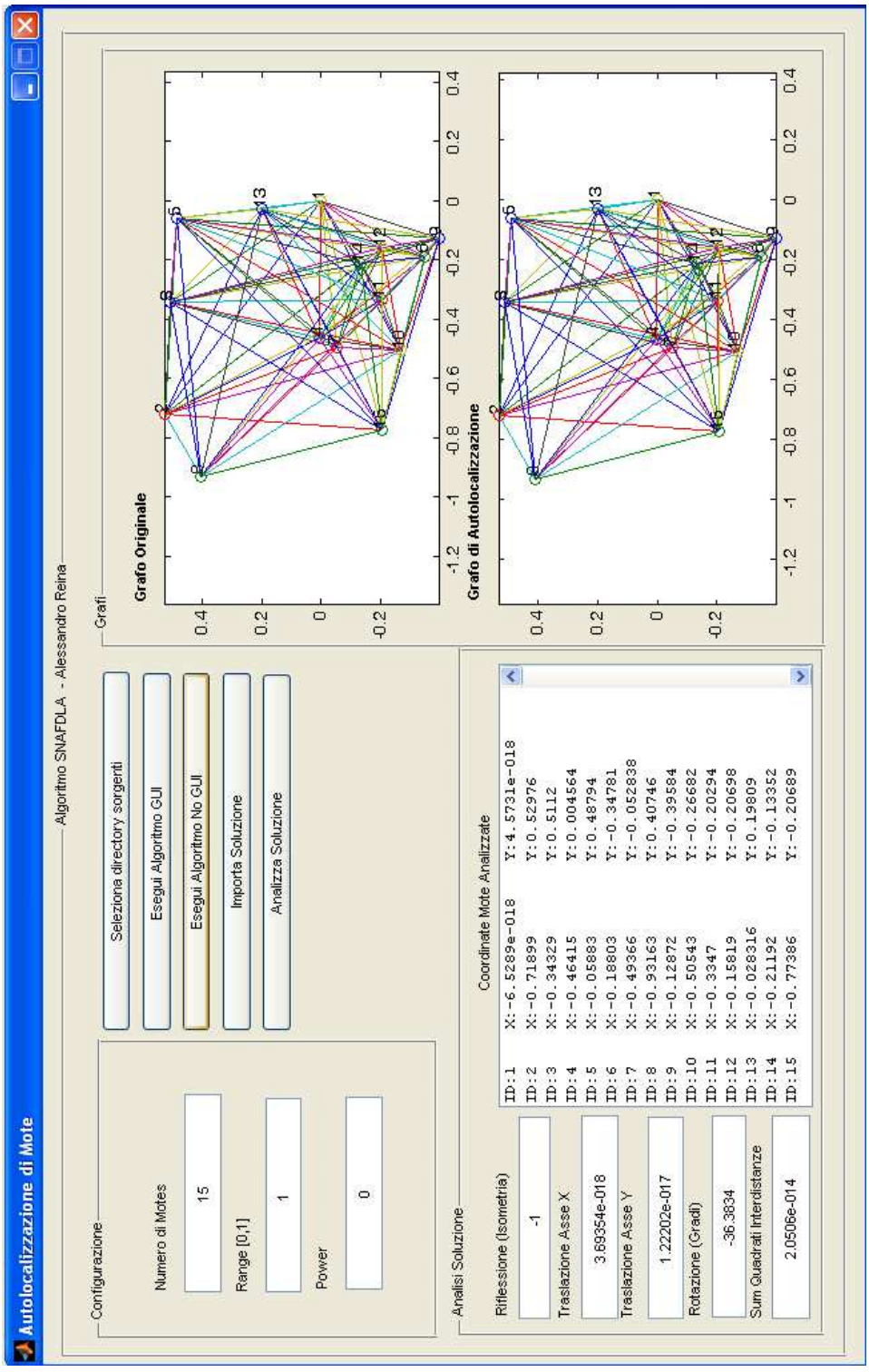


Figura 5.1: Ambiente di testing

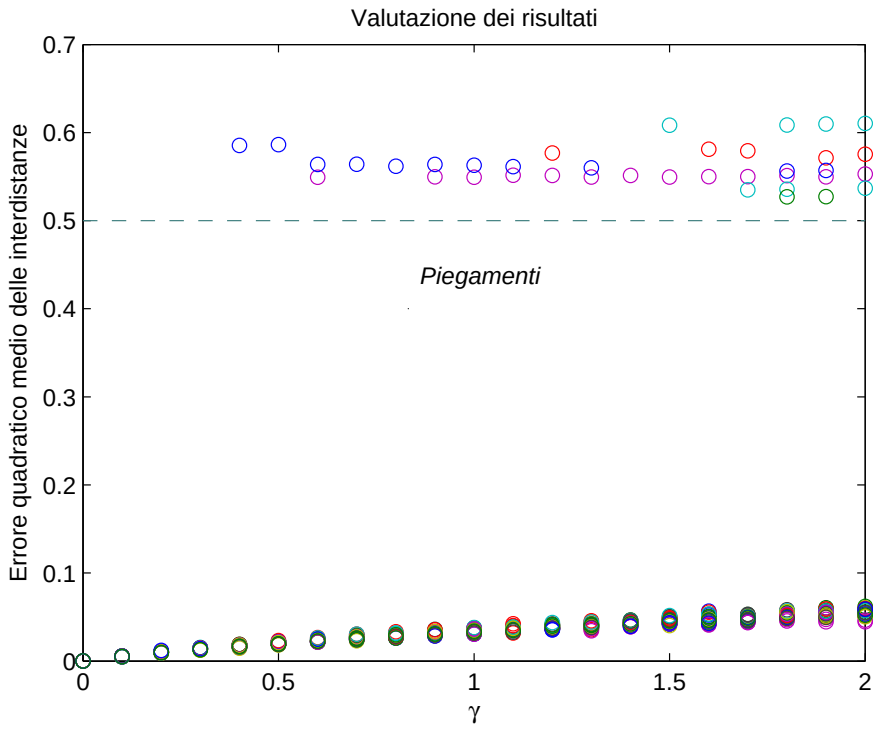


Figura 5.2: Valutazione della robustezza

Numero di prove	**Numero di nodi**	**Range**	**Errore γ**
30	128	0.7	da 0 a 2 passo 0.1

I risultati sono stati rappresentati graficamente in figura 5.2 Sul grafico individuiamo due distribuzioni dei valori. Nella parte *alta* del grafico si vedono i casi di insuccesso, dove l'algoritmo ha trovato una soluzione che, rispetto al grafo originale, presenta piegamenti causati dalla topologia del grafo. Osserviamo che i piegamenti iniziano a verificarsi dopo un valore limite di γ di circa 0.5 ed aumentano al crescere di γ, pur essendo sempre

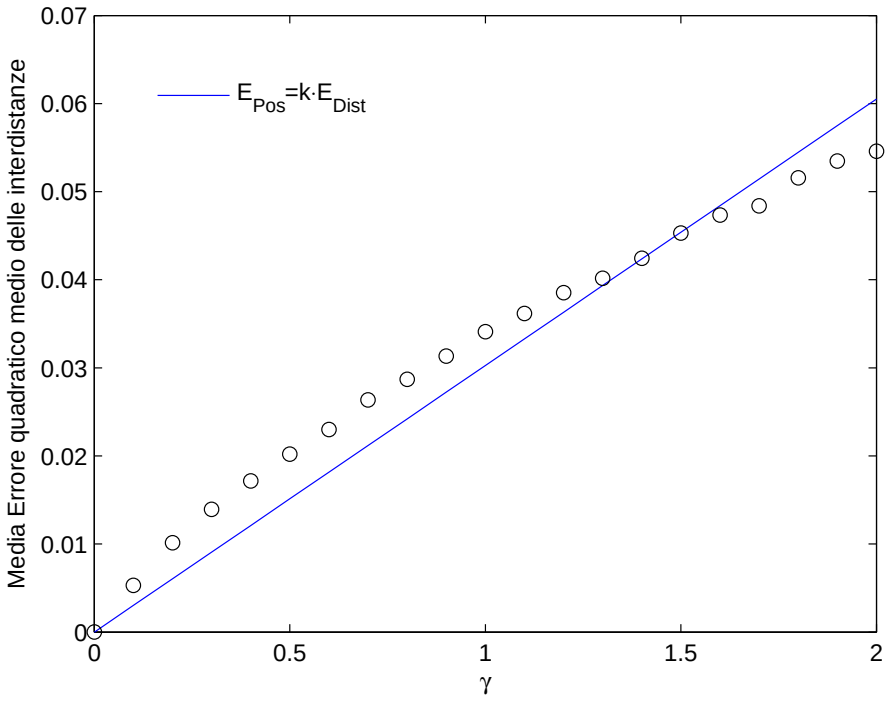


Figura 5.3: Medie degli errori di posizione in funzione di γ

radi rispetto alla maggioranza dei valori che sono concentrati nella parte bassa del grafico. Dopo questa valutazione possiamo definire il limite di robustezza con γ pari a 0.5 dipendente dalla configurazione assegnata. Nella parte *bassa* del grafico intuiamo un comportamento lineare degli errori di posizione in funzione degli errori sulle distanze. Analizziamo meglio questo comportamento andando a rappresentare graficamente, come mostrato in figura 5.3, le medie per ciascun valore assunto da γ degli errori di posizione. Definiamo l'errore delle posizioni come E_{Pos} e l'errore sulle distanze come E_{Dist} e spieghiamo l'andamento lineare come:

$$E_{Pos} = k \cdot E_{Dist}$$

Definiamo k il *fattore di abbattimento* dell'errore. Quando il valore k è minore di 1 l'errore sulle posizioni viene attenuato rispetto agli errori sulla

distanza. Come si vede dal grafico 5.3, nella prova abbiamo ottenuto un fattore di abbattimento di circa $\frac{1}{30}$ dell'errore sulle distanze.

5.3 Dipendenza dalla densità della rete

In questo esperimento si vuole valutare la prestazione dell'algoritmo facendo variare il numero di nodi, e fissando il valore di picco dell'errore e il range. L'esperimento è caratterizzato dai seguenti parametri:

Numero di prove	**Numero di nodi**	**Range**	**Errore γ**
60	da 20 a 120 passo 5	0.8	0.3

I risultati dell'esperimento sono rappresentati in figura 5.4. Si può osservare che nella parte *alta* ci sono alcuni casi di insuccesso causati dai piegamenti, come descritto in precedenza, i quali, inizialmente numerosi, a partire da un numero di mote pari a 40 si assestano su un valore minimo compreso tra 0 e 3. Questo significa che all'aumentare del numero di nodi, e quindi della densità della rete, il miglioramento apportato è minimo da non incidere su una ulteriore decisa diminuzione dei casi di insuccesso. Osservando più in dettaglio la parte bassa del grafo, evidenziata nel particolare in figura 5.5, notiamo come i risultati migliorano con l'aumentare dei nodi, diminuendo sia la media che la dispersione degli errori di posizione. Ciò significa che, maggiore è il numero di nodi (a pari dimensione dell'area che li contiene), maggiore è la loro densità e quindi maggiore sarà il numero di vicini di ogni nodo. Ogni nodo ha quindi a disposizione una maggior quantità di informazioni indipendenti per effettuare la stima della propria posizione, il

che abbassa la varianza della stima.

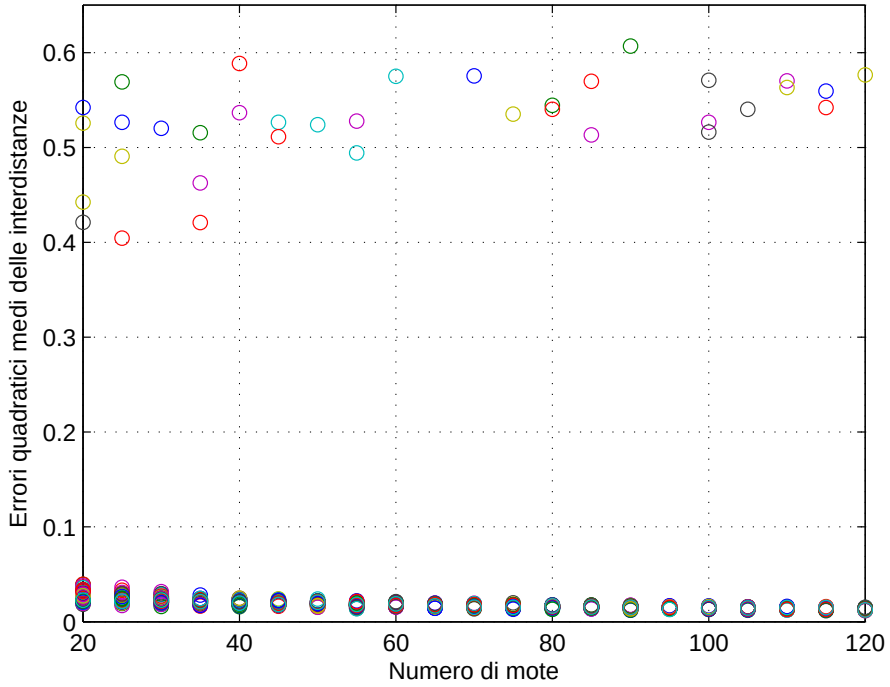


Figura 5.4:

Distribuzione degli errori di posizione al variare del numero di nodi

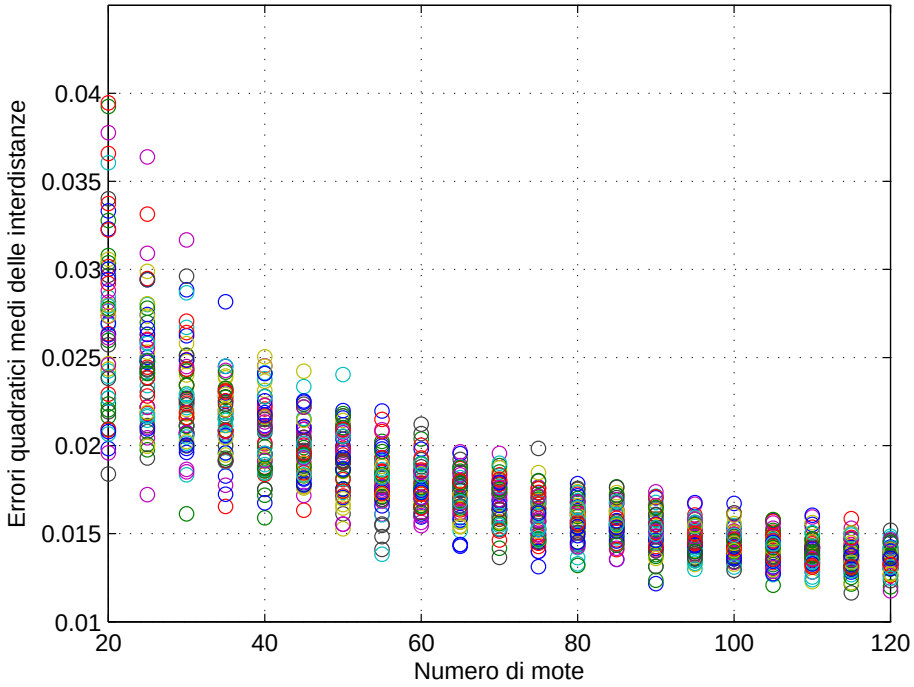


Figura 5.5:

Dispersione degli errori delle posizioni al variare del numero di nodi

Capitolo 6

Conclusioni

Nella presente tesi è stato condotto lo studio, lo sviluppo, la simulazione e l'analisi delle prestazioni di un algoritmo distribuito di localizzazione per reti di sensori, i cui nodi siano in grado di valutare approssimativamente la distanza dai propri vicini.

L'algoritmo è stato codificato in nesC, un linguaggio di programmazione di dispositivi per reti di sensori (TMote SKY), in modo da poter essere non solo simulato, ma in un prossimo futuro anche installato sui dispositivi, in modo da poter provare la procedura di localizzazione sul campo.

Programmando opportunamente l'ambiente di simulazione (TOSSIM) di tali dispositivi, sono state organizzate le sessioni di simulazione, al variare di vari parametri di rete quali la densità di rete e le caratteristiche dell'errore sovrapposto alle distanze. Mediante tale strumento sono state condotte sessioni di prove ripetute, mediante le quali l'algoritmo è stato caratterizzato ed analizzato in termini di prestazioni.

Per agevolare la gestione e la programmazione dell'ambiente di simulazione per la generazione degli esperimenti, è stata sviluppata una pratica interfaccia-utente visuale utilizzando MATLAB. Tale interfaccia costituisce un valido strumento che permette di utilizzare l'infrastruttura di simulazione

per progettare ed effettuare altri esperimenti di simulazione.

Confrontato con la letteratura, l'algoritmo sviluppato presenta il notevole vantaggio di essere completamente distribuito, non necessitando mai di informazioni globali sulla rete. Algoritmi simili [21], ad esempio, necessitano di una fase iniziale molto gravosa, per raccolta di informazioni di topologia della rete.

Numerosi sono i possibili sviluppi futuri del presente lavoro. Innanzitutto, sfruttando il fatto che il codice, così come è scritto, può essere direttamente compilato ed installato sui dispositivi, si potrebbe valutare sperimentalmente, con prove sul campo, il funzionamento della localizzazione su un sistema reale. Tali prove sperimentali, oltre alla verifica della simulazione, permetterebbero di raccogliere dati relativi alla reale distribuzione statistica dell'errore di stima della distanza fra i nodi. Un altro aspetto di notevole interesse sarebbe lo studio di una modifica dell'algoritmo, in modo tale da renderlo in grado di riconoscere i *piegamenti* del grafo, che danno luogo a soluzioni erronee per parte del grafo. Per fare questo, però, è necessario ripensare la strategia di comunicazione tra nodi, poichè probabilmente la sola comunicazione tra i diretti vicini non è più sufficiente.

Appendice A

TinyOS

TinyOS è un sistema operativo *open-source* progettato per *wireless sensor networks*. Si caratterizza per un'architettura basata sui componenti che permette una rapida innovazione ed implementazione minimizzando la dimensione del codice per andare incontro al vincolo della scarsità di risorse presenti nei dispositivi utilizzati nelle reti di sensori. Le librerie dei componenti di TOS includono protocolli di rete, servizi distribuiti, driver dei sensori, e tools che permettono l'acquisizione dei dati. Tutti questi componenti possono essere usati e ridefiniti da applicazioni personalizzate. TOS ha un modello di programmazione adatto per applicazioni *event-driven* che occupano poco spazio. Il linguaggio di programmazione utilizzato per lo sviluppo delle applicazioni per TOS, e anche per la maggior parte di TOS stesso, è *nesC*.

Appendice B

TinyOS 2.x

B.1 Punti di forza

TOS 2.0 rispetto alla versione precedente cerca di portare numerosi miglioramenti: dalla semplicità di utilizzo ad una attenta ingegnerizzazione del sistema operativo. Si pone particolare attenzione alla portabilità delle applicazioni su *diverse piattaforme* favorendo *robustezza* e *flessibilità* di utilizzo. Analizzando in modo particolare i punti di forza:

- **Diverse Piattaforme**: vengono implementate numerose piattaforme in modo semplice ed efficiente grazie al *Hardware Abstraction Architecture*.

- **Flessibilità**: grazie alla sua natura *open-source* ogni sviluppatore, gruppo di ricerca, ha a sua completa disposizione, il codice sorgente al quale è possibile apportare modifiche e miglioramenti.

- **Robustezza**: mediante il TOS 2.x Working Group, responsabile delle *abstractions* e *interfaces* del *core* del sistema operativo, si è cercato

di migliorare la robustezza della applicazioni, permettendo un facile sviluppo e una diminuzione degli errori, mediante appunto, una semplice e chiara definizione delle *interfaces* ed *abstractions*.

B.2 Hardware Abstraction Architecture

Hardware Abstraction Architecture (HAA [23]) cerca un compromesso tra i requisiti delle applicazioni *Wireless Sensor Networs* (WSN), in continua crescita, e il desiderio di portabilità e ottimizzazione del codice. Spesso la portabilità del codice non permette di sfruttare al massimo le capacità della singola piattaforma, ma come si vedrà in seguito, sarà comunque permesso di decidere in base alle proprie esigenze se sviluppare un'applicazione completamente portabile, oppure un'applicazione che sfrutta al massimo le potenzialità dell'hardware in modo semplice, grazie alle *interfaces* e *abstractions* che saranno rispettate ai diversi livelli dell HAA. HAA si suddivide in tre livelli, come visibile in figura B.1, rispettivamente in ordine crescente:

1. *Hardware Presentation Layer* (HPL)

2. *Hardware Adaption Layer* (HAL)

3. *Hardware Interface Layer* (HIL)

B.2.1 Hardware Presentation Layer

I componenti che appartengono a questo livello sono posizionati subito sopra l'interfaccia HW/SW. Il loro compito è presentare le caratteristiche

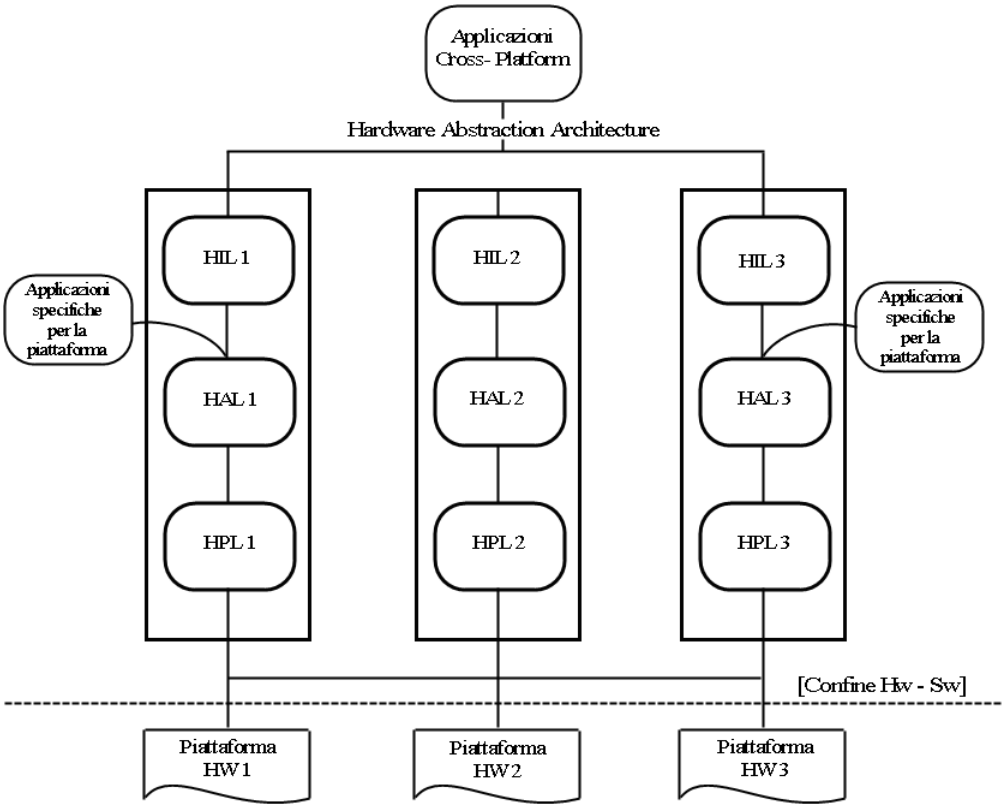


Figura B.1: Hardware Abstraction Architecture

dell'hardware utilizzando i concetti classici di comunicazione del sistema operativo. L'accesso all'hardware avviene tramite memoria oppure mediante *port mapped I/O*, mentre l'hardware può richiedere assistenza mediante la segnalazione di un interrupt. Le routine di gestione degli interrupt, dovranno svolgere solo le operazioni time critical. In questo modo, l'HPL, nasconde come il componente interagisce con l'hardware presentando semplici funzioni, che compongono l'interfaccia verso il sistema al livello superiore. Questa astrazione, che i componenti appartenenti ad una determinata classe dovranno rispettare, permette al programmatore di effettuare un cambio tra differenti moduli ricollegando (*rewiring*), e non riscrivendo, i componenti HPL, senza dover modificare l'implementazione del proprio codice. In particolare i componenti HPL dovranno essere *stateless* ed esibire un'interfaccia

che rappresenta un'astrazione completa delle caratteristiche dell'hardware. Ogni componente sarà unico, anche se dovrà pur sempre mantenere una struttura generale; infatti ogni componente HPL dovrà avere:

- comandi per inizializzare, eseguire, fermare il modulo, che sono necessari per *power management policy*
- comandi `get` e `set` per gestire i registri hardware
- comandi per abilitare e disabilitare gli interrupt generati dal modulo
- routine di gestione degli interrupt che sono generati dall'hardware
- comandi con nomi che descrivono le principali operazioni di flag-setting e testing

B.2.2 Hardware Adaption Layer

Prima caratteristica fondamentale dell'HAL è lo stato, a differenza dell'HPL, usato per gestire (arbitration) e controllare risorse. Questo livello è il cuore pulsante dell'architettura perché, anche se sembrerà un controsenso con il concetto di portabilità, individua tutte le caratteristiche della classe hardware, inclusi specifici comportamenti che rendono l'hardware differente da un altro, fornendo un'astrazione che rende efficiente lo sviluppo dell'applicazione, pur mantenendo un efficace uso delle risorse. HAL dovrà fornire un'accesso a questa astrazione mediante un'interfaccia, che include ogni singola caratteristica specifica dell'hardware, senza nascondere alcuna funzionalità. Come si può intuire dal diagramma dell'architettura, questo livello permette di sviluppare particolari applicazioni che necessitano di efficienza, ovviamente rinunciando alla portabilità.

B.2.3 Hardware Interface Layer

Il livello HIL permette la completa astrazione per lo sviluppo di applicazioni multi piattaforma (*cross-platform*). Ciò avviene nascondendo le differenze tra i vari hardware e mantenendo un'interfaccia comune. I componenti HIL dovranno pertanto rispettare il contratto dell'API che rappresenta il comportamento tipico richiesto nelle applicazioni WSN. Quando le caratteristiche dell'hardware sono maggiori rispetto a quelle dell'API, l'HIL effettua un downgrade dell'astrazione fornita dall'HAL finchè l'API non sarà aggiornata. Viceversa, quando l'hardware non può coprire completamente l'API, l'HIL potrà appoggiarsi ad un software di simulazione. Descritto questo livello, si può intuire che, se si vuole sviluppare applicazioni particolarmente performanti, ci si potrà appoggiare all'HAL, mentre se non è richiesta una particolare efficienza, sarà conveniente basarsi sull'HIL, sviluppando così un'applicazione multi piattaforma.

B.3 Boot e Inizializzazione del sistema operativo

La sequenza di boot [12, 1] di TOS 2.x è divisa in tre passi rappresentati in figura B.2:

1. Inizializzazione dello scheduler

2. Inizializzazione dei componenti

3. Notifica, mediante un segnale, che il processo di boot è stato completato

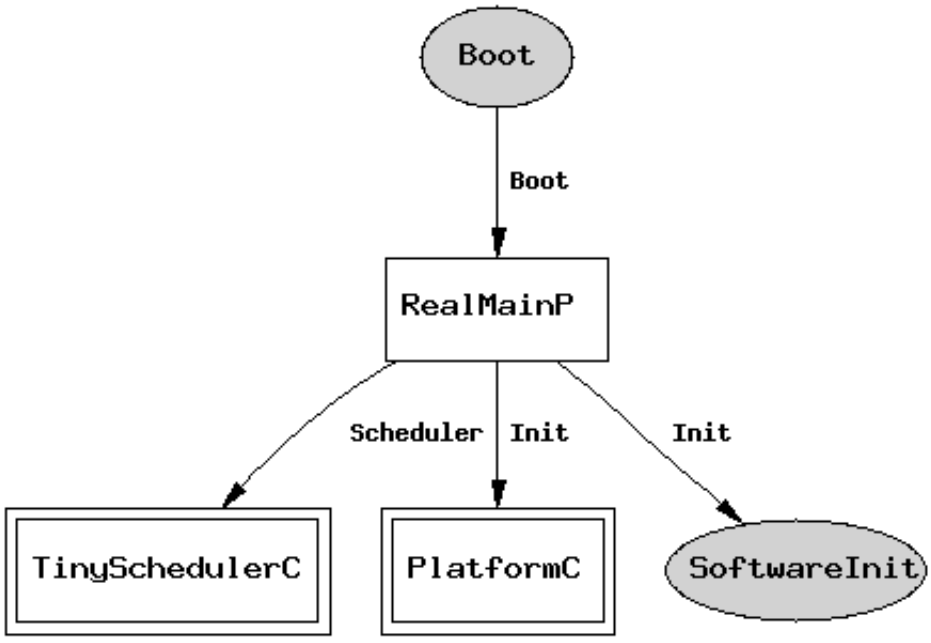


Figura B.2: Grafico dei componenti

B.3.1 MainC passo dopo passo

Per meglio descrivere la procedura di avvio del sistema operativo, riteniamo necessario procedere con una breve analisi del codice, descrivendo, passo dopo passo, gli stadi della fase di boot di TOS 2.x. Il file di configurazione, che ogni applicazione deve collegare, è `MainC`, che non è realmente il vero main. Il vero main è `RealMainP` collegato, come *pass-through wiring* in ciascuna applicazione TOS 2.x attraverso un collegamente presente in `MainC`.
`tos/system/MainC.nc`:

```
#include "hardware.h"

configuration MainC {
  provides interface Boot;
```

```
  uses interface Init as SoftwareInit;
}
implementation {
  components PlatformC, RealMainP, TinySchedulerC;

  RealMainP.Scheduler -> TinySchedulerC;
  RealMainP.PlatformInit -> PlatformC;

  // Export the SoftwareInit and Booted for applications
  SoftwareInit = RealMainP.SoftwareInit;
  Boot = RealMainP;
}
```

La vera implementazione del main è la seguente:

/tos/system/RealMainP.nc

```
module RealMainP {
  provides interface Boot;
  uses interface Scheduler;
  uses interface Init as PlatformInit;
  uses interface Init as SoftwareInit;
}
implementation {
  int main() __attribute__ ((C, spontaneous)) {
    atomic
      {

call Scheduler.init();
```

```
call PlatformInit.init();
while (call Scheduler.runNextTask());

call SoftwareInit.init();
while (call Scheduler.runNextTask());
      }

    /* Enable interrupts now that system is ready. */
    __nesc_enable_interrupt();

    signal Boot.booted();

    /* Spin in the Scheduler */
    call Scheduler.taskLoop();

 ...

  }

  default command error_t PlatformInit.init() {return SUCCESS;}
  default command error_t SoftwareInit.init() {return SUCCESS;}
  default event void Boot.booted() { }

}
```

Come si evince dal codice, possiamo individuare i tre passi precedentemente evidenziati, ma che in questo caso, andremo ad analizzare più precisamente.

B.3.2 Inizializzazione dello Scheduler

Lo scheduler è il primo componente inizializzato prima di qualsiasi altro. Se ciò non fosse vero, un componente che volesse fare il post dei task, non potrebbe, in quanto lo scheduler non sarebbe stato inizializzato. Si può implementare [1, 16], a seconda delle proprie esigenze un nuovo scheduler, scrivendo nella propria directory un file di configurazione chiamato `Scheduler` in modo da sostituire quello di default. Modificando tale file di configurazione, in modo opportuno, sarà quindi possibile collegare la propria implementazione dello scheduler. L'interfaccia per uno scheduler TOS 2.x si trova seguendo questo percorso:
`tos/interfaces/Scheduler.nc`

```
interface Scheduler {

  command void init();

  command bool runNextTask();

  command void taskLoop();
}
```

TOS è un sistema operativo *event-driven* che utilizza uno scheduler [22] strutturato a due livelli, in modo che, piccole quantità di lavoro associate con l'hardware, vengano svolte subito, interrompendo, se in esecuzione, i task. I due livelli di scheduler sono i seguenti:

- **Event**: gli eventi, ovvero *interrupt handler*, svolgono piccole quantità di lavoro e hanno il diritto di prelazione sui task
- **Task**: i task possono svolgere grandi quantità di lavoro e non sono *time critical* a differenza degli eventi

Con questa soluzione vi è un alto livello di concorrenza e non è necessario spazio aggiuntivo per il *context switch*. I task, come si può intendere dalla spiegazione precedente, hanno la proprietà di *run to completion*, che implica l'utilizzo di un solo stack. Ovviamente i task non hanno il diritto di prelazione su altri task con la conseguenza di essere eseguiti atomicamente rispetto agli altri tasks, ma non atomicamente rispetto ai gestori degli eventi (*interrupt handlers*). Lo scheduler di default, di TOS 2.x, è responsabile per la policy dei differenti tipi di task che potrebbero essere implementati. I task di base in TOS 2.x sono senza parametri e FIFO. Quando viene richiesto il `post` di un task, vi è un fallimento solo se è già stato richiesto il `post` del medesimo task e non è ancora in esecuzione. Di conseguenza un task può sempre essere eseguito, ma non è possibile eseguire il `post` più di una volta.

B.3.3 Inizializzazione dei componenti

Dopo l'inizializzazione dello scheduler, vi è l'inizializzazione dei componenti. La sequenza di boot suddivide l'inizializzazione dei componenti in:

- *platform initialization phase*
- *software initialization phase*

Affinchè tali fasi siano complete, è necessario fornire l'implementazione per l'uso dell'interfaccia `Init` presente al seguente percorso:
`tos/interfaces/Init.nc`

```
#include "TinyError.h"

interface Init {

  command error_t init();
```

```
}
```

La differenza tra le due fasi è che la *platform initialization phase* precede la *software initialization phase*; ciò è necessario perché l'hardware deve essere correttamente inizializzato attraverso `PlatformInit`. `SoftwareInit`, chiama, quando eseguita, i componenti che vogliono essere inizializzati prima che l'esecuzione dell'applicazione inizi. Nel caso non vi sia nessuna implementazione collegata, `RealMainP.nc` ha i seguenti *default handler*:

```
...

default command error_t PlatformInit.init() {return SUCCESS;}
default command error_t SoftwareInit.init() {return SUCCESS;}
default event void Boot.booted() { }

...
```

Un *default handler* è una implementazione di una funzione che viene usata se non esiste nessuna implementazione collegata. Ovviamente se esiste un'implementazione collegata di tale funzione, sarà chiamata al posto del *default handler*. Per segnalare il completamento dell'inizializzazione del sistema viene segnalato un evento `signal Boot.booted()`, eseguito in `tos/system/RealMainP.nc`, il quale sarà opportunamente gestito dall'applicazione sviluppata.

B.3.4 Packet Protocols

L'astrazione di base della rete in TOS, è l'*Active Message* (AM) [14], single-hop a trasmissione di pacchetto inaffidabile. Gli AM hanno un'indirizzo di destinazione, forniscono ACK sincroni e possono avere una lunghezza

variabile fino al raggiungimento della massima dimensione fissata. Il *packet-level communication* ha tre classi di interfacce di base:

- *Packet*: le interfacce sono necessarie per accedere ai campi del messaggio e dati (payload)
- *Receive*: le interfacce sono necessarie per la gestione degli eventi di ricezione del pacchetto
- *Send*: le interfacce sono necessarie per la trasmissione del pacchetto e si distinguono per il loro prototipo di funzione che utilizza uno schema di indirizzo diverso

In TOS 2.x il message buffer è un tipo message_t [13], reperibile al seguente path

`tos/types/message.h`

```
#ifndef __MESSAGE_H__
#define __MESSAGE_H__

#include "platform_message.h"

#ifndef TOSH_DATA_LENGTH
#define TOSH_DATA_LENGTH 28
#endif

#ifndef TOS_BCAST_ADDR
#define TOS_BCAST_ADDR 0xFFFF
#endif

typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
```

```
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;

#endif
```

Prima di procedere alla descrizione dell'inivio e ricezione occorre spiegare come viene rappresentato il messaggio in TOS 2.x. Come si deduce dal codice, questo modo di gestire il messaggio permette di tenere il campo `data` ad un offset in una posizione fissa in modo da facilitare il passaggio del message buffer tra due differenti *link layer*. Bisogna porre una particolare attenzione alla posizione di partenza del pacchetto in quanto, essendo il campo `data` ad un offset fissato, potrebbe accadere che il campo `header` non incominci all'inizio del pacchetto, perciò è necessario individuare il campo `header` come un offset negativo rispetto al campo `data`. Ogni link layer definisce `header`, `footer`, e `metadata`. Inoltre il link layer fornisce l'interfaccia per accedere ai campi, quando necessario, in quanto non è possibile accedere ai campi in modo diretto. Il campo `metadata` memorizza informazioni che un single-hop stack usa o raccoglie, ma che non trasmette. Quando viene sviluppato un livello sopra ad un altro, accade che, il payload non è più ad un offset fissato. Per questo motivo viene in aiuto l'interfaccia `Packet` che definisce dei modi per scoprire dove si possono inserire i dati (`data`). tos/interfaces/Packet.nc:

```
#include <message.h>

interface Packet {

  command void clear(message_t* msg);
```

```
  command uint8_t payloadLength(message_t* msg);
  command void setPayloadLength(message_t* msg, uint8_t len);
  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg, uint8_t* len);

}
```

- `clear`: ripulisce il messaggio
- `getPayload`: viene ritornato un puntatore alla *data region* del pacchetto
- `payloadLength`: viene ritornata la dimensione della *data region* del pacchetto
- `setPayloadLength`: imposta la dimensione del payload. Non viene usato quando si vuole trasmettere (durante la trasmissione viene già impostata la lunghezza), ma quando si vuole memorizzare la lunghezza del payload e si vuole utilizzarla, ad esempio, per la gestione delle coda della trasmissione
- `maxPayloadLength`: ritorna la lunghezza massima del payload che il livello di comunicazione che la implementa può fornire

Per la ricezione dei pacchetti si deve implementare la seguente interfaccia:

`tos/interfaces/Receive.nc`

```
#include <TinyError.h>
#include <message.h>

interface Receive {
```

```
  event message_t* receive
(message_t* msg, void* payload, uint8_t len);

  command void* getPayload(message_t* msg, uint8_t* len);
  command uint8_t payloadLength(message_t* msg);

}
```

`getPayload` e `payloadLength` hanno la stessa semantica e motivazione simile di quella descritta per l'interfaccia `Packet`. Per quanto riguarda la gestione dell'evento `receive`, la questione si fa delicata. L'interfaccia `Receive` segue una *buffer-swap* policy. Il gestore dell'evento deve ritornare un puntatore ad un messaggio valido così che lo stack lo possa usare per memorizzare il prossimo pacchetto ricevuto. Questo modo di procedere permette un equilibrio tra il livello superiore ed inferiore, in quanto se il livello superiore non è in grado di gestire i pacchetti con la medesima frequenza con cui arrivano, il livello superiore può comunque ritornare il puntatore ad un messaggio valido al livello inferiore. Questo messaggio può essere lo stesso passato come parametro `message_t* msg`, che viene quindi ritornato senza neanche essere guardato. In questo modo avviene lo scarto (*drop*) del pacchetto, perché non è stato possibile gestirlo, ma viene comunque concesso ad altri componenti di continuare a ricevere pacchetti. In breve, per gestire l'evento si può:

1. ritornare `message_t* msg` senza neanche guardarlo
2. si ricopia il payload e si ritorna `message_t* msg`
3. si memorizza il `message_t* msg` nel *local frame* e si ritorna un nuovo `message_t*` per il livello inferiore

Per la trasmissione del pacchetto vi sono diverse interfacce, corrispondenti a differenti modi di indirizzamento. Ad esempio se vogliamo utilizzare la comunicazione con *Active Message*, l'interfaccia per la trasmissione da implementare sarà:

tos/interfaces/AMSend.nc

```
#include <TinyError.h>
#include <message.h>
#include <AM.h>

interface AMSend {

  command error_t send
 (am_addr_t addr, message_t* msg, uint8_t len);
  command error_t cancel(message_t* msg);

  event void sendDone(message_t* msg, error_t error);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg);

}
```

mentre per un'address-free, vi è l'interfaccia tos/interfaces/Send.nc:

```
#include <TinyError.h>
#include <message.h>

interface Send {
```

```
  command error_t send(message_t* msg, uint8_t len);
  command error_t cancel(message_t* msg);

  event void sendDone(message_t* msg, error_t error);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg);

}
```

Appendice C

nesC

C.1 Caratteristiche principali

Il linguaggio di programmazione utilizzato per lo sviluppo di applicazioni per TOS e per lo sviluppo di TOS stesso è *nesC*, un'estensione del linguaggio C [10]. La grammatica e la semantica sono presentate in [7]. È un linguaggio di programmazione per *networked embedded system* come ad esempio lo sviluppo di applicazioni per reti di sensori. Tale linguaggio deve avvicinarsi il più possibile alle caratteristiche dei sistemi embedded, favorendo un semplice sviluppo, ottimizzazione e prevenzione da possibili errori. Infatti il compilatore nesC compie una completa analisi del programma, compresa di rilevazione di *race-condition* che migliora l'affidabilità, riduzione della dimensione del codice, e l'*inlining* delle funzioni che riduce l'uso di risorse. In generale ogni mote può eseguire una singola applicazione alla volta che viene sviluppata appoggiandosi su un framework di componenti di sistema fornito da TOS. Tutte le risorse disponibili sono conosciute staticamente; non è possibile, ad esempio, l'allocazione di memoria dinamica, permettendo in questo modo la rilevazione al momento della compilazione di race-condition. Questo linguaggio permette un rapido e semplice sviluppo di applicazioni

Figura C.1: Dispositivo Tmote Sky - IEEE 802.15.4 compliant device for wireless mesh networking featuring a 250kbps radio, 10kB RAM, 48kB flash, and 1MB storage

riutilizzabili. I mote sono dispositivi che sono comandati principalmente da eventi (*event-driven*), quali ad esempio il segnalamento di una particolare temperatura rilevata da un sensore, definendo così un sistema concorrente piuttosto che un sistema guidato dall'interazione o *batch processing*. Proprio perché si tratta di attività concorrenti bisogna portare particolare attenzione alle race-condition e a potenziali bug. Questi dispositivi inoltre, non hanno particolari risorse e non sono nemmeno molto performanti, in quanto il loro obiettivo è essere molto piccoli, consumare e costare poco. In figura C.1 è possibile vedere uno dei tanti mote prodotti. Non sono richieste particolari operazioni *time critical*, tranne alcune eccezioni, procedendo così allo sviluppo di applicazioni *soft real-time*. nesC cerca di dare una certa sicurezza sulla rilevazione di errori in quanto, a questi mote, potrebbe essere richiesto di lavorare per molto mesi, collezionando dati acquisiti dai sensori senza l'intervento da parte dell'uomo. L'unica soluzione ad un crash dell'applicazione è il riavvio automatico del sistema.

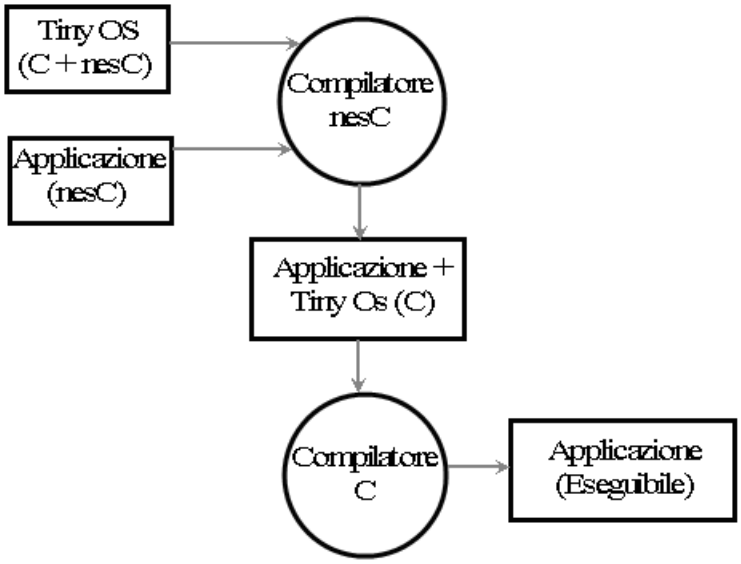


Figura C.2: Compilazione sorgente nesC

C.1.1 Compilazione

La compilazione di un'applicazione sviluppata in nesC per TOS, visibile in figura C.2, avviene nei seguenti passi:

1. il file sorgente viene passato al compilatore nesC
2. nesC ricerca le librerie necessarie per la compilazione dell'applicazione
3. nesC ingloba quindi applicazione, kernel e librerie
4. viene generato un file sorgente in C
5. il compilatore C compila il file sorgente C generato

C.2 nesC in breve

nesC [20] è un dialetto del linguaggio C basato sui componenti [15]. I componenti usano un *local namespace*. Ciò significa che oltre a dichiarare le funzioni che implementano, devono anche dichiarare le funzioni che chiamano.

Ogni componente ha una descrizione (*signature*), ovvero un pezzo di codice dove vengono dichiarate le funzioni che il componente fornisce(*provides*) e le funzioni che chiama(*uses*). nesC possiede anche le interfacce(*interfaces*), che sono un insieme di dichiarazioni di funzioni che i componenti che forniscono tale interfaccia dovranno implementare. La connessione tra *providers* e *users* è chiamata *wiring*.

C.2.1 Interfacce

nesC possiede due tipi di componenti: configurazioni (*configuration*) e moduli (*modules*). Le configurazioni sono necessarie per stabilire come i componenti sono collegati insieme (*wiring*). I moduli invece sono le implementazioni delle funzioni dichiarate nei componenti. Come spiegato in precedenza, TOS è un sistema operativo (*event-driven*), con un alto grado di concorrenza. Proprio per questo motivo è importante definire il concetto di *split-phase*. Split-phase significa che non ci sono operazioni bloccanti, ovvero operazioni di richiesta e completamento sono separate. Le interfacce split-phase sono bidirezionali: c'è una *downcall* che dà inizio all'operazione e una *upcall* che ne segnala il completamento. Possiamo chiamare le downcall, *command*, e le upcall, *event*. Riprendiamo come esempio l'interfaccia `Send` già vista in precedenza al seguente percorso:
`tos/interfaces/Send.nc` già vista in precedenza:

```
#include <TinyError.h>
#include <message.h>

interface Send {

  command error_t send(message_t* msg, uint8_t len);
  command error_t cancel(message_t* msg);
```

```
  event void sendDone(message_t* msg, error_t error);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg);

}
```

In questo caso vi sarà un componente, provider oppure user, che deciderà quale parte dell'operazione split-phase rappresentare. Un provider di `Send` definisce i command, `send`, `cancel`, `maxPayloadLength`, `getPayload` e può segnalare (*signal*) l'evento `sendDone`. Viceversa, un user di `Send` definisce l'evento `sendDone`, e potrà chiamare (*call*) i comandi `send` e `cancel`. Sostanzialmente quando una chiamata a `send` ritorna `SUCCESS`, il parametro `msg` è stato passato al provider, il quale proverà a trasmettere il pacchetto. Quando il pacchetto sarà stato trasmesso, il provider segnalerà l'evento `sendDone` al componente user che avrà gestito la sua implementazione. Le interfacce possono prendere dei tipi (*types*) come argomenti, che vengono inseriti nella definizione dell'interfaccia nel seguente modo: `interface name <type>`. Possono essere utilizzate per definire il tipo di una variabile all'interno di una funzione oppure per obbligare il controllo dei tipi. Questo controllo è possibile perché non è consentito collegare (wiring) providers e users, che usano interfacce con argomenti, dove i tipi non corrispondono. Ad esempio l'interfaccia:
`tos/lib/timer/Timer.nc`

```
interface Timer<precision_tag>
{
  ...
}
```

Questa interfaccia non ha nessuna funzione che utilizza `precision_tag`. In questo caso, infatti, viene usato per il controllo del tipo quando si effettua il collegamento(*wiring*). `precision_tag` indica la precisione del timer. In questo modo si previene il possibile errore di collegare componenti che si aspettano precisioni diverse.

C.2.2 Moduli

I moduli sono quei componenti che possiedono una implementazione. Un modulo deve implementare ogni command delle interfacce che fornisce e ogni event delle interfacce che usa. `apps/Blink/BlinkC.nc`

```
module BlinkC
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
```

```
  {
    dbg("BlinkC", "Timer 0 fired @ %s.\n", sim_time_string());
    call Leds.led0Toggle();
  }

  ...
}
```

Come si vede dall'esempio per ogni interfaccia che viene utilizzata, vengono implemetati tutti i suoi *events*.

C.2.3 Tasks

I tasks vengono utilizzati da quei componenti che non hanno bisogno di eseguire operazioni in modo immediato. Infatti i task, una volta eseguita l'operazione di `post`, che ritorna immediatamente, vengono schedulati, e quindi rimandata la loro esecuzione a quando lo scheduler li eseguirà. L'operazione di `post` ritorna `error_t`. Se il task non è presente nella coda, allora `post` ritorna `SUCCESS`, altrimenti significa che il task è ancora nella coda, cioè, è stata eseguita l'operazione di `post`, ma non è ancora stato eseguito il task, e viene ritornato `FAILED`. Il task viene eseguito fino al suo completamento (run to completion) e non può essere prelazionato da nessun'altro task. Solamente gli event, interrupt handler, da non confondere con funzioni `sync`, hanno il diritto di prelazione sui task. Tutto il codice chiamato da un interrupt handler deve essere `async` (asincrono). Quando un interrupt handlers vuole eseguire del codice `sync`, l'unica soluzione è eseguire il post (operazione `async`) di un task nel quale può essere eseguito del codice `sync`, ovvero codice non bloccante, che ovviamente non può essere prelazionato. Ribadiamo il fatto che ogni funzione `async` può prelazionare un'altra funzione `async` oppure un task, ma non una funzione (`sync`) non

bloccante. Siccome con l'esecuzione di codice asincrono sono possibili le race-condition, si può inserire il codice che può essere soggetto ad uno stato inconsistente, in un blocco `atomic`, che rende l'esecuzione del codice al suo interno atomico, ovvero non può avvenire prelazione e quindi nessuna race-condition. Una semplicissima soluzione per l'esecuzione del codice in modo atomico, sarebbe disabilitare gli interrupt, comportando però un grande spreco di CPU (vi sono soluzioni più efficienti, ad esempio i MUTEX). Un semplice esempio che mostra una possibile race-condition è il seguente:

```
...
bool state;

async command bool setState()
{
  if (state == TRUE)
  {
    state = FALSE;
    return FALSE;
  }

  if (state == FALSE)
  {
    state = TRUE;
    return TRUE;
  }
}
```

Ora pensiamo a questa possibile esecuzione:

```
setState()
```

```
state = FALSE;
  --> interrupt
      setState()
      state = TRUE;
      return TRUE;
return FALSE;
```

Come si evince dal codice è avvenuta una race-condition, come di seguito descritto:

1. viene chiamato `setState()`
2. viene impostata la variabile `state = FALSE`, prima però dell'istruzione `return FALSE`
3. arriva la segnalazione dell'interrupt
4. si chiama nuovamente `setState()`, che in questo caso trova la variabile a `FALSE`
5. di conseguenza imposta `state = TRUE`
6. ritorna poi `TRUE`
7. si riprende dal punto in cui la prima `setState()` è stata interrotta, e in questo caso viene ritornato `FALSE`

Il codice non è stato eseguito come ci si aspettava, in quanto il valore finale di ritorno sarebbe dovuto essere `TRUE`. Una soluzione come menzionato in precedenza, sarebbe includere il codice, potenzialmente soggetto a inconsistenze, in un blocco atomico, che ovviamente dovrà essere il più breve possibile per non portare ad uno spreco della CPU. Il seguente codice risolve il problema:

```
...
bool state;

async command bool setState()
{
  if (state == TRUE)
  {
    atomic
    {
      state = FALSE;
      return FALSE;
    }
  }

  if (state == FALSE)
  {
    atomic
    {
      state = TRUE;
      return TRUE;
    }
  }
}
```

Per ridurre la latenza dovuta al run to completion è necessario sviluppare task brevi. Inoltre segnalare un evento con un comando, può portare ad un loop delle chiamate, corruzione della memoria e addirittura al crash dell'applicazione. Ad esempio un loop delle chiamate può avvenire se un componente `A` esegue un comando `X` che segnala un evento gestito nel

componente `B` che a sua volta chiama il comando `X` implementato nel componente `A`. Il seguente codice mostra come deve essere definito e messo in coda un task tramite l'operazione di `post`:

```
task void taskname() {
  ...
}

event void eventname() {
  ...
  post taskname();
}
```

C.2.4 Configurazioni e Wiring

Come già descritto in precedenza, ci sono due tipi di componenti: *module* e *configuration*. I moduli forniscono l'implementazione di una o più interfacce. Le configurazioni sono usate per unire i componenti, ovvero collegano le interfacce usate dai componenti, oppure le interfacce che vengono fornite ad altri componenti. Oltre al collegamento di componenti, le configurazioni permettono anche di esportare le interfacce, mediante un metodo chiamato *pass-through wiring*.

Semplici esempi di configurazione sono: `tos/system/LedsC.nc`

```
configuration LedsC {
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;
```

```
  Leds = LedsP;

  LedsP.Init <- PlatformLedsC.Init;
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```

apps/Blink/BlinkAppC.nc:

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;


  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

Vi sono tre operatori:

- `->` collega un user ad un provider, ad esempio// `BlinkC.Leds -> LedsC`. Infatti se andiamo a rileggere il codice precedentemente presentato, troviamo il modulo `BlinkC` che usa (user) l'interfaccia `Leds`, mentre `LedsC` è un file di configurazione che fornisce `Leds` (provider). Questo dimostra che si tratta sempre di un collegamento da user a provider. É da notare inoltre che non viene scritto esplicitamente `BlinkC.Leds -> LedsC.Leds`, in quanto viene mappato lo stesso nome, altrimenti non sarebbe stato ovviamente possibile.

- `<-` collega un provider ad un user, ad esempio `LedsP.Init <- PlatformLedsC.Init`

- `=` viene delegata l'implementazione di un'interfaccia ad un componente, mentre gli altri operatori permettono di combinare altri componenti per completare le *signature* esistenti. Un esempio è `Leds = LedsP`, dove `Leds` è un'interfaccia e `LedsP` l'implementazione.

La keyword `as` viene utilizzata quando all'interno della signature del componente viene nominata due volte la stessa interfaccia. Infatti il compito della keyword `as` è solamente rinominare un interfaccia nella signature del componente. All'interno dell'implementazione sarà poi possibile utilizzare tale interfaccia ridenominata, semplicemente con il nuovo nome. É possibile vedere il suo uso mediante il precedente esempio:
`apps/Blink/BlinkAppC.nc`

```
configuration BlinkAppC
{
}
implementation
{
  ...
  components new TimerMilliC() as Timer0;
```

```
  components new TimerMilliC() as Timer1;

  ...

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  ...
}
```

Si effettua *Pass Through Wiring* quando una configurazione collega due interfacce nella sua signature. Ad esempio:

```
configuration changeNameC
{
  provides interface AMPacket as NewPacket;
  uses interfaces AMPacket as SubPacket;
}
implementation
{
  NewPacket = SubPacket;
}
```

Un componente che effettua un collegamento a `changeNameC.NewPacket` viene collegato a qualunque cosa è stato connesso `changeNameC.SubPacket`.

Vi è inoltre la possibilità di effettuare *Multiple Wiring*, *Fan-Out* e *Fan-In*. Per esempio osserviamo la seguente configurazione:
`tos/chips/cc2420/CC2420TransmitC.nc`

```
configuration CC2420TransmitC {
```

```
  provides interface Init;
  provides interface AsyncStdControl;
  provides interface CC2420Transmit;
  provides interface CsmaBackoff;
  provides interface RadioTimeStamping;

}

implementation {

  components CC2420TransmitP;
  components AlarmMultiplexC as Alarm;
  Init = Alarm;
  Init = CC2420TransmitP;
  ...
}
```

Questo multiple wiring, fan-out per la precisione, significa che quando un componente chiama `CC2420TransmitC.Init.init()`, chiamerà sia `Alarm.Init.init()` che `CC2420TransmitP.Init.init()`. Quindi mediante un solo punto di chiamata vengono automaticamente eseguite altre due chiamate (1 a molti). É possibile anche il contrario, fan-in (molti a 1), ovvero diversi componenti sono collegati con lo stesso. Se i componenti che effettuano il collegamento allo stesso componente sono single-phase, ovvero hanno solo comandi, si avrà un puro fun-in. Nel caso in cui i componenti che effettuano il collegamento allo stesso componente sono split-phase, allora quando il componente collegato agli altri segnalerà un evento, tale evento sarà segnalato a tutti i componenti a cui il componente è collegato, comportando così un *Fun-In Fun-Out* allo stesso tempo. Bisogna porta-

re particolare attenzione ai valori di ritorno quando vi sono collegamenti fun-out. Infatti il valore di ritorno sarà il risultato dell'applicazione della *combine function* a tutti i valori di ritorno delle chiamate a funzione. Se tutte le funzioni ritornano lo stesso valore, allora sarà proprio tale valore ad essere ritornato, altrimenti viene ritornato `FALSE`.

C.2.5 Wiring Parametrizzato

Quando un componente vuole fornire diverse istanze di un'interfaccia si possono usare le interfacce parametrizzate, tramite un *parametrized wiring*, che non sono altro che un array di interfacce dove l'indice dell'array è il parametro. Un largo uso delle interfacce parametrizzate, in TOS 2.x, viene effettuato negli AM. Prendiamo come esempio l'applicazione di test: `apps/tests/TestAM/TestAMAppC.nc`

```
configuration TestAMAppC {}
implementation {
  components MainC, TestAMC as App, LedsC;
  components ActiveMessageC;
  components new TimerMilliC();

  App.Boot -> MainC.Boot;

  App.Receive -> ActiveMessageC.Receive[240];
  App.AMSend -> ActiveMessageC.AMSend[240];
  App.SplitControl -> ActiveMessageC;
  App.Leds -> LedsC;
  App.MilliTimer -> TimerMilliC;
}
```

Quando `TestAM`, rinominato `App`, chiama `AMSend.send`, che in realtà chiama `ActiveMessageC.AMSend[240]`, ovvero specifica il *protocol ID* del pacchetto che viene trasmesso. Una volta che viene trasmesso il pacchetto, l'evento `sendDone`, sarà segnalato solamente al componente che ha specificato quel protocol ID. Osservando le varie implementazione dei componenti, tale comportamento sarà più chiaro. `ActiveMessageC` è una configurazione che racchiude un particolare chip, ad esempio `CC2420ActiveMessageC`, il quale a sua volta trova la sua implementazione nel modulo `CC2420ActiveMessageP`.

`tos/chips/cc2420/CC2420ActiveMessageC`

```
#include "CC2420.h"

configuration CC2420ActiveMessageC {
  provides {
    ...
    interface AMSend[am_id_t id];
    interface Receive[am_id_t id];
    interface Receive as Snoop[am_id_t id];
    ...
  }
}
implementation {

  components CC2420ActiveMessageP as AM;
  ...

  AMSend   = AM;
  Receive  = AM.Receive;
  Snoop    = AM.Snoop;
```

```
  ...
}
```

tos/chips/cc2420/CC2420ActiveMessageP

```
module CC2420ActiveMessageP {
  provides {
    interface AMSend[am_id_t id];
    interface Receive[am_id_t id];
    interface Receive as Snoop[am_id_t id];
    interface AMPacket;
    interface Packet;
  }
  uses {
    interface Send as SubSend;
    interface Receive as SubReceive;
    command am_addr_t amAddress();
  }
}
implementation {
   ...

   command error_t AMSend.send[am_id_t id](am_addr_t addr,
  message_t* msg,
  uint8_t len) {
    cc2420_header_t* header = getHeader( msg );
    header->type = id;
    header->dest = addr;
    header->destpan = TOS_AM_GROUP;
```

```
    return call SubSend.send( msg, len + CC2420_SIZE );
  }

  ....

  event void SubSend.sendDone(message_t* msg, error_t result) {
    signal AMSend.sendDone[call AMPacket.type(msg)](msg, result);
  }

  ...

}
```

Attraverso l'uso delle interfacce parametrizzate è possibile eliminare il codice ridondante e prevenire possibili bug. Quando non si è interessati al valore del parametro è possibile utilizzare la funzione `unique`. É una compile-time function, perché risolta a compile-time. Prende una stringa come argomento, e promette che il valore intero ritornato, sia unico per tutte quelle volte che viene chiamata con quella stringa. É possibile che con stringhe diverse sia ritornato lo stesso valore. Se si effettuano n chiamate a `unique`, allora `unique` ritornerà dei valori compresi tra 0 e $(n-1)$. Se si vuole sapere il numero di chiamate a `unique` con una particolare stringa, si può utilizzare la funzione `uniqueCount` passando la stringa come parametro. Se ci sono state n chiamate a `unique` allora il valore di ritorno di `uniqueCount` sarà n. Tale funzione, come la precedente, è una compile-time function.

C.2.6 Componenti Generici

Normalmente, i componenti sono *singleton*, cioè il nome di un componente è una singola entità in un *global namespace*. Ad esempio due configurazione

che fanno riferimento a `MainC`, fanno riferimento allo stesso blocco di codice. I componenti generici (*generic components*) non sono singleton. Possono essere istanziati all'interno di una configurazione (`components new nameGenericComponent`). Attraverso l'uso dei componenti generici è possibile scrivere un solo componente è riutilizzarlo più volte, a differenza dei componenti singleton che non possono virtualizzare lo stesso componente in più componenti. Ad esempio `VirtualizeTimerC`, presente in `tos/lib/timer`, usa un singolo `Timer`, per creare fino 255 timer virtuali. I componenti generici possono avere argomenti e sono i seguenti:

- tipi: dichiarati con l'utilizzo della keyword `typedef`
- costanti numeriche
- costanti alfanumeriche

La signature di `VirtualizeTimerC` è la seguente:

```
generic module VirtualizeTimerC
(typedef precision_tag, int max_timers)
{
  provides interface Timer<precision_tag> as Timer[uint8_t num];
  uses interface Timer<precision_tag> as TimerFrom;
}
```

Per esempio se abbiamo queste due linee di codice:

```
components new VirtualizeTimerC(TMicro, 10) as TimerA;
components new VirtualizeTimerC(TMilli, 2) as TimerB;
```

nesC genererà due copie del codice di `VirtualizeTimerC` perché vengono effettuate due istanze di `VirtualizeTimerC` con due tipi di precisioni diverse. Oltre ai moduli generici vi sono anche le configurazioni generiche. Come

un modulo diventa una porzione di codice riusabile, anche le configurazioni diventano un'insieme di relazione riusabili.

Ad esempio: `tos/system/TimerMilliC.nc`

```
#include "Timer.h"

generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
implementation {
  components TimerMilliP;

  Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

Il suo utilizzo e vantaggio, è meglio esplicabile con un esempio pratico. Prendiamo la seguente applicazione:

`apps/Blink/BlinkAppC.nc`

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;


  BlinkC -> MainC.Boot;
```

```
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

Una volta che nesC compila il file, tutti i livelli intermedi tra `BlinkC` e `VirtualizedTimerC` vengono eliminati, lasciando solo un collegamento diretto tra i due componenti. Per concludere, possiamo osservare quanto l'uso dei generic component renda lo sviluppo dell'applicazione più semplice, assenza di codice ridondante, prevenzione da bug per la copia e modifica del codice, possibilità di cambiare un solo componente senza bisogno di andar a modificare tutti quelli che l'hanno implementato. Tutti questi vantaggi vanno a discapito della riduzione dello spazio occupato dal codice, in quanto ogni volta che viene fatto una `new` viene creata una nuova istanza e quindi una nuova copia del medesimo codice. Un'approfondimento maggiore su questo argomento, utile perché fortemente utilizzato in TOS 2.x, lo si può trovare in [15].

Ringraziamenti

Desidero ringraziare il mio Relatore Federico Pedersini e la mia correlatrice Anna Morpurgo per la fiducia, incoraggiamento e sostegno che mi hanno dato prima e durante lo stesura di questa tesi.

Ringrazio in particolare Marco, Francesca, Chiara, Désirée, Massimiliano e Alberto che hanno avuto sempre una buona parola e un consiglio nel momento giusto.

Un ringraziamento speciale a Valeria Ferrari che mi ha insegnato a non demordere davanti ai problemi, ed alla mia famiglia, per avermi fornito i mezzi e il sostegno morale in tutte le situazioni di difficoltà che mi si sono presentate.

Bibliografia

[1] *TinyOS Boot and System Initialization.* Tutorial. http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/lesson6.html, 2006. [citato a p. 77, 81]

[2] Chih-Chieh Han Andreas Savvides and Mani B. Strivastava. *Dynamic Fine-Grained Localization in Ad-Hoc Networks of Sensors.* Proceedings of the 7th annual international conference on Mobile computing and networking (MobiCom '01), 2001. [citato a p. 21, 26]

[3] Maja J Mataric Andrew Howard and Gaurav Sukhatme. *Relaxation on a Mesh: a Formalism for Generalized Localization.* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS01), October 2001. [citato a p. 23]

[4] Jan Beutel Chris Savarese, Jan M. Rabaey. *Locationing in distributed Ad-Hoc wireless sensor networks.* Berkeley Wireless Research Center, Computer Engineering and Networks Lab ETH Zurich. [citato a p. 22, 26]

[5] Jan M. Rabaey Chris Savarese and Koen Langendoen. *Robust positioning algorithms for distributed ad-hoc wireless sensor networks.* USENIX technical annual conference, (Monterey, CA), 2002. [citato a p. 22, 26]

[6] Mani Srivastava David Culler, Deborah Estrin. *Overview Of Sensor Networks.* IEEE Computer Society, August 2004. [citato a p. 1]

[7] David Culler Eric Brewer David Gay, Philip Levis. *nesC 1.1 Language Reference Manual*. May 2003. [citato a p. 91]

[8] Y. Sankarasubramaniam E. Cayirci I.F. Akyildiz, W. Su. *Wireless sensor networks: a survey*, volume 38. Computer Networks, December 2001. [citato a p. 7]

[9] C. Sharp J. Polastre, R. Szewczyk and D. Culler. *The Mote Revolution: Low Power Wireless Sensor Network Devices*. Proceedings of Hot Chips 16: A Symposium on High Performance Chips, August 2004. [citato a p. 6]

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (2nd Edition)*. Prentice Hall PTR; 2 edition, March 1988. [citato a p. 91]

[11] Laurent El Ghaoui Lance Doherty, Kristofer S. J. Pister. *Convex Position Estimation in Wireless Sensor Networks*. Infocom, April 2001. [citato a p. 20, 26]

[12] Philip Levis. *TinyOS 2.x Boot Sequence*. TinyOS Enhancement Proposals 107. http://www.tinyos.net/tinyos-2.x/doc/html/tep107.html, 2005. [citato a p. 77]

[13] Philip Levis. *message_t*. TinyOS Enhancement Proposals 111. http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html, 2006. [citato a p. 84]

[14] Philip Levis. *Packet Protocols*. TinyOS Enhancement Proposals 116. http://www.tinyos.net/tinyos-2.x/doc/html/tep116.html, 2006. [citato a p. 83]

[15] Philip Levis. *TinyOS Programming*. http://csl.stanford.edu/ pal/pubs/tinyos-programming.pdf, February 2006. [citato a p. 93, 112]

[16] Philip Levis and Cory Sharp. *Schedulers and Tasks*. TinyOS Enhancement Proposals 106. http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html, 2005. [citato a p. 81]

[17] Dragos Niculescu and Badri Nath. *Ad Hoc Positioning System(APS) Using AoA*. INFOCOM, San Francisco, CA, 2003. [citato a p. 22]

[18] Deborah Estrin Nirupama Bulusu, John Heidemann. *GPS-less Low Cost Outdoor Localization For Very Small Devices*, volume 7. IEEE Personal Comm. Magazine, April 2000. [citato a p. 21, 26]

[19] Anit Chakraborty Nissanka B. Priyantha and Hari Balakrishnan. *The Cricket Location-Support System*. Proc. of the Sixth Annual ACM International Conference on Mobile Computing and Networking (MOBICOM), August 2000. [citato a p. 21]

[20] Robert von Behren Matt Welsh Eric Brewer Philip Levis, David Gay and David Culler. *The nesC Language: A Holistic Approach to Networked Embedded Systems*. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03), 2003. [citato a p. 93]

[21] Nissanka B. Priyantha, Hari Balakrishnan, Erik Demaine, and Seth Teller. *Anchor-Free Distributed Localization in Sensor Networks*. MIT Laboratory for Computer Science, April 2003. [citato a p. 20, 21, 26, 27, 33, 70]

[22] Claudio Scordino. *Scheduling in TinyOS*. Computer Science Department - University of Pisa, Aprile 2005. [citato a p. 81]

[23] Jan-Hinrich Hauer Cory Sharp Adam Wolisz Vlado Handziski, Joseph Polastre and David Culler. *Hardware Abstraction Architecture*. TinyOS Enhancement Proposals 2. http://www.tinyos.net/tinyos-2.x/doc/html/tep2.html, 2004. [citato a p. 74]

[24] M. Liebowitz-B. Pister K.S.J. California Univ. Berkeley CA Warneke, B. Last. *Smart Dust: communicating with a cubic-millimeter computer*, volume 34. IEEE Computer, January 2001. [citato a p. 3]

Elenco di simboli e abbreviazioni

Abbreviazione	Descrizione	Definizione
TOS	TinyOS	page 71
WSN	Wireless Sensor Networks	page 1
HAA	Hardware Abstraction Architecture	page 74
HPL	Hardware Presentation Layer	page 74
HAL	Hardware Adaption Layer	page 74
HIL	Hardware Interface Layer	page 74
API	Application Program Interface	page 77
FIFO	First In First Out	page 82
AM	Active Message	page 83
ACK	Acknowledge	page 83
nesC	network embedded system C	page 93
MUTEX	Mutual Exclusion	page 98
MEMS	MicroElectroMechanical System	page 3
ADC	Analog to Digital Converter	page 2
GPS	Global System Position	page 17
RSSI	Receive Signal Strength Indication	page 27
TOSSIM	TinyOS mote SIMulator	page 49

Abbreviazione	Descrizione	Definizione
SNAFDLA	Sensor Network Anchor Free Distributed Localization Algorithm	page 25
AoA	Angle of Arrival	page 22
TDoA	Time Difference of Arrival	page 21
QoS	Quality of Service	page 8
CPU	Central Processing Unit	page 2

Elenco delle figure

Indice analitico