UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

SVILUPPO DI UN SISTEMA DI MONITORAGGIO E CONTROLLO AMBIENTALE A BASSO CONSUMO ENERGETICO PER RETI DI SENSORI WIRELESS

RELATORE: Ch.Mo Prof.re Luca Schenato

LAUREANDO: Nicola Franceschini

Padova, 16 Dicembre 2009

Indice

Sommario

Il lavoro svolto per questa tesi consiste nella progettazione e implementazione di WirMoS (Wireless Monitoring and control System), un sistema di monitoraggio e controllo ambientale, basato sulla tecnologia delle reti di sensori wireless. Questo sistema non è pensato per un particolare ambiente, ma al contrario può essere utilizzato in diversi contesti, con caratteristiche proprie, come ad esempio differenti frequenze di campionamento. I sensori wireless di WirMoS interagiscono tra loro per formare una topologia ad albero, e raccolgono dati riguardanti temperatura, umidità e luminosità dell'ambiente monitorato, e li trasmettono ad una workstation assieme ad informazioni sul loro stato. Sono inoltre in grado di ricevere dei comandi provenienti dalla workstation, per rispondere a delle interrogazioni, o eseguire particolari istruzioni. Il sistema è in grado di adattarsi alle variazioni della topologia della rete, a causa della diminuzione della qualità dei collegamenti, o dell'aggiunta di nuovi sensori. Sulla workstation risiede un'applicazione che fa da interfaccia con gli utilizzatori, elaborando, visualizzando e memorizzando dati e permettendo l'invio dei comandi per i sensori. Questo software implementa inoltre la funzione di controllo, pilotando degli attuatori che modificano i parametri ambientali monitorati. L'applicazione permette di controllare diverse reti contemporaneamente, e può essere acceduta anche via web, semplicemente utilizzando sulle postazioni remote un browser. Come strategia di base per ottenere il risparmio energetico, problema dominante nelle WSN, è stato utilizzato il protocollo Flexible Power Scheduling [1] che si basa sulla divisione del tempo in slot (TDMA). La funzione di controllo richiede basse latenze sui dati che vengono trasmessi alla workstation, per questo motivo sono stati studiati degli accorgimenti da utilizzare nel protocollo per ottenere questo risultato. Viene poi proposta un estensione del protocollo FPS, i *supercicli*, che rendono il sistema utilizzabile con frequenze di campionamento molto diverse tra loro, modificando in modo congruo i consumi, e lasciando inalterate le latenze. Inoltre si affianca alla suddivisone del tempo in slot, l'accesso multiplo a divisione di frequenza (FDMA), aggiungendo quindi un ulteriore dimensione sulla quale effettuare l'accesso al canale condiviso, e migliorando di conseguenza le prestazioni. La trattazione teorica, e le prove sperimentali fatte, dimostrano l'efficacia delle scelte progettuali adottate.

Capitolo 1

Introduzione

1.1 Applicazioni delle WSN e stato dell'arte

Una rete di sensori senza fili (WSN-Wireless Sensor Network), elemento principale del sistema realizzato, è una rete wireless auto configurante, con nodi che sono piccoli dispositivi, poco costosi, alimentati a batteria, e dotati di speciali sensori. Questi dispositivi vengono chiamati con diversi nomi: mote, piattaforme wireless, sensori wireless o anche solo sensori. L'obiettivo principale delle WSN è di collezionare dati dall'ambiente, e di trasmetterli in un particolare sito dove i dati vengono osservati e analizzati. I sensori wireless inoltre rispondono a interrogazioni inviate dal "sito di controllo", per eseguire specifiche istruzioni, o fornire campioni di rilevazione. Infine, una rete di sensori wireless può essere equipaggiata con attuatori che si attivano sotto certe condizioni [2].

I recenti miglioramenti tecnologici nei circuiti integrati, nei sensori, nelle comunicazioni wireless, e nell'immagazzinamento dell'energia, hanno permesso negli ultimi anni un grande sviluppo delle WSN. Queste reti aggiungono alle già esistenti reti di sensori cablate diversi vantaggi, ovvero la maggior flessibilità, semplicità e rapidità di installazione e manutenzione, e con un costo minore. Con le WSN si possono inserire sensori in ambienti dove la stesura dei cavi non è praticabile, risolvendo problematiche che con la sua tecnologia delle reti di sensori cablate rimarrebbero tali. Anche la modifica della rete, attraverso l'aggiunta e lo spostamento dei sensori, che nelle reti cablate è spesso complessa o addirittura non fattibile, diventa nelle WSN un'operazione estremamente facile. E' così sempre possibile migliorare la copertura della zona interessata, utilizzando la rete stessa come strumento di progettazione. Inoltre la soluzione wireless consente di realizzare reti in cui i sensori non sono fissi,

ma si possono spostare per seguire soggetti in movimento, ad esempio merci e generi alimentari, per tenere sotto controllo in tempo reale la qualità e lo stato di conservazione nella fase di trasporto dal produttore ai centri di distribuzione [3]. Un altro esempio di questo tipo di utilizzo, è il monitoraggio di persone in movimento, per rilevare i loro parametri fisiologici durante l'attivit'a fisica [4].

I settori che possono trarre benefico delle WSN sono molti, a partire dalle note applicazioni per reti di sensori cablate nel controllo industriale, buiding automation (automazione degli edifici) e home automation (domotica). Applicazioni emergenti riguardano invece l'utilizzo delle WSN in ambito militare, per tenere sotto controllo il campo di battaglia e lo spostamento delle truppe nemiche [5, 6], in quello agricolo, dove ad esempio sono allo studio impieghi per aumentare la produttivit'a delle coltivazioni nei paesi in via di sviluppo [7], e in quello medico, all'interno degli ospedali o per migliorare la qualit'a delle cure a domicilio [8]. Si trovano numerose applicazioni dei sensori senza fili anche nella ricerca scientifica per l'osservazione non intrusiva della fauna [9], per lo studio di fenomeni naturali [10], o per valutare la salute della vegetazione [11].

1.2 Sfide e soluzioni utilizzate

Le sfide nelle reti di sensori wireless sono molteplici dato che per motivi tecnologici ed economici i sensori hanno risorse limitate in termini di energia, potenza computazionale, memoria, e capacità di comunicazione. E' necessario affrontare problematiche tipiche dei sistemi distribuiti, come la sincronizzazione e la scalabilità, e delle reti wireless, come l'accesso al canale radio condiviso, la perdita di pacchetti, e il consumo energetico.

Quest'ultimo è il problema dominante nelle WSN, perché se non correttamente gestito limita fortemente la loro utilità. I sensori sono alimentati a batterie che vanno periodicamente cambiate o ricaricate. Quest'attiva può risultare molto laboriosa se effettuata troppo frequentemente, soprattutto quando i sensori si trovano in posti difficilmente accessibili. E' necessaria quindi una corretta gestione dei consumi, che permetta alle batterie dei sensori di durare per un intervallo di tempo che sia adeguato alla particolare applicazione. La gestione dei consumi può essere effettuata attraverso scelte progettuali riguardanti l'hardware o il software. Per quanto riguarda l'hardware in commercio è possibile trovare diverse piattaforme da poter utilizzare come nodi della rete, ognuna caratterizzata da un diverso consumo di energia, in

base alle tecnologie utilizzate per la loro realizzazione. Per implementare il sistema WirMoS sono stati utilizzati i Tmote Sky della Moteiv (ora Sentilla) che rispetto agli altri sensori, hanno un consumo di corrente molto ridotto. Una scelta sicuramente vincente per diminuire il tempo di intervento sui sensori, è l'utilizzo di piccoli pannelli fotovoltaici con cui equipaggiare i mote [13], soprattutto se sono posizionati all'aperto. Alcune aziende offrono piattaforme che già comprendono i pannelli fotovoltaici come ad esempio sistema eKo della Crossbow [12]. Nel sistema realizzato i pannelli fotovoltaici non sono stati utilizzati, e la loro integrazione resta quindi uno dei primi punti da sviluppare per proseguire il lavoro. Le soluzioni software per risparmio energetico consistono nell'utilizzo, di sistemi operativi nei sensori, e di protocolli nelle applicazioni che vengono eseguiti su questi sistemi operativi, che implementino una corretta gestione dei consumi. Nei Tmote Sky della soluzione progettata il sistema operativo utilizzato è il TinyOS [14], uno dei più conosciuti e utilizzati sistemi operativi open source per WSN, che tra i vari obiettivi principali ha quello di ottenere il risparmio energetico. Per quanto riguarda le applicazioni, è possibile trovare molti articoli che espongono soluzioni per limitare i consumi attraverso due principali strategie: la riduzione della potenza di trasmissione, oppure spegnere i nodi quando sono nello stato idle [2]. La riduzione della potenza di trasmissione ha come effetto immediato l'eliminazione di alcuni link diretti, e forza i pacchetti ad effettuare più hop. Quindi il risparmio energetico ottenuto con questo metodo non è solo dovuto al fatto che i nodi trasmettono con una potenza minore, ma anche all'utilizzo di una serie di comunicazioni multi-hop, invece che di lunghi link diretti, che permettono di ottenere una maggiore efficienza [2, 15]. La seconda strategia consiste nella suddivisione del tempo in slot temporali nei quali avviene un accordo su chi trasmette e chi riceve, permettendo ai nodi idle di commutare nella modalità di basso consumo energetico. E' questo il concetto principale dell'algoritmo FPS (Flexible Power Scheduling [1]), la strategia di base utilizzata per il risparmio energetico nel sistema progettato.

Per la sua esecuzione i nodi della rete devono essere disposti ad albero, la cui radice ha il compito di trasmettere i dati di tutti i sensori al punto di raccolta dati. Il sistema quando viene avviato esegue quindi una fase di inizializzazione, nella quale i nodi prima vengono a conoscenza di quali sono i loro vicini affidabili, ovvero quelli con i quali realizzano link con bassa probabilità di perdita di pacchetti, e poi tra questi individuano qual'é quello che permette di arrivare alla radice con il minor numero di hop, creando cosi il *minimun spanning tree* della rete, con una funzione

di costo che è il numero di hop. Dopo questa fase viene avviato il protocollo FPS. In FSP gli eventi sono periodici, e la periodicità è data dal ciclo FPS che è un insieme di n slot temporali consecutivi. Ogni nodo mantiene nella propria memoria uno schedule, un'array di lunghezza pari al numero di slot del ciclo. Ogni cella dell'array definisce se il nodo nel corrispondente slot, deve trasmettere qualcosa al genitore, riceve qualcosa da un figlio, oppure non deve far niente. Nell'ultimo caso il chip radio del nodo viene spento, e dato che gran parte degli slot in un nodo sarà di questo tipo, si ottiene in questo modo il risparmio energetico desiderato. Il Tmote Sky infatti consuma con radio accesa ben 4000 volte di più che con la radio spenta. Gli schedule sono diversi da nodo a nodo riflettendo le specifiche necessità di ognuno, e vengono creati attraverso lo scambio di informazioni di controllo tra genitore e figlio.

Il protocollo FPS oltre a limitare fortemente i consumi, permette anche di coordinare l'accesso al mezzo radio condiviso, poiché si comporta in modo simile al noto protocollo TDMA. Tuttavia essendo gli schedule locali, un nodo non ha la visione globale della rete, e possono quindi verificarsi delle collisioni di nodi che trasmettono nello stesso slot. Anche se l'algoritmo FPS riduce significativamente la contesa, al punto che per le collisioni rimanenti si serve del semplice schema CSMA, si è deciso utilizzare in sostituzione a quest'ultimo l'accesso multiplo a divisione di frequenza (FDMA). Utilizzando anche questa tecnica, i nodi oltre che a trasmettere in istanti temporali diversi, trasmettono anche su diverse frequenze. Dato che il Tmote Sky permette di scegliere tra 16 diversi canali, si ottiene un ulteriore e importante riduzione delle collisioni. Il vantaggio nell'utilizzo di questo schema al posto del CSMA, oltre che la maggior efficacia, sta nel fatto che se due o più nodi trasmettono nello stesso slot, si può cambiare il canale di trasmissione, invece che ritardare la trasmissione come accade nel CSMA. In questo modo si possono mantenere gli slot di breve durata, migliorando latenza e consumi. Inoltre è possibile scegliere dei canali dedicati per la sincronizzazione o le informazioni di controllo, migliorando ulteriormente le prestazioni.

La latenza, ovvero il tempo che intercorre dall'istante in cui un messaggio viene trasmesso da un qualsiasi nodo, e l'istante in cui viene ricevuto dal PC, è un altro importante parametro, soprattutto per quanto riguarda la funzione di controllo dell'ambiente monitorato. Se la latenza è troppo alta difatti, il controllo può diventare impossibile. Nel lavoro svolto si è cercato quindi di capire quali sono le impostazioni delle variabili del protocollo FPS, che permettono di minimizzare la latenza mantenendo una corretta esecuzione dell'algoritmo.

Se le variabili da monitorare variano lentamente, come nel caso della temperatura o dell'umidità in un ambiente grande, non è necessaria un alta frequenza di campionamento. Il tempo di campionamento nel protocollo FPS è dato dalla durata di un ciclo, quindi per avere un tempo di campionamento lungo è necessario aumentare la durata di un ciclo, aumentando il numero di slot che lo compongono, oppure la loro durata. Nel primo caso si ha una proporzionale diminuzione del consumo medio nel ciclo, in quanto aumentano gli slot dove la radio è spenta, però avviene anche un aumento della latenza. Nel secondo caso aumenta la latenza, e inoltre il consumo medio non diminuisce. Risulta quindi impossibile con il protocollo FPS, utilizzare una bassa di frequenza di campionamento, con conseguenti bassi consumi, e mantenere contemporaneamente latenze basse. Per risolvere questo problema si propone la strategia dei *supercicli*. Un superciclo è una sequenza n di cicli FPS consecutivi dove solo nei primi H (con H altezza dell'albero FPS) ci possono essere trasmissioni e ricezioni di dati, mentre per i restanti la radio rimane spenta. La periodicità delle trasmissioni dei campioni di rilevazione, è data ora dal superciclo, e la durata è di quest'ultimo è quindi il tempo di campionamento. In questo modo se si desidera aumentare il tempo di campionamento, è sufficiente aumentare il numero di cicli FPS dove la radio è sempre spenta, mantenendo inalterata la latenza. Sono necessari H cicli consecutivi in quanto un messaggio che parte dal ciclo i può arrivare alla radice nel caso peggiore nel ciclo $i + h - 1$. Le informazioni di controllo del protocollo FPS originale vengono trasmesse ad ogni ciclo per permettere alla topologia di variare. Dato che in molte applicazioni la variazione della topologia non è un evento che accade frequentemente, si è deciso di modificare quest'aspetto per limitare ulteriormente i consumi. Difatti soprattutto per basse frequenze di campionamento, il consumo di energia associato alla trasmissione e ricezione delle informazioni di controllo, è una porzione considerevole del consumo totale. Per questo motivo in WirMos le informazioni di controllo vengono inviate solo quando la rete avverte un cambiamento nella topologia, oppure quando l'utente desidera inserire un nuovo nodo.

Per poter inviare dei comandi, o effettuare delle interrogazioni sui nodi, si utilizzano due tipi particolari di slot nei quali la radio è periodicamente accesa. Questi slot vengono utilizzati anche per effettuare la sincronizzazione dei nodi, necessaria per eliminare il disallineamento tra gli slot di nodi differenti, che si viene a formare a causa della diversa oscillazione dei quarzi degli orologi dei mote. La sincronizzazione parte dalla radice e coinvolge tutti i nodi dell'albero inondando la rete dall'alto verso il basso, in questo modo ogni figlio è sempre ben allineato con il genitore.

Viene proposta un'altra estensione all'algoritmo FPS per permettere alla topologia di variare quando la qualità dei link decade, o quando si vuole aggiungere un sensore senza re-inizializzare la rete. Questo risultato si ottiene innanzitutto con il controllo su ogni nodo di quanti messaggi di dati sono stati ricevuti in un particolare slot; se il numero di messaggi ricevuti non è sufficientemente alto per uno slot, questo viene re-inizializzato. Un secondo accorgimento prevede che se un nodo riceve un messaggio contenente dei dati, esso risponda al mittente con un messaggio Acknowledge. Infine ogni nodo controlla periodicamente la percentuale Acknowledge ricevuti, rispetto al numero di messaggi inviati, e se questa non è sufficientemente alta re-inizializza tutti gli slot di trasmissione dello schedule, e sceglie un nuovo genitore con i dati raccolti nella fase di inizializzazione. Si utilizzano nuovamente queste informazioni, perché lo spostamento dei nodi senza reinizializzazione della rete non è in questa implementazione consentita. Quando la radice dell'albero si accorge dei cambiamenti, invia un comando che viene propagato a tutti i nodi, e che abilita la trasmissione dei messaggi di controllo. Questi messaggi servono al nodo sconnesso per decidere in quali slot FPS trasmettere al nuovo genitore. Anche quest'ultimo per soddisfare l'esigenza nel nuovo figlio, si accorderà con il proprio genitore su quali slot trasmettere, e il processo proseguirà fino a quando arriverà alla radice. L'aggiunta di nuovi sensori senza re-inizializzare la rete avviene in modo simile. Quando c'è la necessità di aggiungere un nuovo nodo, un comando dell'utente attiva l'invio delle informazioni di controllo. Il nuovo nodo si sincronizza quindi con un nodo qualsiasi. Dopo alcuni scambi di informazioni che permettono di decidere al nuovo nodo chi è il suo genitore, viene avviato il protocollo FPS, e il procedimento avanza come per il caso precedente.

Per consentire all'utente di interagire con il sistema sono state realizzate due applicazioni per PC. La prima, WirMoS-Dashboard, permette di visualizzare i dati provenienti dai sensori, archiviarli su memoria secondaria, creare grafici sui dati memorizzati, controllare e interrogare i nodi, e pilotare in modo automatico gli attuatori. WirMoS-Dashboard può essere utilizzata sia per il normale esercizio del sistema di monitoraggio e controllo, sia nella fase di progettazione, per identificare al meglio quali sono le esigenze della funzione di controllo. La seconda applicazione è WirMoS-ReadNode permette di leggere e visualizzare dati di logging, che vengono memorizzati nei nodi durante il loro funzionamento.

1.3 Descrizione capitoli

La parte restante del testo è organizzata in questo modo:

Nel *secondo capitolo* vengono descritte le tecnologie utilizzate per implementare il sistema. In primo luogo le wireless sensor network: i sensori wireless, le sfide tecnologiche che si incontrano nella loro progettazione e realizzazione, le varie topologie utilizzate attualmente, i protocolli e gli standard specifici di questa tecnologia. Nel secondo paragrafo del capitolo vengono presentate le caratteristiche principali del Tmote Sky, la piattaforma wireless utilizzata per il progetto. Infine nell'ultimo paragrafo vengono illustrati i concetti base del sistema operativo TinyOS e del linguaggio di programmazione nesC, utilizzati per programmare il Tmote Sky. Il paragrafo presenta il modello di programmazione di TinyOS/nesC (componenti, interfacce, moduli e configurazioni) e il modello di esecuzione (task e gestori degli interrupt hardware).

Nel primo paragrafo del *terzo capitolo* viene illustrata l'architettura di sistema del progetto, ovvero le componenti che lo costituiscono, e le relazioni che ci sono tra di loro. Nel secondo paragrafo vengono descritte le sotto fasi dell'inizializzazione del sistema, che permettono di creare la topologia ad'albero richiesta dal protocollo FPS.

Il *quarto capitolo* descrive il funzionamento del sistema a regime. Viene spiegato il funzionamento del protocollo FPS, utilizzato per ottenere il risparmio energetico, come vengono evitate le collisioni sul mezzo radio condiviso, com'è possibile inviare un messaggio in broadcast a tutti i nodi, come viene effettuata la sincronizzazione, come mantenere minima la latenza dei messaggi inviati dai nodi, i supercicli, ovvero l'estensione dell'FPS proposta per gestire diverse frequenze di campionamento, e infine come poter permettere alla topologia della rete di variare.

Nel *quinto capitolo* vengono presentate le interfacce sistema-utente realizzate, WirMoS-Dashboard e WirMoS-ReadNode. Sono due applicazioni per PC con interfaccia grafica, la prima utilizzata per archiviare, elaborare e visualizzare i dati di una WSN, e per comandare, secondo funzioni specifiche, gli attuatori che forzano le condizioni ambientali del sito monitorato. Con WirMoS-ReadNode è possibile leggere e visualizzare dati di logging, che i sensori memorizzano nella loro memoria FLASH.

Il *sesto capitolo* tratta il controllo degli attuatori realizzato da WirMoS-Dashboard, e le performance del sistema, mostrando anche i risultati delle sperimentazioni.

1.4 Sigle utilizzate

Segue una lista in ordine alfabeto, del significato delle principali sigle utilizzate nei prossimi capitoli.

$\boldsymbol{AC(j)}$: Consumo medio di un nodo j.

ALPHA : Variabile presente in ogni nodo non Base Station che fornisce il valore di α.

$\boldsymbol{H}$: Altezza dell'albero che costituisce la topologia della WSN.

HEIGHT : Variabile presente in ogni nodo non Base Station che fornisce il valore di H.

$\boldsymbol{D(k)}$: Domanda (demand) FPS di un nodo all'inizio del $k-esimo$ ciclo.

$\boldsymbol{N_d(j)}$: Numero di nodi discendenti del nodo j.

$\boldsymbol{N_{dn}(j)}$: Numero di nodi discendenti, non figli, del nodo j.

$\boldsymbol{N_n}$: Numero massimo di nodi del sistema.

$\boldsymbol{N_R(k)}$: Numero di slot di ricezione di un nodo all'inizio del $k-esimo$ ciclo.

$\boldsymbol{N_T(k)}$: Numero di slot di trasmissione di un nodo all'inizio del $k-esimo$ ciclo.

$\boldsymbol{N_{TS}}$: Numero di slot di trasmissione di un nodo utilizzati per se stesso (domanda iniziale).

$\boldsymbol{N_{slot}}$: Numero di slot dello schedule FPS.

$\boldsymbol{N_{slot}(j)}$: Numero il numero di slot minimo che deve avere lo schedule del il nodo j, perché si possa portare a termine le prenotazioni FPS.

$\boldsymbol{N_{so}(j)}$: Numero di nodi figli del nodo j.

NUM_NODES : Variabile della Base Station che fornisce il numero di nodi iniziale della rete.

$\boldsymbol{S(k)}$: Offerta (supply) FPS di un nodo all'inizio del $k-esimo$ ciclo.

$\boldsymbol{T_c}$: Durata di un ciclo FPS.

$\boldsymbol{T_{re}}$: Intervallo di tempo tra una sincronizzazione e l'altra.

$\boldsymbol{T_s}$: Durata di uno slot FPS.

$\boldsymbol{T_{sc}}$: Durata di un superciclo, che corrisponde al tempo di campionamento effettivo del sistema.

$\boldsymbol{T_{su}}$: Tempo di campionamento scelto dall'utente.

$\boldsymbol{\alpha}$: Numero cicli FPS in un superciclo.

$\boldsymbol{\delta t}$: Massimo disallineamento temporale tra gli slot dello schedule dei nodi.

Capitolo 2

Tecnologie utilizzate

2.1 Wireless Sensor Networks

In base alla definizione e alle applicazioni viste nel capitolo introduttivo, la tecnologia delle WSN dovrebbe soddisfare i seguenti requisiti:

- I nodi devono essere dispositivi di piccole dimensioni e a basso costo.
- Bassi consumi.
- Utilizzo bande radio non licenziate.
- Scalabilità: supportare un grande numero di nodi.
- Flessibilità: deployment e estensione della rete semplici.
- Un basso date rate è sufficiente.
- Supportare la sicurezza dei dati e della rete.

Tra le tecnologie wireless esistenti, quella delle WSN è l'unica che punta ad una semplice comunicazione, con un basso data rate, e un basso consumo di energia, vedi Figura 2.1.

2.1.1 I sensori wirless

L'architettura di un sensore wireless, del quale è stata data la definizione nel Paragrafo 1.1, è molto semplice. Come mostrato in Figura 2.2 un sensore wireless contiene un processore, una memoria, una ricetrasmittente con un'antenna radio, una sorgente di energia, e un'interfaccia imput/output che permette l'integrazione di sensori

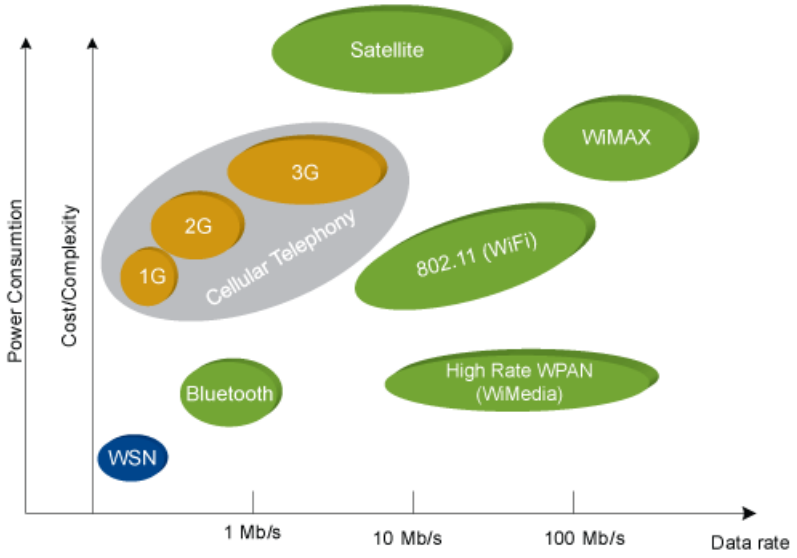


Figura 2.1: Posizione delle WSN tra le altre tecnologie wireless.

esterni al dispositivo wireless. I sensori wireless sono equipaggiati con microcontrollori a basso consumo energetico e poca potenza computazionale. I microprocessori sono spesso esclusi dalle WSN a causa del loro costo, e perché si suppone che i sensori debbano eseguire dei processi semplici e specializzati. Inoltre i microcontrollori consumano molta meno energia delle CPU, e questa è una caratteristica molto importante dato che i sensori wireless alimentati da batterie. In aggiunta a questo, i microcontrollori sono molto adatti alle WSN, perché alcune sue parti possono essere disattivate se non necessarie, diminuendo ulteriormente il consumo energetico.

Nei sensori wireless c'é anche la necessità di una Random Access Memory (RAM) per memorizzare i dati delle applicazioni, come le letture dei sensori, e di una Read-Only Memory (ROM) per memorizzare in modo permanente, anche quando l'alimentazione viene a mancare, il codice del programma. La memoria RAM è spesso inclusa nel microcontrollore e limitata a qualche kilobytes. Normalmente i sensori wireless includono anche EEPROM (Electrically Erasable Read-Only Memory), spesso di tipo FLASH, per i dati di configurazione. Queste memorie non solo preservano i dati nel caso di interruzione dell'alimentazione, ma possono essere utilizzate come memoria RAM quando necessario. La memoria FLASH è leggibile da dispositivi esterni sconnessi alla piattaforma attraverso l'interfaccia USB. I drive di tipo flash, o le memory stick, forniscono lo spazio necessario con un costo energetico minore rispetto alle memorie RAM e ROM.

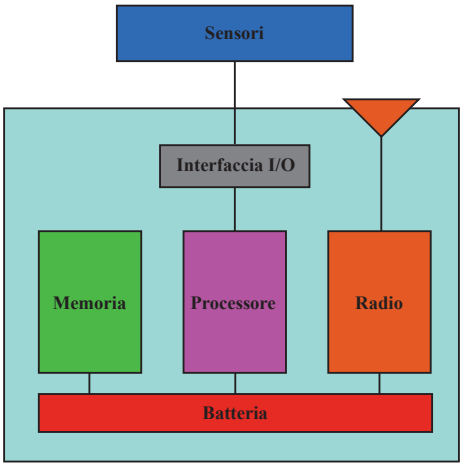


Figura 2.2: Architettura generale di un sensore wireless.

Un sensore wireless dispone anche di una ricetrasmittente radio, che contiene tutti circuiti necessari per spedire e ricevere dati sul canale radio: i moduli di modulazione e demodulazione, i convertitori analogico-digitale e digitale-analogico, amplificatori di potenza e low noise, filtri, mixer e antenna sono tra i componenti più importanti. La ricetrasmittente radio solitamente lavora in half-duplex, di conseguenza la ricezione e la trasmissione simultanea dei dati sul mezzo wireless è impraticabile. In modo simile ai microcontrollori, le ricetrasmittenti possono operare in differenti modalità. Normale, trasmissione, ricezione, idle e sleep sono modalità operative solitamente disponibili, ciascuna caratterizzata da un particolare consumo energetico. La modalità sleep è molto importante per il risparmio energetico delle WSN, perché in questa modalità la piattaforma ha il minor consumo. Comunque, l'accensione e lo spegnimento della ricetrasmittente devono essere gestiti attentamente, in quanto la prima operazione richiede un apprezzabile quantitativo energetico. Alcune ricetrasmittenti offrono capacità addizionali, tra le quali diverse modalità di sleep, a cui corrisponde l'accensione o lo spegnimento di diverse parti dell'hardware. Le ricetrasmittenti disponibili per le WSN hanno differenti caratteristiche e capacità. Normalmente lavorano su tre differenti intervalli di frequenze: 400 MHz, 800-900 MHz e 2.4 GHz o nella banda ISM (industriale, scientifica, medica). La potenza di trasmissione, gli schemi di modulazione, e la frequenza di trasmissione dei dati variano da produttore produttore.

I sensori wireless vengono solitamente alimentati con due batterie AA collegate

in serie. Le batterie AA disponibili in commercio hanno una capacità fino a 3 Ah, e forniscono una tensione di circa 1.5V.

Infine, i sensori wireless sono equipaggiati con delle schede per sensori, che contengono dei sensori specifici per l'applicazione. La varietà dei sensori e delle schede per sensori, che possono essere direttamente collegati al dispositivo wireless è molto ampia. Sensori di temperatura, luminosità, umidità, qualità dell'aria, pressione, magnetometri, ottici, acustici, accelerometri; questo è solo un piccolo campione dei tipi di sensori commercialmente disponibili. Questa flessibilità di interfacciamento è il motivo della grande popolarità delle WSN, rendendole una piattaforma generale per risolvere problemi pratici in molti domini applicativi [2].

2.1.2 Le sfide nelle WSN

Le reti di sensori wireless presentano una serie di sfide impegnative che richiedono ancora un considerevole sforzo di ricerca. Mentre alcune di queste sfide sono la diretta conseguenza della limitata disponibilità di risorse nei sensori wireless, e quindi sono molto specifiche delle WSN, altre sfide vengono continuamente affrontate dalla maggior parte delle tecnologie di rete. La seguente lista espone brevemente le sfide più importanti, che si presentato oggi nella progettazione e implementazione delle WSN [2].

- **Durata della vita della rete**: I nodi nelle WSN sono alimentati da batterie, quindi la durata della vita della rete dipende da quanto saggiamente viene utilizzata l'energia disponibile. Nelle reti di sensori wireless su larga scala, o nelle applicazioni pericolose, è impostante minimizzare il numero di volte che le batterie devono essere cambiate. E' desiderabile avere una durata della vita della rete che sia nell'ordine di uno o più anni. Per ottenere questo risultato si deve fare in modo che i sensori operino con un duty-cycle molto basso. Per esempio, il tipico consumo di energia di un microcontrollore è circa di 1 nJ per istruzione. Se il microcontrollore è alimentato da una batteria da 1 J è il nodo lavora continuamente per un giorno, non dovrebbe consumare più di 11 μW, il che non è fattibile nemmeno per il solo microcontrollore. Di conseguenza, l'uso delle modalità di sleep per il microcontrollore e la ricetrasmittente radio è cruciale nelle le lunghe operazioni delle WSN.

- **Scalabilità**: Alcune applicazioni richiedono centinaia o anche migliaia di sensori wireless. Per esempio, immaginiamo una WSN per monitorare il confine

tra due stati, o per monitorare un oleodotto. Queste WSN su larga scala presentano delle sfide mai viste in WSN con pochi sensori. Algoritmi e protocolli che lavorano bene su piccole reti non lavorano necessariamente bene anche in quelle grandi. Un tipico esempio è la funzione di routing. Nelle reti su piccola scala i ben noti algoritmi di routing proattivi o reattivi, che usano l'algoritmo del cammino minimo di Dijkstra, funzionano bene. Nelle WSN su larga scala invece quest'approccio non è efficiente per quanto riguarda il consumo energetico, e sono più adatti gli algoritmi di routing basati sulla posizione (location-based), nei quali la posizione di ogni nodo è nota e viene utilizzata per determinare i percorsi per le informazioni. Simili problemi di scalabilità si verificano per altre funzionalità delle reti.

- **Interconnettività**: Le WSN hanno la necessità di essere interconnesse in modo tale che i dati raggiungano la destinazione desiderata per essere memorizzati, analizzati, e per intraprendere le opportune azioni. Ci si immagina che le WSN possano essere interconnesse con molte tecnologie di rete differenti, ad esempio con la rete dei cellulari, la rete Internet, reti wirelless ad hoc etc. . Questa operazione però risulta più difficile di quello che solitamente si immagina. Nuovi protocolli e meccanismi devono essere progettati per ottenere queste interconnessioni, e permettere il trasferimento dei dati da, e verso le WSN. Normalmente queste interconnessioni sono gestite attraverso l'uso di gateway, che richiedono nuove capacità per la scoperta delle reti, e la traduzione di differenti protocolli di comunicazione.

- **Affidabilità**: I sensori wireless sono dispositivi economici con un tasso di guasto abbastanza alto. Inoltre, in molte applicazioni questi dispositivi vengono lanciati sull'area di interesse da un elicottero, o veicoli simili. Come risultato, diversi nodi si guastano, oppure si guastano parzialmente alterando il loro normale funzionamento. L'affidabilità dei nodi dipende anche molto dall'energia disponibile nel nodo stesso.

- **Eterogeneità**: Nelle nuove WSN vengono utilizzati assieme sensori wireless con differente capacità e funzionalità. Questa eterogeneità richiede lo sviluppo nuovi algoritmi e protocolli di comunicazione. Per esempio, le architetture basate sui cluster possono utilizzare dispositivi con più potenza per aggregare i dati, e trasmettere informazioni per conto di nodi con limitate risorse. Tale

strategia però include la necessità di algoritmi di clustering e di aggregazione dei dati che non sono di banale progettazione.

- **Privacy e sicurezza**: La privacy e la sicurezza sono preoccupazioni all'ordine del giorno nel networking, e le WSN non sono un'eccezione. Comunque, i meccanismi di sicurezza solitamente richiedono molte risorse, che nei sensori wireless invece sono limitate. Quindi sono necessari nuovi algoritmi di sicurezza che richiedono poca potenza computazionale e energia.

2.1.3 Topologie di rete

La seguente lista illustra le più comuni topologie di rete utilizzate nelle reti di sensori wireless.

- *Stella* - Un nodo, detto centro della stella, funge da coordinatore della rete. Un qualsiasi nodo per comunicare con un altro nodo invia il messaggio al coordinatore, il quale lo inoltra al destinatario, vedi Figura 2.3(a). Tutti i messaggi quindi transitano attraverso il centro stella, e un messaggio compie al massimo due hop. Questa è la topologia più semplice, che permette quindi l'utilizzo di protocolli e algoritmi altrettanto semplici e di facile implementazione. Per esempio le informazioni di routing sono statiche, e minimali, in quanto ogni nodo non coordinatore comunica direttamente solo con quest'ultimo. Generalmente il nodo coordinatore, a causa del maggior carico di lavoro, ha un consumo di energia tale da obbligare, per la sua alimentazione, l'utilizzo della rete elettrica, mentre gli altri nodi possono utilizzare delle batterie. La semplicità di questa topologia si paga con l'impossibilità di realizzare reti con dimensioni che vanno oltre la copertura radio dei sensori. Un altro svantaggio risiede nel fatto che la rete non è molto robusta, in quanto il coordinatore è un *Single Point of Failure*, ovvero se si guasta, tutta la rete smette di funzionare.

- *Mesh* - Nelle reti mesh o peer to peer, ogni nodo può potenzialmente comunicare con ogni altro nodo nel suo raggio di copertura radio, vedi Figura 2.3(b). Si parla di topologia "full mesh" se ogni nodo, durante il normale funzionamento della WSN, comunica con tutti gli altri nodi, di "partial mesh" altrimenti. I nodi sono considerati tutti uguali, e quindi possono avere tutti le stesse prestazioni. Questa topologia viene definita di tipo *multihop*, ovvero un messaggio per arrivare a destinazione può attraversare più nodi intermedi con funzionalità di inoltro. In questo modo è possibile realizzare reti di dimensioni

arbitrariamente grandi. Rispetto alla topologia a stella inoltre, si ha il vantaggio di avere percorsi ridondanti, che consentono l'aumento dell'affidabilità e della robustezza, richiedendo però l'utilizzo dei algoritmi più complessi.

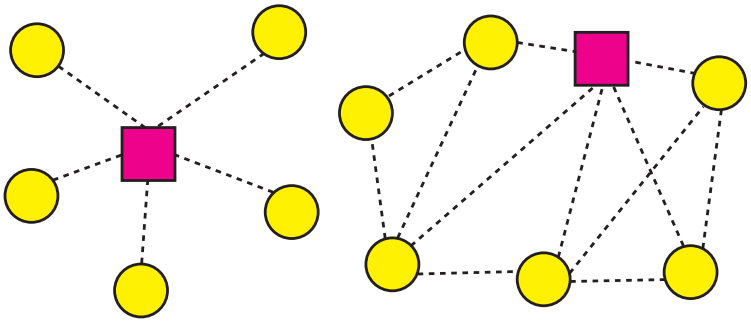
- *Albero* - Nella topologia ad albero come dice il nome stesso, i nodi formano logicamente un albero, vedi Figura 2.3(c). I messaggi generalmente partono da un nodo per risalire l'albero e arrivare alla radice (detta *sink*), che svolge la funzione di punto di raccolta dei dati e coordinatore della rete. Per questo motivo i nodi hanno un carico di lavoro che aumenta al diminuire della loro profondità. Con questa topologia si deve prestare attenzione alla radice, che essendo il nodo che svolge più lavoro di tutti, corre il rischio di risultare sovraccaricata. Il vantaggio di questa topologia, rispetto a quella mesh, è la riduzione dei percorsi di comunicazione possibili, consentendo lo sviluppo di sistemi di gestione meno complessi. Rispetto alla topologia a stella invece ha il vantaggio permettere la creazione di reti arbitrariamente grandi, utilizzando il multihop come nella topologia mesh.

2.1.4 Protocolli utilizzati nelle WSN

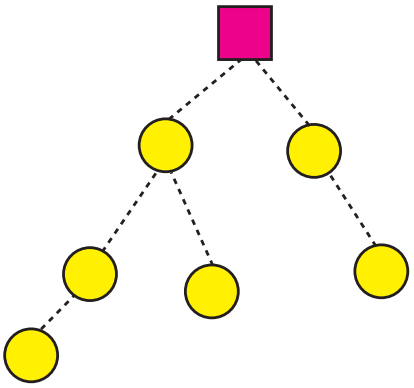
Lo standard maggiormente utilizzato nelle reti di sensori wireless è l'IEEE 802.15.4 che definisce i livelli fisico e MAC per le LOWPAN, wireless personal area network con basso data-rate. I dispositivi interessati hanno un basso data-rate, un basso consumo energetico, e semplici trasmissioni a corto raggio in radiofrequenza. Lo standard utilizza il meccanismo di accesso al mezzo CSMA-CA (carrier sense multiple access with collision avoidance) e supporta topologie a stella, ad albero e mesh. L'accesso al mezzo è basato sulla contesa, ma utilizzando la struttura opzionale *superframe*, degli slot temporali possono essere allocati dal coordinatore della PAM, per dispositivi che trasmettono dati con stringenti vincoli temporali. Le ricetrasmittenti conformi al protocollo operano in una delle seguenti tre bande non licenziate:

- 868-868.8 MHz: Europa, canale 0
- 902-928 MHz: Nord America, canali 1-10.
- 2400-2483.5 MHz: Internazionale, canali 11-26.

Esistono molti protocolli, sia standard che proprietari, realizzati sopra lo standard IEEE 802.15.4 per definire i livelli rimanenti dello stack di protocolli di rete.



(a) Topologia a stella. (b) Topologia mesh o peer to peer.



(c) Topologia ad albero.

Figura 2.3: Topologie di rete per WSN.

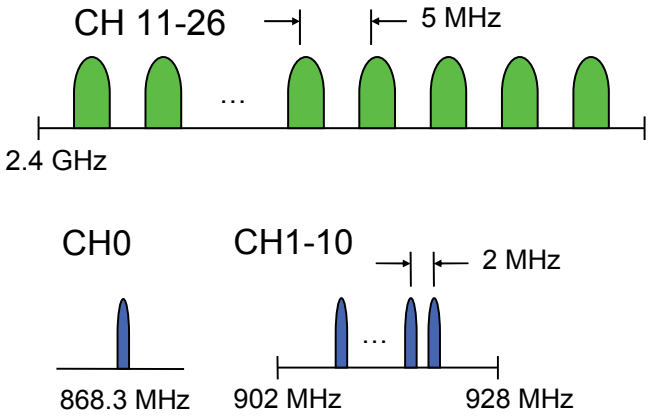


Figura 2.4: Canali IEEE 802.15.4.

La Tabella 2.1 mostra i più conosciuti con un'indicazione della loro popolarità, data dal numero di risultati trovati da Google, cercando il protocollo sul motore di

ricerca. Nel sistema progettato si utilizza lo standard IEEE 802.15.4 implementato nel Tmote Sky. Per implementare i livelli superiori dello stack invece, non è stato utilizzato nessuno dei protocolli di Tabella 2.1, ma è stato progettato un protocollo custom.

Protocollo	**Risultati**
ZigBee	1.650.000
6lowpan	69.600
WirelessHART	51.200
DigiMesh	21.400
ISA100	17.600

Tabella 2.1: Protocolli over IEEE 802.15.4 .

ZigBee, 6lowpan, WirelessHART e ISA100 sono degli standard, mentre DigiMesh è un formato proprietario dell'azienda Digi International. I protocolli standard sono vantaggiosi, perché favoriscono l'interoperabilità, sono realizzati da gruppi esperti del settore, permettono di accedere ad un grande quantità di conoscenza, e si può disporre delle specifiche e spesso anche di codici sorgenti. I protocolli proprietari d'altra parte vengono creati e aggiornati più velocemente, in quanto non necessitano di ottenere il consenso di differenti compagnie. Inoltre sono migliori in determinate situazioni, questo perché non seguendo uno standard possono essere ottimizzati per specifiche applicazioni. In alcuni casi anche per i protocolli proprietari si può ottenere le specifiche o il codice sorgente, anche gratuitamente. Segue una breve descrizione di questi protocolli.

- **ZigBee**: E' il protocollo più conosciuto nell'ambito delle low rate wireless personal area network. Viene mantenuto e pubblicato dalla ZigBee Alliance, un gruppo di compagnie formato da più di 150 membri distribuiti su tre livelli di menbership. Le specifiche del protocollo sono proprietà intellettuale della ZigBee Alliance, ma possono essere scaricate gratuitamente dal loro sito per scopi non commerciali. Per utilizzare il protocollo per scopi commerciali si deve essere membri dell'alleanza, e la quota annuale minima, ovvero quella di primo livello, è attualmente di $3.500 (USD) più $500 per altro SKU. ZigBee supporta le topologie a stella, cluster tree e mesh, ma i router, ovvero i nodi che sono in grado di ricevere un messaggio da un nodo e inoltrarlo verso un altro,

devono essere continuamente in ascolto, e quindi non possono essere alimentati a batteria [18]. Lo standar ZigBee ha tre versioni: 2004, 2006, 2007. ZigBee 2004 non è più utilizzato e ZigBee 2006 delle significanti limitazioni. Sebbene sia un protocollo molto conosciuto e diffuso, a tutt'oggi non ci sono prodotti specificatamente realizzati per l'utente finale. Poiché la ZigBee alliance definisce solo delle specifiche, esistono molte implementazioni dello stack ZigBee, come eZeeNet della MeshNetics, BeeStack della Freescale, AirBee-ZNS della AirBee, EmberZNet PRO Stack della Ember. Alcune implementazioni sono open source come Z-Stack della Texas Instrument, lo stack della Microchip, lo stack di Open-ZB basato su TinyOS, MSState_LRWPAN dell'Università del Mississipi [17], quest'ultimo però implementa solo un piccolo sottoinsieme dello standard ZigBee.

- **6LoWPAN**: La sigla 6LoWPAN è una specie di acronimo di IPv6 over Low Power Wireless Personal Area Networks. E' un nuovo standard proposto, basato sulla IETF RCF 4944 datata settembre 2007. Questo standard permette di incapsulare pacchetti IPv6 all'interno del payload 802.15.4, e di sfruttare la suite esistente di protocolli internet TCP/IP. Con questo protocollo i nodi di una WPAN hanno il proprio IP e quindi possono essere connessi direttamente ad Internet. Lo standard è molto nuovo, e sono probabili diversi cambiamenti. Attualmente con questa tecnologia è possibile realizzare solamente delle reti con topologia a stella, e inoltre non esistono molti chipset che la supportano. Arch Rock [25] è un'azienda di San Francisco che per prima, nel marzo del 2007, ha messo in vendita una soluzione commerciale di rete di sensori basata sul draft 6LoWPAN, e attualmente propone soluzioni basate su questo standard. 6loWPAN è uno standard aperto e per questo motivo esistono diverse implementazioni open source dello stack.

- **WirelessHART**: E' la variante wireless dello standard industriale HART, la cui prima versione risale al 1989. E' uno standard per l'automazione industriale e il controllo dei processi. WirelessHART utilizza il protocollo TSMP (Time Synchronized Mesh Protocol) creato dalla Dust Networks, basato sulla tecnica TDMA (Time Division Multiple Access). Un nodo speciale detto Gateway è responsabile per la sincronizzazione del tempo, e non può essere alimentato a batteria. Tutti gli altri nodi, sebbene possano funzionare da router, non è necessario che siano sempre in ricezione, e quindi possono essere alimentati a batteria. Questi nodi consumano pochissima energia, perché la maggior parte

del tempo sono spenti [18]. **ISA100** è molto simile a WirelessHART e sostanzialmente risolve lo stesso problema, ovvero viene utilizzato per l'automazione industriale e il controllo di processi, tanto che potrebbero confluire in un unico standard. Lo stack di ISA100 incorpora una parte di 6LoWPAN. Gli standard industriali possono essere utilizzati in modo vantaggioso anche nelle applicazioni di building automation (automatizzazione degli edifici) o home automation (automatizzazione delle case), anche se per queste applicazioni sono sufficienti protocolli meno complessi e costosi [19].

- **DigiMesh**: E' simile a WirelessHART e permette infatti di alimentare i nodi router a batteria con un bassissimo consumo. Supporta pienamente la topologia mesh e i nodi vengono considerati tutti router. L'accesso al canale è una sorta di CSMA sincronizzato temporalmente.

2.2 Tmote Sky

La piattaforma mote Tmote Sky è stata progettata dagli sviluppatori di TinyOS nell'Università della California Berkeley, e prodotta dalla MoteIV Corporation. Le versioni precedenti sono la piattaforma Telos Revision A e Telos Revision B. Dal 2007 MoteIV ha cambiato nome in Sentilla e ha interrotto la produzione e il supporto per la sua linea di prodotti Tmote, in favore di una nuova piattaforma hardware progettata per le applicazioni Java. Comunque la nuova piattaforma è retro compatibile con Tmote Sky, e inoltre è possibile acquistare dall'azienda Crossbow il mote TelosB, le cui funzionalità sono identiche al Tmote Sky. La Tabella 2.2 illustra le caratteristiche chiave della piattaforma, e la Figura 2.5 il diagramma a blocchi.

Microcontrollore	TI MSP430F1611 (10kB RAM, 48kB Flash)
Flash memory	ST M25P80 (1024KB)
Ricetrasmittente	Chipcon CC2420 (250kbps 2.4GHz IEEE 802.15.4)
Antenna	Integrata (copertura 50m interno-125m esterno)
Sensori	Temperatura, umidità, luce integrati.
Convertitori	ADC, DCA integrati
Consumi	min 5μA, max 21mA
Sleep Wakeup	$< 6\mu$S

Tabella 2.2: Caratteristiche chiave Tmote Sky.

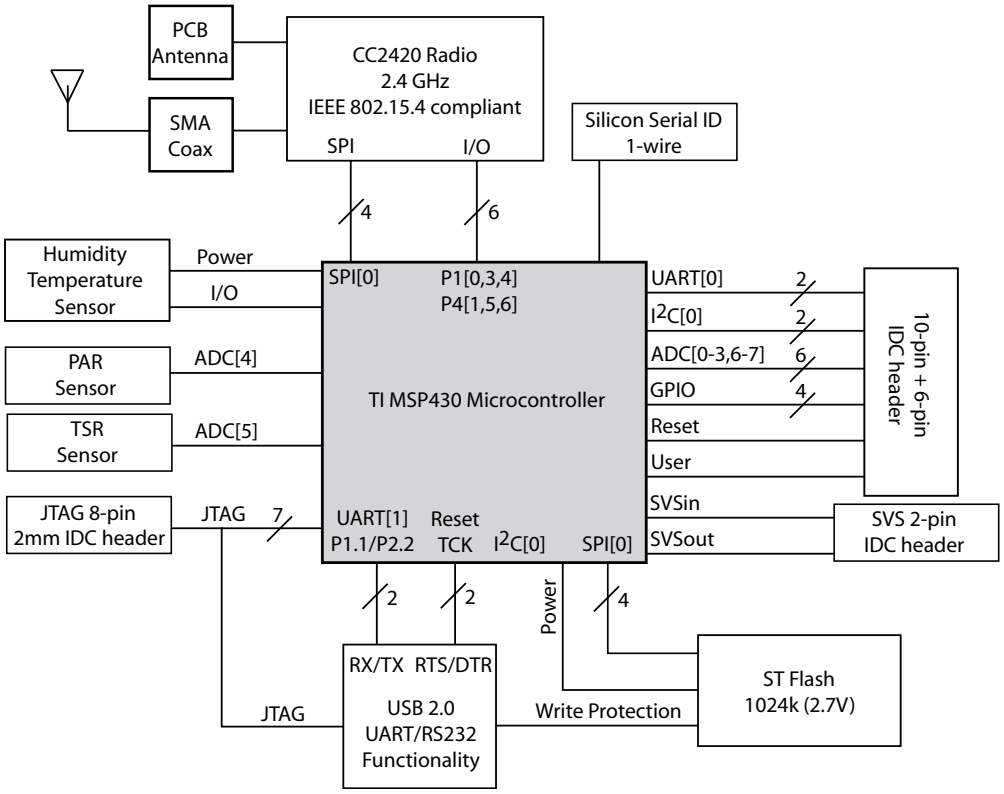
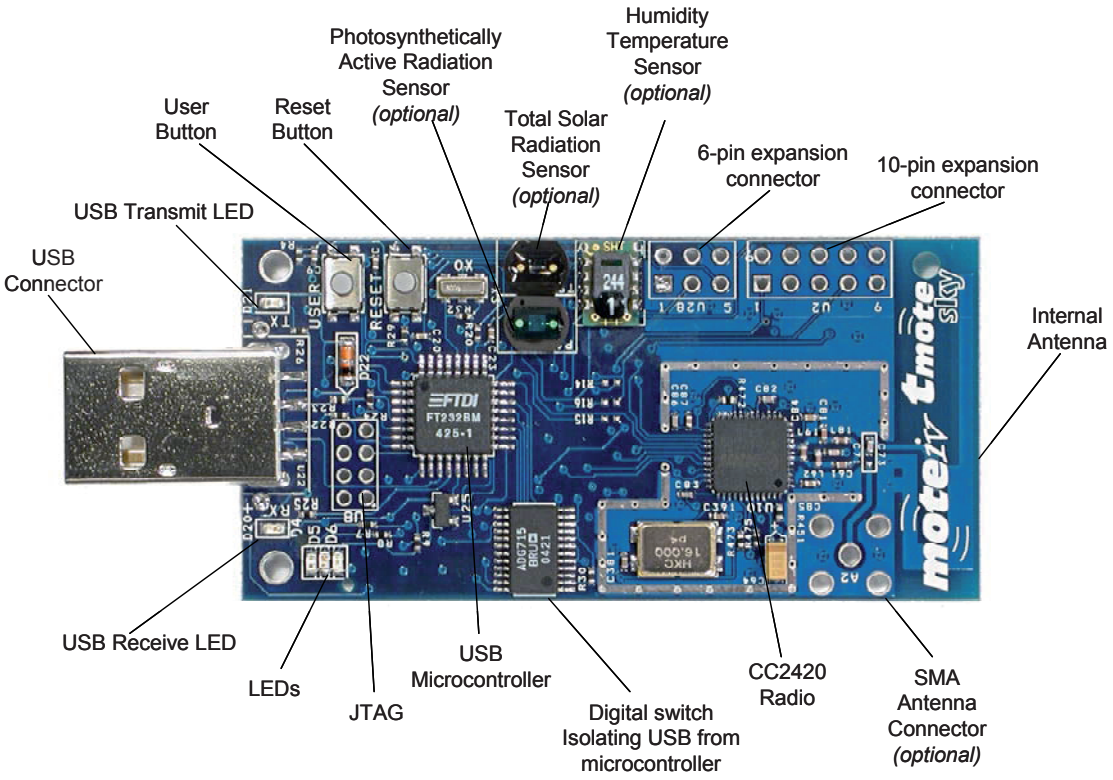


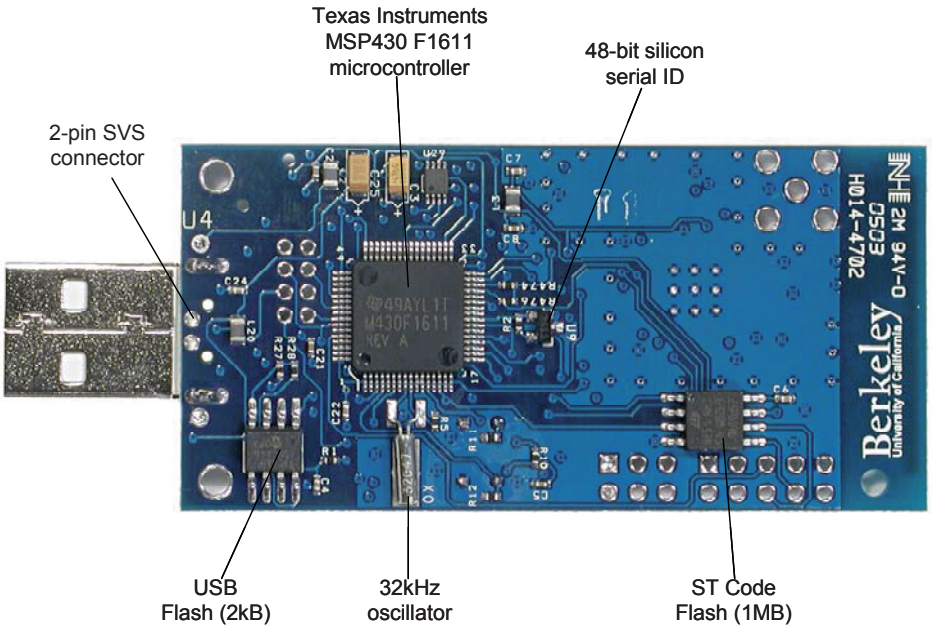
Figura 2.5: Diagramma a blocchi funzionali del modulo Tmote Sky.

Il modulo incorpora la MCU (MicroController Unit) MSP430F1611 della Texas Instruments con architettura RISC a 16 bit, che lavora alla frequenza di 8MHz. Questo microcontrollore ha una program memory FLASH da 48KB e una RAM da 10KB, convertitori analogico-digitale e digitale-analogico a 12 bit.

Il chip radio utilizzato dal Tmote Sky per le comunicazioni wireless è il CC2420, a basso consumo, basso voltaggio e basso costo, prodotto dalla Chipcom. Il CC2420 è conforme allo standard IEEE 802.15.4 fornendo il livello fisico e parte del livello MAC descritti nelle specifiche IEEE. La trasmissione avviene sulla banda dei 2.4GHz dell'IEEE 802.15.4, quindi si possono utilizzare i canali dall'11 al 26 dello standard, vedi Figura 2.4. Il data rate effettivo è 250 kbps. Il CC2420 fornisce un ampio supporto hardware per la gestione dei pacchetti, buffering dei dati, trasmissioni burst (brevi periodi di trasmissione intervallati da lunghi intervalli di inattività), critto-



(a) Vista dall'alto.



(b) Vista dal basso.

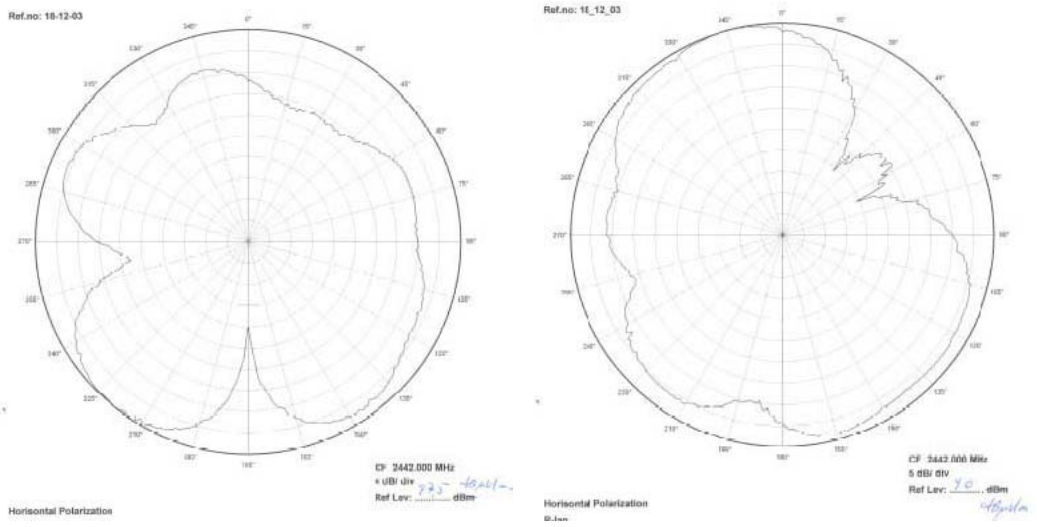
Figura 2.6: Piattaforma Tmote Sky.

PA_LEVEL	TXCTRL register	Output Power [dBm]	Current Consumption [mA]
31	0xA0FF	0	17.4
27	0xA0FB	-1	16.5
23	0xA0F7	-3	15.2
19	0xA0F3	-5	13.9
15	0xA0EF	-7	12.5
11	0xA0EB	-10	11.2
7	0xA0E7	-15	9.9
3	0xA0E3	-25	8.5

Tabella 2.3: Alcune configuarzioni della potenza in uscita del CC2420.

grafia dei dati, autenticazione dei dati, clear channel assessment, indicazione della qualità dei link e informazioni temporali sui pacchetti. Non tutte le caratteristiche dello standard IEEE 802.15.4 sono implementate, e per ottenere una conformità completa, le funzioni rimanenti devono essere realizzate via software. Il chip viene comandato dal controllore MSP430 attraverso la porta SPI, e una serie di linee I/O e interrupt, vedi Figura 2.5. L'alta configurabilità del CC2420 permette di soddisfare al meglio la specifica applicazione. Attraverso dei registri di configurazione è possibile programmare la modalità di ricezione e trasmissione (es. invio Acknowledge), il canale su cui avviene la trasmissione, la potenza di comunicazione, e altri parametri. La configurazione di default fornisce la conformità allo standard IEEE 802.15.4. La possibilità di impostare la potenza di trasmissione è una caratteristica molto desiderabile in quanto, come già anticipato, il consumo del chip radio domina il consumo totale del mote. La Tabella 2.3 tratta dal datasheet del Tmote Sky mostra alcuni livelli di potenza di trasmissione con il relativo consumo energetico (il livello è impostabile da 0 a 31, per un totale di 32 livelli). E' possibile inoltre conoscere l'RSSI (received signal strength indicator) relativo ad ogni ricezione effettuata.

Il Tmote Sky può essere alimentato con due batterie di tipo AA. Il modulo è stato progettato per essere adatto al fattore di forma delle batterie AA. L'alimentazione deve essere compresa tra gli 1.8 V e i 3.6 V, ma deve essere di almeno 2.7 V, quando si programma la flash del microcontrollore o la flash esterna. Quando il modulo è connesso alla porta USB per la programmazione o per la comunicazione con un PC, può ricevere l'alimentazione da quest'ultimo; la tensione operativa in questo caso è di 3 V. Se il mote è costantemente connesso alla porta USB il pacco di batterie non è necessario. La Tabella 6.2 mostra le condizioni operative tipiche del Tmote Sky. L'alimentazione può essere fornita anche attraverso i pin 1 e 9 del connettore di espansione U2, oppure attraverso i terminali dedicati alla batteria. In nessun



(a) In posizione orrizontale. (b) In posizione verticale.

Figura 2.7: Diagramma di radiazione dell'antenna ad F invertita del Tmote Sky.

punto la tensione deve però essere superiore ai 3.6 V, altrimenti si corre il rischio di danneggiare il microcontrollore, la radio, o altre componenti.

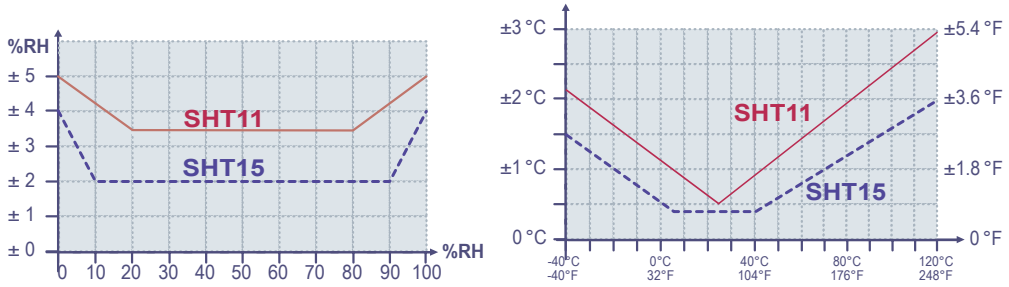
L'antenna usata dalla piattaforma è un'antenna a F invertita stampata sulla scheda, con un diagramma di radiazione quasi omnidirezionale con una copertura di 50 metri all'interno, e fino a 125 metri all'esterno. In Figura 2.7 sono riportati diagrammi di radiazione dell'antenna, quando il mote si trova in posizione verticale e orizzontale.

La memoria EEPROM utilizzata nel Tmote Sky è la M25P80 della STMicroelettronics. E'una memoria FLASH che può memorizzare fino a 1024kB di dati, ed è composta da 16 segmenti, ciascuno di 64kB. La memoria condivide la linea di comunicazione SPI con la ricetrasmittente CC2420. Bisogna quindi prestare attenzione quando le letture o scritture della flash sono intervallate da comunicazioni radio, in quanto non possono avvenire contemporaneamente. La Tabella 2.4 mostra le condizioni operative tipiche della M25P80. Si può osservare nella tabella che i consumi di corrente per le operazioni di scrittura e lettura sono molto alti. Per questo motivo per ottenere il risparmio energetico è necessario limitare più possibile l'utilizzo della memoria.

Il modulo Tmote Sky può essere provvisto di un sensore di umidità/temperatura direttamente montato sulla scheda in posizione U3, prodotto da Sensirion AG. I modelli utilizzabili sono SHT11 e SHT15 con differente accuratezza della misura effettuata, si veda Figura 2.8. Anche i sensori di luce possono essere direttamente

	MIN	NOM	MAX	UNIT
Supply voltage during flash memory programming	2.7		3.6	V
Operating free air temperature	-40		85	°C
Erase/Programming cycles			100,000	cycles
Data Retention			20	years
Active current (READ)			4	mA
Active current (WRITE/ERASE)			20	mA
Standby current		8	50	µA
Deep Power Down current		1	10	µA

Tabella 2.4: Condizioni operativi tipiche M25P80.



(a) Accuratezza umidità relativa. (b) Accuratezza temperatura.

Figura 2.8: Confronto accuratezza dei sensori Sensiorion SHT11 e SHT15.

montati sulla scheda, che dispone delle connessioni per due fotodiodi. La MoteIV utilizzava per i Tmote Sky i fotodiodi della Hamamatsu Corporation S1087 per rilevare la radiazione fotosinteticamente attiva, e S1087-01 per rilevare l'intero spettro visibile incluso gli infrarossi. Il primo è un sensore di radiazione solare in grado di misurare la radiazione sulle lunghezze d'onda 320-730 nm. Poiché questo spettro di lunghezze è quello assorbito dai vegetali per la fotosintesi clorofilliana, si riescono a monitorare con questo sensore i processi di fotosintesi.

2.3 TinyOS e nesC

TinyOS è un sistema operativo open-source progettato per le reti di sensori wireless. La prima versione del sistema (v0.43) è stata rilasciata nel 2000, mentre l'ultima (v2.1) è nel 2008. TinyOS è stato originariamente sviluppato come un progetto di ricerca all'Università della California Berkeley, ma la sua popolarità è cresciuta fino ad avere una comunità internazionale di sviluppatori e utilizzatori. La sua archi-

tettura è basata sui componenti, TinyOS è infatti composto da un grande numero di piccoli componenti riutilizzabili. L'ingegneria del software a componenti è un processo che pone l'accento sulla progettazione e costruzione di sistemi software, utilizzando componenti pronti per l'uso, standardizzati, e riutilizzabili, in grado di adattarsi ad ogni stile di architettura per un determinato dominio di applicazioni. I componenti sono inoltre indipendenti tra loro, e per utilizzarli non è necessario conoscere il funzionamento interno. I sistemi basati su componenti sono così più facili da assemblare, modificare ed espandere, e pertanto presentano costi inferiori. Oltre a ciò l'ingegneria del software a componenti incoraggia l'uso di schemi di architettura prevedibili, e di un'infrastruttura software standard, portando così a risultati di qualità superiore [20]. L'architettura a componenti di TinyOS permette di minimizzare la dimensione del codice, come richiesto dalle memorie ridotte dei sensori wireless. Infatti quando un'applicazione viene installata su un sensore, assieme viene installato anche TinyOS, ma grazie all'architettura a componenti, solo i componenti che sono strettamente necessari all'applicazione vengono installati. Per questo motivo il software che deve essere installato su un sensore occupa pochissima memoria. Oltre che per piattaforme con memoria ridotta, TinyOS è stato progettato appositamente per considerare tutti i restrittivi vincoli inerenti le risorse dei sensori wireless, in primo luogo la scarsa disponibilità energetica. La libreria di componenti di TinyOS include protocolli di rete, servizi distribuiti, driver per sensori, e strumenti di acquisizione dati. Tutti questi componenti possono essere utilizzati così come sono, oppure ridefiniti per un'applicazione custom. Oltre che nel Tmote Sky TinyOS può essere installato su molti altri sensori, commerciali e non commerciali. Sono compatibili ad esempio le piattaforme della Crossbow: Telos, Iris, Mica, Mica2, MicaZ, Mica2dot; la piattaforma iMote della Intel; la piattaforma Eyes [24].

Il linguaggio di programmazione nesC (Nested C o Networked Embedded System C) è un estensione del C, ed è stato progettato per incorporare i concetti strutturali e il modello di esecuzione di TinyOS, che viene implementato proprio con questo linguaggio. Essendo un estensione del C può sfruttarne tutta la potenza, nella realizzazione di codice a basso livello e con alte prestazioni. L'ultima versione di nesC è la 1.1.3 che aggiunge alla precedente i *Network Types*. I tipi di rete permettono allo sviluppatore di scrivere applicazioni per reti di sensori, nelle quali sono presenti piattaforme di vati tipi. In queste reti eterogenee ci si deve scontrare con la diversità di rappresentazione dei dati, nei diversi tipi sensori, e quindi creare del codice

che si occupi delle conversioni. I tipi di rete possono essere utilizzati al posto dei normali tipi di dato, e il compilatore di nesC si preoccupa che questi tipi abbiano la stessa rappresentazione su tutte le piattaforme, generando in automatico il codice di conversione.

Il linguaggio nesC viene definito "statico". I componenti infatti sono assemblati statisticamente l'uno all'altro attraverso le loro interfacce, non possono quindi essere creati o distrutti in esecuzione. Non c'é allocazione dinamica della memoria, e non ci sono i puntatori alle funzioni. Questo aumenta l'efficienza in esecuzione e incoraggia una progettazione più robusta. La composizione statica inoltre elimina quasi tutto il codice di assemblaggio che ostacola l'analisi statica nei linguaggi tipo C++. Inoltre tutte le risorse e il grafico delle chiamate sono completamente conosciuti già nella compilazione.

Il linguaggio nesC è stato progettato ipotizzando che il codice venga generato da un unico compilatore. Questo garantisce una migliore generazione e analisi del codice. Un esempio di questo concetto è il rilevatore di data race usato in fase di compilazione, vedi paragrafo 2.3.3.

La Filosofia di nesC è la stessa di TinyOS, condividono infatti gli stessi modelli e gli stessi obiettivi. I prossimi tre paragrafi riguardano pertanto concetti che sono validi sia in TinyOS che nesC.

2.3.1 Componenti e interfacce

Gli elementi principali di un'applicazione TinyOS/nesC sono i *componenti* e le *interfacce*. Un componente è unità logica dell'applicazione che svolge un determinato compito, mentre un interfaccia definisce una parte del comportamento di un componente in termini di *comandi* che esegue, o *eventi* che notifica. Le interfacce possono essere fornite o usate dai componenti. Vengono fornite da un componente per rappresentare le funzioni che vuole offrire all'utilizzatore, mentre vengono utilizzate da un componente per sfruttare funzioni necessarie per effettuare il proprio lavoro. Un singolo componente può utilizzare o fornire più interfacce, e più istanze della stessa interfaccia. Le funzioni esposte dall'interfaccia sono i comandi, parola chiave command, e devono essere implementate dal componente fornitore dell'interfaccia. Le interfacce di un componente sono i suoi unici punti di accesso. Un'interfaccia contiene oltre che a comandi anche un altro insieme di funzioni, gli eventi, parola chiave event, che devono essere invece implementati dal componente utilizzatore dell'in-

terfaccia. Difatti il compito del componente fornitore dell'interfaccia è solo quello di notificare gli eventi, ma cosa si deve fare quando un evento viene notificato è compito del componente utilizzatore. Per notificare un evento di usa l'istruzione signal. Tipicamente i comandi vengono chiamati verso il basso, ovvero da un componente al livello applicativo verso un componente più vicino all'hardware, mentre gli eventi vengono segnalati verso l'alto. Il duplice utilizzo dell'interfaccia, specificare comandi ed eventi, viene definito come "bidirezionalità dell'interfaccia", e permette ad una singola interfaccia di rappresentare una complessa interazione tra i componenti (ad es. la registrazione di eventi di interesse, seguita da chiamate di callback quando l'evento accade). Ad esempio il componente *CC2420CsmaP* di TinyOS è fornitore tra le altre dell'interfaccia *RadioBackOff*. Utilizzando questo componente è quindi possibile invocare i comandi dell'interfaccia RadioBackOff, i quali permettono di implementare nell'applicazione il meccanismo CSMA. Ancora utilizzando questo componente è possibile nell'applicazione effettuare le registrazioni per ricevere degli eventi forniti da RadioBackOff, e definire il codice di gestione di questi eventi. Questo è un aspetto critico poiché tutti i comandi lunghi in TinyOS (es. la trasmissione di pacchetti) sono non bloccanti, ovvero ritornano subito, e la fine dell'operazione viene segnalata notificando un evento (callback). Questo approccio si chiama *split-phase*, vedi figura 2.9, perché invocazione e completamento di una funzione vengono divise in due fasi distinte dell'esecuzione. Il codice di tipo split-phase è spesso più verboso e complesso di quello sequenziale, però ha dei vantaggi. Le chiamate split-phase infatti riducono l'utilizzazione dello stack, e rendono il sistema più reattivo. Ad esempio utilizzando l'interfaccia *Send* un componente non può chiamare il comando *send()*, senza fornire un'implementazione dell'evento *sendDone*, altrimenti il compilatore segnala errore.

2.3.2 Moduli e configurazioni

I programmi sono costituiti da componenti che sono assemblati ("wired") per formare il programma intero. Il programma può essere rappresentato come un grafo di componenti. Ogni componente è costituito da due elementi, un *modulo* e una *configurazione*, che nel manuale del linguaggio vengono anch'essi chiamati componenti. Il concetto di componente in TinyO/nesC e quindi duplice, intendendo sia un'unità logica di un programma che svolge un determinato compito, sia un modulo o configurazione. Lo scopo di un modulo è quello di definire la logica applicativa di un componente, eseguendo operazioni, implementando interfacce, e utilizzando

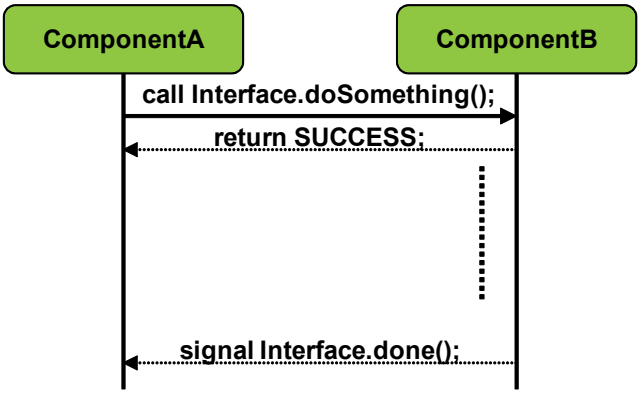


Figura 2.9: Semantica split-phase.

altri componenti, mentre quello della configurazione è di assemblare questo componente con altri componenti che utilizza ("wiring"). Questo approccio viene definito come "separazione tra costruzione e composizione". Un applicazione nesC, o se vogliamo componente nesC, è composta da due file, uno per il modulo e l'altro per la configurazione. Ogni componente, modulo o configurazione, definisce due scope, uno per la *specifica del componente* e uno per l'*implementazione del componente*. Il primo scope, che si crea con il costrutto "module [*nome_mod*] {...}" per i moduli, e "configuration [*nome_conf*] {...}" per le configurazioni, contiene una lista di elementi, ognuno dei quali può essere un'interfaccia che il componente fornisce, o un'istanza di un altro componente che utilizza. Per servirsi di un elemento si utilizza la parola chiave uses, mentre per fornire un elemento si utilizza la parola chiave provides. Il secondo scope, che segue il primo nel file, e che si crea con il costrutto "implementation {...}" sia per il modulo che per la configurazione, contiene nel primo caso la logica applicativa del componente, mentre nel secondo le istruzioni per l'assemblaggio con gli altri componenti.

2.3.3 Modello di esecuzione

Il modello di concorrenza di TinyOS/nesC si basa su due tipi di flusso, task e gestori degli interrupt hardware (a volte indicati solo come eventi). Il secondo tipo viene avviato in modo asincrono dall'hardware. Task e gestori degli interrupt sono run-to-completion [22]. Run-to-completion, che spesso viene confuso con la definizione di non-preenptive, significa che una volta che un flusso è in esecuzione può essere interrotto solo da un flusso a maggiore priorità, e non può riprendere l'esecuzione prima della terminazione del nuovo flusso, anche se questo si autosospende. Poiché

il flusso in TinyOS/nesC è sempre run-to-completion, non c'è l'astrazione equivalente di processo o thread, ed è sufficiente un unico stack (si parla di *stack sharing*). Questo significa che due flussi non possono procedere in un modo interlacciato qualsiasi, come accade nei sistemi multitasking o multithreading. Più nello specifico, non ci sono entità di esecuzione per le quali mantenere lo stato di esecuzione al di fuori di quello che viene memorizzato nei componenti. Quando un sistema TinyOS è quiescente, le variabili del componente rappresentano l'intero stato del sistema [23].

I task inoltre hanno tutti la stessa priorità, quindi tra di loro sono non-preemptive. Non-preemptive significa che un flusso non può cominciare fin tanto che il flusso in esecuzione non cede la CPU di sua spontanea volontà (perché ha terminato, oppure perché si autosospende in attesa di qualche risorsa). I gestori degli interrupt hardware possono invece interrompere task, o altri gestori a priorità inferiore in esecuzione. Poiché i task sono non-preemptive e run-to-completion, il codice di un task viene eseguito in modo sincrono rispetto al codice di un'altro task. Detto in altre parole i task sono atomici l'uno rispetto all'altro, ma non sono atomici rispetto ai gestori degli interrupt hardware [21]. Questa è una buona cosa in quanto i task non possono interferire tra loro, ma significa anche che devono rimanere ragionevolmente brevi [22, 23]. Uno scheduler per nesC può eseguire i task in qualsiasi ordine, ma da rispettare la semantica run-to-completion. In TinyOS lo scheduler standard segue la politica FIFO. Riassumendo quanto detto finora, il modello prevede due code, una di task e una di gestori degli interrupt hardware, ognuna delle quali viene servita in qualche ordine, ma quella dei gestori degli interrupt è più prioritaria di quella dei task.

Più formalmente il codice di una'applicazione nesC può essere di due tipi: codice sincrono e codice asincrono. Il **codice sincrono** viene eseguito in un flusso di tipo task ed raggiungibile solamente da un task. Questo codice quando invocato non può quindi interrompere il flusso in esecuzione, ma verrà eseguito quando deciderà lo scheduler (in TinyOS quando sono stati completati tutti i task prima di lui e non ci sono gestori degli interrupt da mandare in esecuzione). Il codice sincrono può essere il corpo di un comando sincrono `command`, o di un evento sincrono `event`, precedentemente descritti, o del costrutto *task* che si dichiara con lo specificatore `task`. Si deve fare attenzione a non confondere quest'ultimo, con il concetto più generale di task come flusso di esecuzione, che abbiamo utilizzato finora. Il corpo del costrutto `task` viene definito all'interno di un modulo, e può essere utilizzato solo dal modulo stesso invocando il `task` attraverso l'istruzione `post`. L'invocazione

di un command, task, o la notifica di un event ritorna subito e crea un flusso di tipo task. La funzione associata a questi tre elementi non viene pertanto eseguita subito, ma il nuovo task generato viene inserito nella coda dei task. Ad esempio utilizzando l'interfaccia *Send*, e invocando in un task il comando sincrono *send()* che serve per inviare un pacchetto, la trasmissione non avverrà subito ma, supponendo che non ci siano altri task e gestori degli eventi in coda, quando il task in esecuzione ha terminato. Anche l'evento *receive* dell'interfaccia *Receive*, che viene notificato quando si riceve un pacchetto, è contrariamente a quanto si può pensare di tipo sincrono. Quando l'hardware riceve un pacchetto infatti, il gestore degli interrupt non notifica l'evento, ma inserisce nella coda dei task un nuovo elemento. Quando il nuovo task generato verrà eseguito, sarà processato il codice di gestione dell'evento *receive*, definito dall'utilizzatore dell'interfaccia *Receive*. Il costrutto task serve nel caso in cui un componente deve eseguire una qualche funzione interna di lunga durata, che non deve per forza essere eseguita nell'istante in cui viene invocata, ma che al contrario è meglio ritardare, in quanto se eseguita subito può creare dei problemi. Si deve tener presente che non è possibile invocare un task se c'è già una sua istanza in coda o in esecuzione. Questo è vero in generale tranne nel caso in cui un task invoca se stesso. C'é un altro costrutto nesC, la *funzione*, il cui codice è sincrono. Una funzione, come un task, viene definita all'interno di un modulo e può essere utilizzata solo dal modulo stesso per effettuare operazioni interne. La differenza sta nel fatto che quando una funzione viene invocata, vengono subito eseguite le sue istruzioni senza creare un nuovo task. Le istruzioni di una funzione vengono quindi eseguite nel task che l'ha invocata. La funzione pertanto serve ad un modulo per eseguire una routine interna di breve durata, mentre un task serve ad un modulo per eseguire una routine interna di lunga durata.

Il **codice asincrono** viene eseguito in un flusso di tipo gestore di interrupt, ed è raggiungibile da almeno un gestore. Questo codice quindi, quando invocato, interrompe il flusso in esecuzione, se questo è un task o un altro gestore con priorità inferiore. Il codice asincrono è corpo di un comando asincrono async command o di un evento asincrono async event. In ogni caso c'é sempre almeno un interrupt che si verifica. Ad esempio invocando il comando asincrono *led0On()* dell'interfaccia *Leds*, viene generata un'interruzione che accende sul Tmote Sky il led rosso. Se il comando viene invocato in un task il led si accende immediatamente dopo l'invocazione, in quanto il gestore dell'interrupt dell'attivazione del led interrompe il task in esecuzione. Qualsiasi codice che viene eseguito come diretta conseguenza di un

harware interrupt deve essere dichiarato async.

Poiché questo è un modello di esecuzione concorrente, i programmi nesC sono suscettibili di race condition [1], in particolare data race [2] nei programmi che condividono lo stato, ovvero variabili globali e di modulo. Sebbene il non-preemption dei task elimina i data race tra di essi, ci sono ancora potenziali race tra il codice sincrono e quello asincrono, come anche tra codice asincrono e altro codice asincrono. In generale, ogni aggiornamento ad uno stato condiviso che è raggiungibile da codice asincrono è un potenziale data race. Le invarianti di base da rispettare in nesC per non avere data race sono:

> **Invariante Race-Free 1**: Se una variabile x viene acceduta in scrittura in codice asincrono, allora tutti gli altri accessi a x devono avvenire in blocchi protetti dallo specificatore atomic.
>
> **Invariante Race-Free 2**: Se una variabile x viene acceduta in lettura in codice asincrono, allora tutti gli altri accessi in scrittura a x devono avvenire in blocchi protetti dallo specificatore atomic.

L'istruzione atomic permette di creare un blocco di istruzioni che non può essere interrotto da nessun altro flusso.

Il compilatore di nesC segnala i potenziali data race al programmatore. E' possibile introdurre una race condition che il compilatore non riesce a rilevare, ma questa deve coinvolgere molte istruzioni atomiche o task, e usare la memoria in variabili intermedie. Il compilatore nesC può riportare data race che in pratica non possono verificarsi (falsi positivi), ad esempio quando gli accessi alle variabili condivise sono protetti da guardie. Per evitare messaggi ridondanti in questo caso, il programmatore può annotare una variabile con lo specificatore norace per eliminare tutti gli avvertimenti relativi a data race per quella variabile. Lo specificatore norace deve essere comunque utilizzato con cautela. Il compilatore inoltre riporta un errore di compilazione per qualsiasi comando sincrono che viene invocato, o evento sincrono che viene notificato, all'interno di codice asincrono. Questo perché qualsiasi codice eseguito a partire da codice asincrono è anch'esso asincrono, e quindi "non sicuro" se non sono state prese delle adeguate precauzioni. Il compilatore evita in questo modo che venga eseguito del codice, che non è stato scritto per essere eseguito in

[1] quando il risultato dell'esecuzione interallacciata delle istruzioni di due o più processi concorrenti, che condividono dati, dipende dall'ordine con cui vengono eseguite le singole istruzioni

[2] accesso di due task concorrenti ad una variabile condivisa, di cui almeno uno in scrittura e senza meccanismi di protezione

modo sicuro in codice asincrono. L'unico modo per chiamare una funzione sincrona all'interno di codice asincrono, è di invocare un `task` attraverso l'istruzione `post`.

Capitolo 3

Le componenti e l'inizializzazione del sistema WirMoS

3.1 Architettura di sistema

La componente principale dell'architettura di sistema è la wireless sensor network, che permette di raccogliere i dati provenienti dall'ambiente da tenere sotto controllo, e di farli pervenire all'utente. Nel primo capitolo sono stati descritti i diversi tipi di topologia per una WSN, e per questo progetto si è deciso di implementare una topologia ad albero per due motivi. Il primo motivo riguarda la specifica iniziale relativa alla dimensione della rete. Si vuole infatti realizzare un sistema di controllo per una rete di sensori di dimensioni arbitrariamente grandi, e per ottenere questo risultato la topologia a stella non può essere utilizzata. D'altro canto il sistema è stato concepito per essere utilizzato in contesti, nei quali la complessità della topologia mesh risulta eccessiva. Ques't ultima infatti è nata per soddisfare esigenze di alta affidabilità e robustezza, tipiche di alcuni ambienti come quello industriale. Il secondo motivo è che il protocollo FPS utilizzato per il risparmio energetico, assume che i vari sensori siano, a livello logico, disposti ad albero

I nodi dopo una fase iniziale di setup della rete, che permette di creare la topologia ad albero, trasmettono periodicamente un campione di dati (misure di temperatura, umidità, luminosità, tensione batterie etc.) al nodo genitore. I nodi interni trasmettono al proprio genitore sia il loro campione, sia quelli ricevuti dai figli. Dopo alcuni passaggi tutti i dati arrivano al sensore radice dell'albero. La radice, che viene detta *Base Station*, è collegata tramite connessione USB ad una workstation, sulla quale è installato un software (WirMoS-Dashboard) che ha il compito di memorizzare i dati

su memoria di massa, di interagire con gli utenti, e di controllare degli attuatori per forzare le condizioni ambientali. L'unico nodo dell'albero che è alimentato dalla rete elettrica, attraverso la connessione USB della workstation, è la Base Station. Tutti gli altri nodi vengono alimentati da delle batterie stilo.

Il software WirMoS-Dashboard controlla gli attuatori trasmettendo i comandi ad un mote connesso ad un altra porta USB (*bridge*). Questo mote inoltra via radio i comandi ad altri mote detti (*activator*). Gli activator hanno il compito di pilotare gli attuatori attraverso delle connessioni cablate. In alternativa gli activator possono essere connessi direttamente alla workstation via USB.

Se nell'ambiente monitorato è presente una connessione ad Internet cablata o wireless, è possibile connettersi al software WirMoS-Dashboard utilizzando da remoto semplicemente un browser, per eseguire una versione applet dell'applicazione che permette di controllare la situazione degli ambienti monitorati.

Oltre all'applicazione Java WirMoS-Dashboard, per questo progetto sono state sviluppate altre sei applicazioni. Quelle presenti nell'elenco che segue sono scritte in linguaggio nesC vanno installate sui Tmote Sky.

- WirmosBStationP.nc
- WirmosNodeP.nc
- WirmosActivatorP.nc
- WirmosBridgeP.nc
- WirmosReadNodeP.nc

Le prime quattro applicazioni sono in esecuzione sui mote durante il normale esercizio del sistema, mentre WirmosReadNodeP.nc viene utilizzata in abbinamento a WirMoS-ReadNode per leggere offline dei dati di logging, che ogni mote può memorizzare nella propria memoria FLASH durante il funzionamento. WirMoS-ReadNode è un'applicazione Java, installata sulla workstation, che riceve i dati estratti da WirmosReadNodeP.nc, e li memorizza su dei file di testo che l'utente può analizzare. L'applicazione WirmosBStationP.nc deve essere installata nel mote Base Station, e conferisce ad esso le funzionalità di coordinatore della rete e di interfaccia con la workstation, oltre a quelle di normale sensore. WirmosNodeP.nc è l'applicazione che va installata su tutti gli altri nodi della WSN, e serve per l'inizializzazione della rete, per eseguire il protocollo FPS, e per inviare i campioni ambientali. WirmosActivatorP.nc è l'applicazione che va installata sul nodo attivatore, e consente di

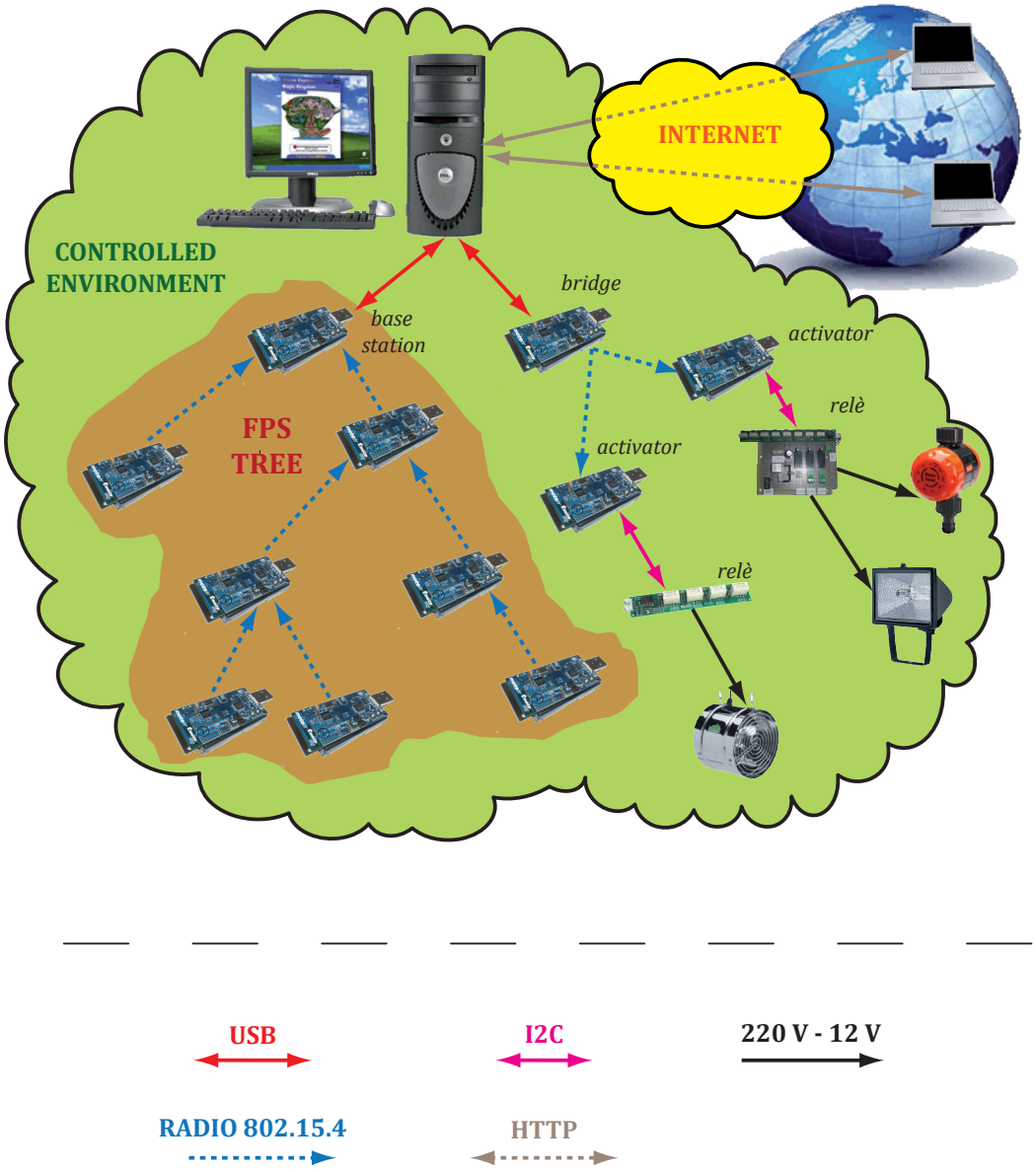


Figura 3.1: Architettura del sistema realizzato.

ricevere i comandi impartiti WirMoS-Dashboard, relativi al controllo delle variabili monitorate, e di azionare gli opportuni attuatori. Il mote sul quale è stata installata quest'applicazione può essere collegato direttamente alla workstation attraverso una porta USB, oppure ricevere i comandi via radio da un altro mote collegato alla workstation, sul quale deve essere installata l'applicazione WirmosBridgeP.nc.

3.2 Inizializzazione del sistema

Il lavoro svolto dall'applicazione WirmosNodeP.nc può essere suddiviso in due macro fasi, la fase di *inizializzazione della rete* che viene descritta in questo capitolo, e la fase di *regime* che viene descritta nel prossimo.

La fase di inizializzazione può essere scomposta in altre cinque sottofasi:

- *Discovering* - Permette ai nodi di venire conoscenza di essere all'interno di una rete e di cominciare la fase di inizializzazione.
- *Pinging* - Valuta la qualità dei link tra i nodi in termini di probabilità di pacchetti persi.
- *Linking* - Effettua la creazione di link affidabili.
- *Routing* - Creazione dell'albero che costituisce la topologia logica della WSN.
- *Logging* - Memorizzazione di informazioni importanti su memoria FLASH.

Queste sei fasi, le cinque di inizializzazione più quella di regime, vengono gestite dall'applicazione attraverso l'utilizzo di un componente timer chiamato *GlobalTimer*. Ogni volta che il GlobalTimer scatta viene avviata una nuova fase e fatto partire il tempo di attesa per l'inizio della successiva. L'invio dei messaggi delle sotto fasi della fase di inizializzazione viene gestito dal timer *SendTimer*, mentre la fase di regime è a carico del timer *FPSTimer*. Prima dell'avvio del GlobalTimer ogni mote durante la fase di bootstrap, ovvero durante l'esecuzione dell'evento *Boot.booted*, esegue delle operazioni di inizializzazione del nodo. Queste operazioni sono:

- Selezione del canale radio per le informazioni di controllo e accensione della radio.
- Inizializzazione del seme dell'algoritmo di generazione di numeri casuali.
- Inizializzazione delle strutture dati.
- Formattazione della flash in due volumi, e loro apertura.

La Figura 3.2 mostra la successione delle sottofasi dell'inizializzazione della rete, si per l'applicazione WirmosBStationP.nc installata nella Base Station, sia per l'applicazione WirmosNodeP.nc installata in tutti gli altri nodi della WSN. Come si vede dalla figura la Base Station non esegue tutte le sottofasi precedentemente illustrate.

Figura 3.2: Sottofasi dell'inizializzazione della rete.

3.2.1 Fase di Discovering

La Base Station incomincia la fase di Discovering quando riceve dall'applicazione WirMoS-Dashboard un messaggio *StartNet*. Per gli altri nodi la fase incomincia non appena viene ricevuto via radio un primo messaggio Discovering. Questo messaggio non ha contenuto e serve solo ad indicare al nodo che si trova all'interno di una WSN, e di cominciare la fase di inizializzazione avviando la fase di Discovering. Il gestore del SendTimer di un nodo, una volta avviata la fase di Discovering, invia in broadcast un numero di messaggi Discovering pari a $N_D = 5$ (Number of Discovering messages). Lo scopo di questi messaggi è quello di raggiungere altri nodi nelle vicinanze, che non sono ancora stati avviati all'inizializzazione. Si ha quindi una propagazione dei messaggi Discovering dalla Base Station, fino a raggiungere i nodi più lontani che sono all'interno della copertura della WSN, *flooding* della rete. La comunicazione in questa prima fase non può che avvenire in broadcast, in quanto ogni nodo non ha ancora cognizione di quali siano i suoi vicini. Il fatto di inviare più di un messaggio di Discovering per nodo, serve per ottenere con un'alta probabilità, che tutti i nodi presenti nel raggio di copertura del nodo vengano coinvolti, anche se dei messaggi vengono persi. Quando il numero dei nodi della rete è elevato, possono verificarsi diverse collisioni di messaggi. Questo problema viene in parte risolto dal meccanismo CSMA presente nel Tmote Sky e sfruttato dal sistema operativo TinyOS, ma per diminuire ulteriormente la probabilità di collisioni, il SendTimer invia i messaggi in istanti casuali. Tra un messaggio e il successivo intercorre infatti un tempo compreso nell'intervallo $[0, DM_{MW}]$ con $DM_{MW} = 150$ ms (Discovering Message Max Wait). Per dare una stima del tempo massimo necessario al completamento di questa fase trascuriamo il tempo di trasmissione, e calcoliamo innanzitutto il tempo massimo impiegato da un nodo a inviare i suoi N_D messaggi. Questo tempo è pari a $T_{DN} = N_D \cdot DM_{MW} = 750$ ms. Si considerano inoltre trascurabili i tempi di ricezione, e il tempo per l'esecuzione delle istruzioni dell'applicazione installata sul nodo. L'istante in cui si conclude questa fase corrisponde allora, all'istante in cui l'ultimo nodo che inizia la fase si Discovering, invia il suo ultimo messaggio Discovering. Consideriamo per fare una media che il primo messaggio di Discovering di ogni nodo venga perso, e $N_{HOP} = 4$ (numero massimo di hop). In questo caso l'istante di conclusione della fase corrisponde a $T_{DISC} = 1950$ ms, calcolato utilizzando l'Equazione (3.1).

$$T_{DISC} = 2N_{HOP} \cdot DM_{MW} + T_{DN} \tag{3.1}$$

3.2.2 Fase di Pinging

Quando un nodo riceve il primo messaggio Discovering avvia nuovamente il timer GlobalTimer impostando un tempo di attesa pari a $P_{PW} = 10$ secondi (Pinging Phase Wait). Allo scattare del timer inizia la fase di Pinging, nella quale ogni nodo invia $N_P = 40$ (Number of Ping messages) messaggi Ping in broadcast. Lo scopo di questa fase è quella di testare la qualità dei link che un nodo ha con i propri vicini, utilizzando come metrica di qualità del link, il numero di messaggi che il nodo riesce ad inviare al suo vicino. Un messaggio Ping non ha contenuto, ciò che interessa è solo l'identificativo del mittente. L'identificativo di un nodo viene assegnato nella fase di installazione del codice, utilizzando un apposita istruzione di TinyOS. Nell'applicazione installata è possibile recuperare questo valore attraverso la costante TOS_NODE_ID. Quando ad un nodo arriva per la prima volta un messaggio Ping di un altro nodo, viene creato un record di una struttura dati chiamata *PingTable*. Ogni record di questa struttura contiene l'identificativo del nodo di cui si è ricevuto almeno un Ping, il numero di Ping ricevuti, e una media del valore RSSI espressa in dBm, calcolata utilizzando il metodo *getRssi(message_t *p_msg)* dell'interfaccia *CC2420Packet*, dove *p_msg* è un puntatore al messaggio ricevuto. La media dell'RSSI viene utilizzata nella fase di Routing. Come esempio di Ping Table si veda la tabella 3.1.

Anche in questa fase per diminuire la probabilità di collisione di messaggi inviati dai nodi, il SendTimer invia i messaggi Ping in istanti casuali. Tra un messaggio e il successivo intercorre infatti un tempo compreso nell'intervallo $[0, PM_{MW}]$ con $PM_{MW} = 150$ ms (Ping Message Max Wait). Il tempo massimo necessario al completamento di questa fase è $T_{ping} = N_P \cdot PM_{MW}$, e con i valori assegnati è pari a $T_{ping} = 6$ secondi.

Mote ID	**Num Ping**	**Mean RSSI**
5	2	33
53	20	25
26	16	5
...	...	...

Tabella 3.1: Esempio di Ping Table.

3.2.3 Fase di Linking

Questa fase comincia dopo un tempo pari a L_{PW} (Linking Phase Wait) dall'inizio della fase di Pinging. Il valore di L_{PW} deve essere scelto in base alla topologia della rete, si veda Figura 3.3. Il suo scopo è quello di creare dei link affidabili tra i nodi, sfruttando le informazioni acquisite nella fase precedente. Per raggiungere questo risultato ogni nodo invia i record della propria PingTable, ai nodi con TOS_NODE_ID contenuto nella prima colonna. In questo modo ogni nodo riceverà l'informazione di chi sono i suoi vicini, e per ognuno di essi quanti ping hanno ricevuto da lui, e con che media RSSI. Il contenuto di un messaggio di linking è quindi un record di PingTable. Per fare in modo di creare dei link affidabili, un nodo non invia tutti i record della propria PingTable, ma solo quelli con un numero sufficiente di ping e con un valore medio di RSSI superiore ad una certa soglia. A questo scopo si utilizza la variabile $MIN_P = 32$ (ovvero l'80%), e la variabile $MIN_{RSSI} = 5$, e si blocca l'invio di un record se il numero di Ping associato ad esso è minore MIN_P, o la media RSSI è minore di MIN_{RSSI}. La Base Station non deve rilevare i suoi vicini affidabili, quindi se un record della Ping Table ha l'identificativo della Base Station nella prima colonna non viene inviato. Poiché è importante che i messaggi contenenti i record non vengano persi, in quanto significherebbe perdere dei link affidabili, viene implementato un meccanismo di ARQ (Automatic Retransmission reQuest) di tipo Stop and Wait. Un nodo dopo aver inviato un record della PingTable aspetta di ricevere un messaggio di risposta di tipo Acknowledge. Se questo messaggio non arriva entro un tempo scelto casualmente nell'intervallo $[LM_{BW}, LM_{BW} + LM_{MW}]$ con $LM_{BW} = 300$ ms (Linking Message Base Wait) e $LM_{MW} = 100$ ms (Linking Message Max Wait), il record viene ritrasmesso, e riparte nuovamente il timer sempre con un tempo di attesa compreso in quell'intervallo. Il tempo di attesa viene scelto casualmente per diminuire la probabilità di possibili collisioni. Se il nodo riceve il messaggio di Acknowledge prima che scatti il timer, si passa alla trasmissione del record successivo nella PingTable. La ritrasmissione può avvenire per un massimo di $N_{LR} = 3$ volte (Number of Linking Retrasmission). Se dopo N_{LR} ritrasmissioni non è stata ricevuto ancora il messaggio di Acknowledge si passa alla trasmissione del record successivo nella PingTable.
Quando un nodo riceve un messaggio di Linking, contenente un record della PingTable, controlla innanzitutto che questo messaggio non sia già stato ricevuto. Difatti a causa del meccanismo di ARQ i messaggi di Linking possono arrivare duplicati, nel caso in cui il messaggio di Akcnowledge relativo sia stato inviato ma non rice-

vuto. Se il messaggio è già stato ricevuto, quello attuale viene scartato. Se invece è la prima volta che si riceve un messaggio di Linking di un particolare nodo, si controlla sulla PingTable nella riga relativa al mittente del messaggio, il numero di Ping ricevuti dal nodo mittente nella fase di Pinging, e la media RSSI. Il messaggio di Linking viene scartato se il numero di Ping è inferiore a $MIN_P = 32$ oppure se l'RSSI medio è inferiore di $MIN_{RSSI} = 5$. In questo modo si accettano come vicini solo i nodi con i quale esiste un link affidabile da questi verso il nodo in questione. Infatti è importante che un link sia affidabile sia verso il genitore, per l'invio dei campioni di rilevazione, sia verso il figlio per l'invio dei messaggi di sincronizzazione, comandi o interrogazioni. Se il messaggio non viene scartato viene creato un nuovo record di una struttura dati chiamata *AdjTable* (tabella dei vicini). Un record della AdjTable contiene il TOS_NODE_ID del vicino, il numero di hop per arrivare alla Base Station attraverso di lui, e la media di RSSI calcolata nella fase di Pinging e relativa ai messaggi inviati a quel nodo. Il numero di hop viene calcolato nella fase di Routing.
Anche la Base Station partecipa alla fase di Linking trasmettendo i record della sua PingTable. Non avendo però inviato Ping nella fase precedente, non riceverà nessun record, non invierà nessun messaggio di Acknowledge e non avrà la AdjTable.
La fase di Linking si conclude quando il nodo con il maggior numero di record nella PingTable ha inviato il suo ultimo record. Il tempo massimo necessario per questa fase se non ci sono ritrasmissioni è quindi $T_{link} = N_N \cdot (LM_{BW} + LM_{MW})$, dove N_N (Number of Neighbours) è il numero massimo di vicini, quindi il numero massimo di record nella PingTable. La Figura 3.3 mostra la durata in secondi della fase di Linking in funzione di N_N, con $LM_{BW} = 300$ ms e $LM_{MW} = 100$ ms.

3.2.4 Fase di Routing

Dopo fase di Linking ogni nodo (tranne la Base Station) ha una AdjTable contenente tutti i suoi vicini collegati da link affidabili. A questo punto un nodo deve scegliere a quale nodo inviare i propri dati, ovvero scegliere chi sarà il genitore nell'albero. La fase di Routing, comincia dopo un tempo di attesa dall'inizio della fase di Linking, impostato nella variabile R_{PW} (Routing Phase Wait). Lo scopo della fase è la determinazione dell'albero che costituisce la topologia logica della rete. Il valore di R_{PW} deve essere maggiore della durata della fase di Linging, che come abbiamo visto dipende dal numero massimo di vicini, e quindi la variabile va impostata caso per caso. Per la scelta del genitore si è deciso di individuare per ogni nodo il

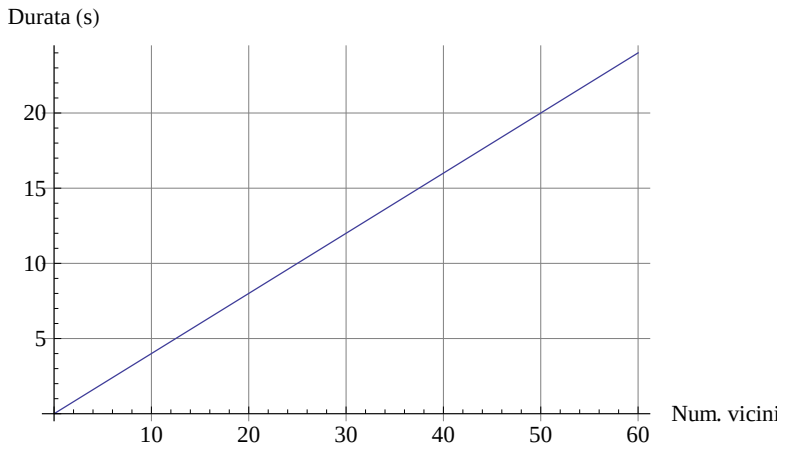


Figura 3.3: Tempo per il completamento della fase di Linking al variare del numero massimo di vicini.

percorso più breve (quindi quello con il minor numero di hop) che si conclude nella Base Station. Questo significa che l'albero in questione sarà un *Minimun Spanning Tree*, considerando uno il peso di ogni link. La funzione di costo in questo modo è il numero di hop. In letteratura si possono trovare vari algoritmi di routing per questo scopo, e nel nostro caso la scelta è stata quella di utilizzare la strategia a *Vettore di Distanza*. E' stato scelto questo algoritmo per la sua semplicità di implementazione, soprattutto in questa applicazione dove l'unica destinazione per ogni nodo è la Base Station. Per questo motivo il vettore di distanza è in realtà uno scalare.

Ogni nodo ha una struttura chiamata *BaseStationPath* che contiene tre campi: *nextHop* che fornisce il TOS_NODE_ID del genitore del nodo sull'albero, *numHop* che contiene il numero hop necessario a raggiungere la Base Station, e *meanRSSI* che contiene la media degli RSSI ottenuta nella fase di Pinging con il nodo genitore. Inizialmente il valore del campo *numHop* è impostato a 255 che corrisponde a ∞. All'inizio della fase di Routing, ogni nodo controlla nella propria AdjTable se possiede un record con il TOS_NODE_ID della Base Station. Se così è, modifica i campi della BaseStationPath, impostando come *nextHop* il TOS_NODE_ID della Base Station e *numHop*=1, e invia quindi un messaggio Routing in broadcast. Questo messaggio contiene il numero di hop che distano dal mittente alla Base Station.

Quando un nodo riceve un messaggio Routing, verifica innanzitutto se il mittente è contenuto nella AdjTable, dato che questa contiene solo i vicini "affidabili". Se il mittente è un vicino affidabile il nodo controlla il valore del campo *numHop* della BaseStationPath, e se questo valore è maggiore di uno, sommato al valore contenuto nel messaggio arrivato, significa che utilizzando come genitore il mittente

del messaggio, il percorso verso la Base Station è più corto dell'attuale. In questo caso il nodo aggiorna i campi della BaseStationPath impostando *nextHop* con il TOS_NODE_ID del mittente, *numHop* con il valore contenuto del messaggio più uno, e *meanRSSI* con il valore del campo meanRSSI della AdjTable, nella riga con il TOS_NODE_ID del mittente. Dato che è cambiato il percorso e il numero di hop necessari per arrivare alla Base Station, il nodo trasmette un messaggio Routing in broadcast, per informare i suoi vicini del cambiamento. Il messaggio viene inviato dopo un tempo di attesa compreso nell'intervallo $[0, RM_{MW}]$, con $RM_{MW} = 300$ ms (Routing Message Max Wait), sempre per diminuire la probabilità di collisione. Nel caso invece in cui il valore del campo *numHop* della BaseStationPath sia uguale al valore contenuto nel messaggio arrivato più uno, allora il percorso attuale ha stessa lunghezza del percorso che si otterrebbe utilizzando come genitore il mittente del messaggio. In questo caso per decidere se cambiare genitore nell'albero, il nodo controlla il campo *meanRSSI* della BaseSationPath. Se quest'ultimo è minore di quello nella riga della AdjTable con il TOS_NODE_ID del mittente, allora sceglie quest'ultimo come genitore, aggiorna la BaseStationPath e invia un messaggio Routing come nel caso precedente. Questo criterio comporta che, nella scelta del genitore tra vicini distanti lo stesso numero di hop dalla Base Station, un nodo scelga quello che riesce a ricevere il suo segnale con potenza maggiore. L'idea che sta alla base del criterio è che, essendo la potenza di trasmissione fissa, il valore di RSSI fornisce una indicazione della qualità del link. Ad un valore di RSSI maggiore infatti corrisponde un nodo che si trova fisicamente più vicino a quello che trasmette, o comunque in posizione più favorevole per la ricezione dei suoi messaggi. Nel caso infine in cui il valore del campo *numHop* della BaseStationPath sia minore del valore contenuto del messaggio arrivato più uno, allora il nodo non fa niente, in quanto è il percorso attuale utilizzato dal nodo ad essere più corto.

Qualsiasi sia il caso dei tre appena illustrati, quando viene ricevuto un messaggio Routing di un vicino affidabile, viene impostato il campo *numHop* della AdjTable relativo a quel vicino. Questo valore viene utilizzato durante il funzionamento a regime del sistema quando la topologia della rete cambia, vedi Paragrafo 4.8.

Concluso il processo di routing, ogni nodo può comunicare con la base station lungo un cammino minimo.

3.2.5 Fase di Logging

Questa fase inizia dopo un tempo di attesa dall'inizio della fase di Routing, impostato nella variabile LO_{PW} (Logging Phase Wait). Il suo scopo è quello di memorizzare nella FLASH, sia durante la fase di inizializzazione della rete, che in quella di normale esercizio, informazioni utili per rilevare eventuali problematiche, conoscere le prestazioni, il comportamento degli algoritmi, etc. . Queste informazioni possono essere poi recuperate off-line, installando sul mote l'applicazione WirmosReadNodeP.nc, che legge la FLASH, e trasmette i dati alla workstation tramite l'interfaccia USB, all'applicazione WirMoS-ReadNode. WirMoS-ReadNode riceve i dati e li memorizza su file. Si è deciso di utilizzare questa strategia per memorizzare i dati di logging, piuttosto che inviare messaggi via radio, in quanto perché la fase di Logging risulti utile, si deve inviare alla workstation un numero abbastanza grande di informazioni. Se queste vengono trasmesse via radio è necessario occupare spesso il canale, aumentando significativamente la probabilità di collisione, e quindi la probabilità di perdita di messaggi. Questa fase ha scopo di ricerca e risoluzione delle problematiche, e non è finalizzata al normale esercizio del sistema. Per questo motivo è sufficiente recuperare i dati memorizzati quando il sistema non è in esecuzione.

Per scrivere sulla FLASH si è utilizzato l'interfaccia *LogWrite* contenuta in "/tos/interfaces". Il comando *append(void* buf, storage_ len_ t len)* scrive sulla FLASH i dati contenuti nella RAM, che partono dall'indirizzo memorizzato nel puntatore *buf*, e che finiscono dopo un numero di byte pari al contenuto della variabile *len*. Una volta che la scrittura è stata eseguita viene lanciato l'evento *appendDone()*, che segnala la fine dell'operazione di scrittura, e riporta eventuali errori. La scrittura tramite quest'interfaccia è sequenziale, il nuovo elemento viene scritto sulla FLASH in coda al successivo. E' comunque possibile conoscere il valore dell'indirizzo di memoria dove verrà scritto il nuovo elemento, utilizzando il comando *currentOffset()*. Il componente che è stato utilizzato per implementare quest'interfaccia è *LogStorageC*, contenuto in "/tos/chips/stm25p". E' possibile dividere la FLASH in diversi volumi, e per far ciò si deve creare un istanza del componente LogStorageC per ognuno, specificando la grandezza del volume, e indicando se deve essere lineare o circolare. Si deve tener presente che il limite massimo di byte, che si possono scrivere in una chiamata del comando *append()*, per il componente LogStorageC, è di 254 byte.

Anche i tempi di scrittura e lettura sono parametri importanti da considerare, dato che quelli della FLASH generalmente sono molto più lunghi di quelli della RAM. Nelle applicazioni WirmosBStationP.nc e WirmosNodeP.nc ciò che ci inte-

ressa è il tempo di scrittura, perché la lettura avviene off-line. Il tempo necessario per scrivere un insieme di dati su una memoria, è dato dalla somma del tempo di accesso alla memoria, che è fisso e dipende dalla particolare memoria utilizzata, e il tempo di scrittura vero e proprio, che dipende dalla quantità di dati scritti. Utilizzando l'interfaccia LogWrite, non è necessario un'operazione specifica di apertura del volume, in quanto viene fatta automaticamente in corrispondenza della prima scrittura, che di conseguenza impiegherà più tempo delle successive. Considerando ciò, sperimentalmente è stato determinato che il tempo che intercorre tra una chiamata del comando append(), e la notifica del corrispondente evento appendDone(), è di 1130 ms nel caso della prima scrittura, e 3 ms per le successive, scrivendo una quantità di dati pari a 46 byte.

Sono stati creati due volumi della FLASH, uno chiamato *Message* e l'altro *Data*. Nel primo vengono memorizzati dei dati ogni volta che viene spedito o ricevuto un messaggio. Per ogni messaggio si tiene traccia, utilizzando una struttura di tipo *LogMessage*, del mittente, del tipo di messaggio (Discovering, Ping, etc.), e del tempo trascorso dall'inizio della fase di inizializzazione, ovvero da quando è stato ricevuto il primo messaggio Discovering. Questi dati vengono memorizzati direttamente sulla FLASH, nel momento della ricezione o dell'invio dei messaggi, e non durante la fase di Logging. Se venissero memorizzati sulla RAM, per poi essere scaricati sulla FLASH durante la fase di Logging, si verificherebbe un sicuro riempimento della RAM, dato che è di dimensione molto ridotta. Inoltre nel volume LogMessage, si vogliono memorizzare anche le informazioni riguardanti i messaggi trasmessi/ricevuti durante la fase di regime del sistema.

Il secondo volume viene utilizzato nella fase di Logging. In questa fase avviene la scrittura di tre struct, rispettivamente di tipo *LogDataTX*, *LogDataRX*, e *VarData*, la scrittura dei record della PingTable e dei record della AdjTable. Nella prima struct sono contenute informazioni relative alla trasmissione dei messaggi nella fase di inizializzazione: il numero totale di messaggi inviati, quello dei messaggi Linking, Acknowledge e Routing. Nella seconda struct sono contenute informazioni relative alle ricezione di messaggi nella fase di inizializzazione: il numero totale di messaggi ricevuti, quello dei messaggi Discovering, Pinging, Linking, Acknowledge e Routing. Nella terza struct sono contenute informazioni di vario tipo: la lunghezza della PingTable e della AdjTable, il TOS_NODE_ID del genitore sull'albero, il numero di hop necessari a raggiungere la base station, e il numero di elementi (istanze della struttura LogMessage) non salvati nel volume Message. Infatti non è detto che per tutti

i messaggi trasmessi/ricevuti, venga salvato un elemento del volume LogMessage. Il motivo è la ridotta velocità di scrittura nella memoria FLASH (3 ms), che in alcuni casi non è sufficiente, rispetto alla velocità di esecuzione degli eventi del mote. Si pensi ad esempio, al caso in cui due nodi inviino un messaggio ad un terzo nodo, in un tempo inferiore a 3 ms . In questa situazione il mote destinatario potrebbe essere in grado di ricevere correttamente i due messaggi, ma non è possibile memorizzarli entrambi. Nelle sperimentazioni fatte questo problema non si è comunque rivelato di un entità tale da compromettere l'utilità della memorizzazione dei dati nel volume Message.

Nel volume Data oltre che le informazioni appena descritte, viene memorizzato anche lo schedule FPS, vedi Paragrafo 4.3.

Nel Paragrafo 2.2 è stato mostrato che il consumo di energia relativo all'utilizzo della memoria FLASH è piuttosto alto, e che quindi per ottenere il risparmio energetico l'accesso alla memoria deve essere limitato il più possibile. Per questo motivo nelle applicazioni installate sui nodi, esiste la possibilità di disattivare completamente la scrittura dai dati di logging sulla FLASH, quando questa risultasse non necessaria. E' possibile attivare o disattivare in modo indipendente la scrittura del volume Data, e del volume Message. Per il volume Message si può scegliere inoltre se memorizzare i messaggi solo nella fase inizializzazione, solo nella fase di regime, o in entrambe.

Capitolo 4

Il sistema WirMoS a regime

Il consumo di energia limita l'utilità delle reti di sensori wireless. Infatti cambiare le batterie dei sensori può essere un'attività molto laboriosa, soprattutto se i sensori sono situati in posti scomodi, e il ricambio deve essere fatto frequentemente. I ricercatori sono concordi nell'affermare che, tra tutte le funzioni, la comunicazione radio domina i consumi nelle reti di sensori wireless. Questo riguarda sia la trasmissione che la ricezione, perché alle distanze di comunicazione tipiche di una rete di sensori, il costo dell'ascolto sul canale radio è simile al costo per la trasmissione dei dati. La frazione di tempo utilizzata da un nodo, per trasmettere o ricevere informazioni via radio, è tuttavia in molte applicazioni molto bassa, rispetto al resto del tempo in cui il nodo esegue altre operazioni, o è idle. E' quindi sicuramente una buona soluzione, per aumentare la vita di un nodo, eliminare tutti quei tempi in cui la radio rimane accesa inutilmente, ovvero quando non deve trasmettere, o restare in ascolto per ricevere messaggi. Utilizzando la topologia a stella, nella quale il centro stella è il sink, ed è quindi l'unico nodo che deve ricevere dati, questa strategia e facilmente applicabile. E' sufficiente infatti, che il centro stella sia alimentato dalla rete elettrica, mantenendo la radio sempre accesa, e imporre che la radio degli altri nodi rimanga accesa solo durante la trasmissione. Se la WSN ha una topologia a mesh o ad albero, come nel caso caso in esame, sono molti i nodi che oltre a trasmettere devono anche ricevere messaggi da inoltrare. In questi casi è necessario usare meccanismi più complessi. Uno di questi è il protocollo FPS, che utilizziamo come strategia base, per ottenere il risparmio energetico per la nostra rete di sensori.

FPS è un protocollo distribuito per la gestione dell'energia, per le reti di sensori wireless con topologia ad albero, che riduce il consumo dell'energia dei sensori, supportando la richiesta variabile di accesso al canale radio dei nodi. FPS ottiene il risparmio energetico, utilizzando la strategia della suddivisione del tempo in slot.

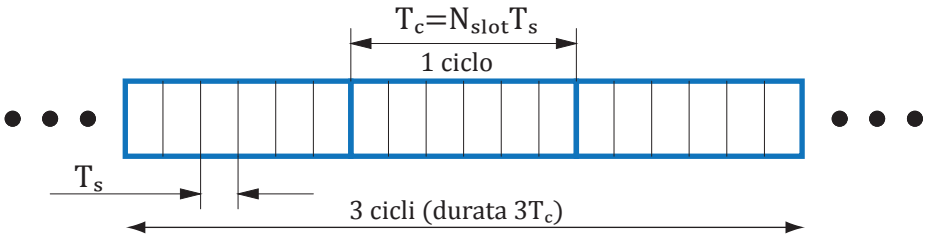


Figura 4.1: Terminologia FPS.

Con questa strategia viene deciso per ogni slot quale nodo deve trasmettere e quale ricevere, e quindi solo questi nodi avranno in quello slot la radio accesa. Gli altri nodi possono mantenere la radio spenta risparmiando così sui consumi. Generalmente, la suddivisione del tempo in slot richiede un controllo centralizzato e una sincronizzazione fine, e utilizza schedule globali e statici. L'algoritmo FPS invece non utilizza un controllo centralizzato, utilizza schedule locali adattivi, e richiede solo una sincronizzazione grossolana. Gli schedule adattivi consentono il cambiamento della topologia della rete o delle richieste dei nodi, senza dover re-inizializzare la rete, come invece accade nel caso di schedule statici.

Il capitolo illustra il protocollo FPS e le estensioni sviluppate per realizzare il protocollo di WirMoS. Le differenze tra i due protocolli sono riassunte nella Tabella 4.1.

4.1 Definizioni nel protocollo FPS

In FPS la WSN è organizzata ad albero e il tempo è suddiviso in slot. Ogni slot s ha una durata T_s. Un ciclo c è una sequenza di N_{slot} consecutivi. Un ciclo c ha quindi durata $T_c = N_{slot}T_s$. Quando finisce un ciclo ne inizia un'altro, e ad ogni ciclo accadono gli stessi eventi, vedi Figura 4.1.

Ogni nodo mantiene uno schedule locale sulle operazioni da compiere nel corso del ciclo. In ogni slot del ciclo un nodo può trovarsi in uno di dei stati stati tra cui i già citati: ricezione, trasmissione o idle. Ad ogni stato corrisponde un tipo di slot. Gli slot di trasmissione e ricezione nello schedule locale, permettono ad un nodo di sapere quando i figli trasmettono, e quando il genitore è in ascolto per ricevere i suoi messaggi. Diamo ora alcune definizioni:

- $N_T(k)$- numero di slot di trasmissione di un nodo all'inizio del $k-esimo$ ciclo.

	FPS	**WirMoS**
Rivelazione vicini affidabili CAP. 3	NO	SI
Creazione topologia ad albero CAP. 3	NO	SI
Gestioni collisioni	CSMA	FDMA (cambio di canale)
Sincronizzazione	grossolana	fine
Aumento tempo di campionamento	aumenta la latenza	latenza invariata (supercicli)
Advertisement CAP. 6	sempre attivi	attivi solo quando varia la topologia
Controllo qualità trasmissioni Variazioni topologia	NO	SI

Tabella 4.1: Differenze tra i protocolli FPS e WirMoS.

- $N_R(k)$ - numero di slot di ricezione di un nodo all'inizio del $k-esimo$ ciclo.
- N_{TS} - numero di slot di trasmissione di un nodo utilizzati per se stesso (domanda iniziale).
- $D(k)$ - domanda (demand) di un nodo all'inizio del $k-esimo$ ciclo.
- $S(k)$ - offerta (supply) di un nodo all'inizio del $k-esimo$ ciclo.

La domanda di un nodo corrisponde al numero di slot di ricezione, che il genitore deve riservare per ricevere tutti i suoi messaggi. Di conseguenza sarà data dalla somma della domanda iniziale (numero di slot di trasmissione usati per se stesso),

più il numero di slot di ricezione del nodo: $D(k) = N_{TS} + N_R(k)$. Infatti il nodo per ogni slot di ricezione dovrà riservare anche uno slot di trasmissione, per inoltrare il messaggio ricevuto. L'offerta di un nodo invece è il numero di messaggi, generati da se stesso o da altri nodi, che riesce a inoltrare, e quindi corrisponde al numero di slot di trasmissione $S(k) = N_T(k)$. Lo schedule viene inizializzato ponendo tutti gli slot di tipo idle, e evolve durante l'esecuzione dell'algoritmo FPS. Se $D(k) > S(k)$ la domanda del nodo non è pienamente soddisfatta, e questo significa che il genitore non è in grado di ricevere tutti i suoi messaggi. Quando invece $D(k) = S(k)$ la domanda del nodo è pienamente soddisfatta, e il genitore è in grado di ricevere tutti i suoi messaggi. A regime (inizio del $k_r - esimo$ ciclo) quando la domanda di ogni nodo è stata pienamente soddisfatta, per tutti vale quindi l'Equazione (4.1):

$$\begin{aligned} D(k_r) &= S(k_r) \\ N_{TS} + N_R(k_r) &= N_T(k_r) \end{aligned} \tag{4.1}$$

Si assume che un nodo possa spegnere il dispositivo radio durante gli slot di tipo idle. Se indichiamo con $b(j) \leq N_{slot}$ il numero di slot (in un ciclo) in cui un nodo j non è idle, possiamo definire il duty cycle del nodo j, $\delta(j)$ come:

$$\delta(j) = \frac{b(j)}{N_{slot}} \tag{4.2}$$

L'Equazione (4.2) mostra che ci sono due modi per diminuire il duty cycle in un nodo: o aumentare N_{slot}, il numero di slot per ciclo, o ridurre $b(j)$, il numero di slot non idle. Ogni nodo può quindi adattare il suo schedule per minimizzare il suo consumo di energia indipendentemente dall'attività degli altri nodi.

4.2 Un Semplice esempio

Come esempio del protocollo, consideriamo la rete illustrata in Figura 4.2. I quadrati rappresentano i nodi, e si considera che ogni nodo abbia domanda iniziale $D(0) = 1$. I pallini neri rappresentano la domanda aggiuntiva creata durante l'esecuzione dell'algoritmo. Le linee che collegano i quadrati sono i link di comunicazione radio tra i nodi. Il nodo 0 è la Base Station.

Iniziando dal basso, i nodi 3 e 4 richiedono entrambi slot di trasmissione al loro genitore, che è il nodo 1. La domanda per questo nodo è otto, quattro per il nodo 3,

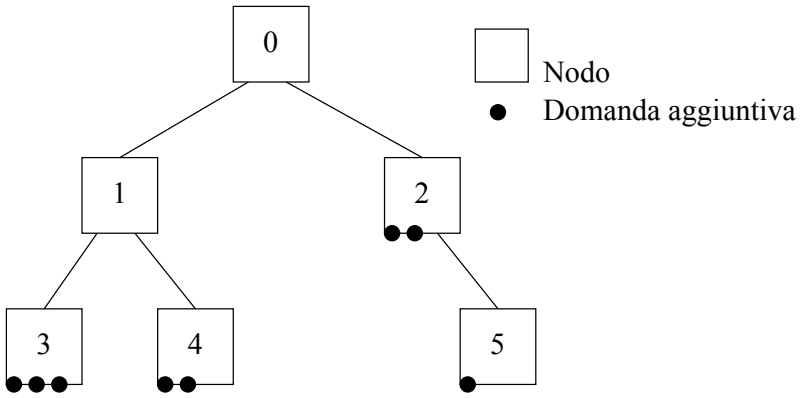


Figura 4.2: Esempio di rete per il protocollo FPS.

più tre per il nodo 4, più uno per se stesso. Il nodo 1 quindi deve richiedere otto slot di ricezione alla base station. Osservando l'altro ramo, vediamo che il nodo 2 deve richiedere cinque slot di ricezione alla base station, tre per se stesso, più due per il nodo cinque. Già da questo semplice esempio appare chiaro che i nodi in prossimità della base station hanno un duty cycle più elevato, come nel caso generale delle reti di sensori con questo tipo di topologia.

Un'altra osservazione importante è che i dati generati da un nodo al livello h, possono essere ritardati di un numero di cicli massimo pari a h, prima di essere ricevuti dalla Base Station. Nel caso peggiore, i dati generati dai nodi 1 e 2 nell'esempio sono ricevuti dopo un 1 ciclo, mentre i dati generati dai nodi 3 e 5 sono ritardati di due cicli. Se è richiesta una particolare latenza, il numero di slot N_{slot} deve avere un limite superiore, il che però incide sul duty cycle complessivo nella rete.

4.3 Esecuzione del protocollo FPS

Ogni nodo mantiene in memoria il proprio schedule, ovvero un array di N_{slot} elementi ciascuno dei quali può assumere uno tra sei valori. Questi valori corrispondono agli stati in cui il nodo può trovarsi in uno slot. Gli stati sono:

1 - *Transmit* (T) - Un nodo in questo slot trasmette un messaggio al nodo genitore.

2 - *Receive* (R) - Un nodo in questo slot riceve un messaggio da uno dei nodi figli.

3 - *Advertisement* (A) - Un nodo in questo slot trasmette in broadcast un mes-

saggio Advertisement, per avvertire i nodi figli che ha slot liberi per poter soddisfare la loro domanda.

4 - *Transmit Pending* (TP) - Un nodo in questo slot trasmette un messaggio Reservation Request al genitore, per soddisfare la sua domanda.

5 - *Receive Pending* (RP) - Un nodo in questo slot riceve un messaggio Reservation Request da un figlio, e eventualmente trasmette un messaggio Reservation Confirm al figlio per confermare la prenotazione di uno slot.

6 - *Idle* (I) - Un nodo in questo slot non deve trasmettere messaggi, e se ha soddisfatto tutta la sua domanda corrente non ha nemmeno la necessità di riceverli. In questo caso quindi può disattivare il dispositivo radio.

Dopo un tempo di attesa di $FPS_{PW} = 10$ s (FPS Phase Wait) dalla fine della fase di Logging, inizia con lo scattare del timer FPSslotTimer, la fase FPS, ovvero di normale esercizio. Da questo momento in poi il timer scatta ogni T_s, a scandire la fine di ogni slot. Un controllo all'inizio di ogni slot stabilisce se un ciclo è terminato e deve iniziare successivo. All'inizio della fase di regime, ci sarà un intervallo di tempo in cui solo alcuni nodi riusciranno trasmettere tutti i loro messaggi, gli altri cercheranno di fare in modo che la propria domanda venga pienamente soddisfatta. Il protocollo FPS è infatti adattivo, ovvero la domanda dei nodi può cambiare in qualsiasi momento, quindi anche questa prima parte della fase FPS è da considerarsi di normale esercizio.

La Base Station si comporta in modo differente rispetto agli altri nodi, in quanto si comporta solo da genitore. La radice infatti non deve trasmettere nessun campione di rilevazione via radio alla workstation, e quindi non ha slot di tipo T. Inoltre non ha ne domanda ne offerta. Il suo comportamento è pertanto sempre uguale, e ha sempre la radio accesa, dato che è alimentata dal PC.

Inizialmente tutti i nodi hanno $D(k) > 0$, la domanda iniziale, e $S(k) = 0$, quindi sono nella condizione $D(k) > S(k)$. In questa condizione mantengono la radio sempre accesa per ricevere messaggi *Advertisement*, che permettono di soddisfare la loro domanda. Quando il protocollo FPS viene avviato solo la Base Station trasmette Advertisement. Ad ogni ciclo la Base Station sceglie due slot a caso tra quelli I, e li definisce uno come A, e l'altro come RP. Quando si trova nello slot A invia un messaggio Avertisement in broadcast, contenente l'indice dello slot che ha definito come RP.

Se un nodo con $D(k) > S(k)$, riceve un messaggio Advertisement, controlla innanzitutto se il mittente corrisponde al proprio genitore nell'albero, e se questo è

il caso, definisce come TP lo slot che ha per indice il valore trovato nel messaggio. Quindi lo slot in questione è definito come TP nel nodo che ha ricevuto l'Advertisement, e come RP nel genitore di questo nodo. Se lo slot nel nodo destinatario non era I, l'Advertisement viene ignorato. Quando un nodo si trova nello slot TP invia un messaggio di tipo *Reservation Request* al genitore. Il genitore che in questo intervallo di tempo è nello slot RP, riceve il messaggio Reservation Request dal nodo. Se il messaggio ricevuto dal genitore è il primo messaggio che riceve in quello slot per il ciclo corrente, definisce lo slot RP come R, aumenta il valore della domanda di uno, e trasmette al mittente un messaggio di tipo *Reservation Confirm*, a conferma del successo della prenotazione del figlio. Il nodo genitore controlla di non aver ricevuto già altre richieste in quel ciclo, perché il suo Advertisement viene trasmesso in broadcast, quindi più di un nodo figlio può effettuare una richiesta nello stesso slot TP. Per questo motivo nel gestore dell'evento Receive.receive si controlla quando si riceve un messaggio di tipo Reservation Request, se questo è il primo del ciclo, e se non lo è il messaggio viene scartato. Ritornando alla prenotazione, il nodo che ha inviato la richiesta riceve nello stesso slot TP il messaggio Reservation Confirm, controlla quindi se proviene dal genitore, e se si, definisce lo slot TP come T e aumenta il valore dell'offerta di uno. Nel ciclo successivo il figlio può inviare al genitore in questo nuovo slot T un campione di rilevazione, che può essere un proprio campione, o quello di un figlio che deve essere inoltrato. Un illustrazione di questo scambio di messaggi è mostrato in Figura 4.3.

Quando un nodo ha $D(k) = S(k)$ ha domanda pienamente soddisfatta, può quindi cercare di soddisfare delle domande non pienamente soddisfatte dei propri figli. Questo significa che si comporta come già descritto per la Base Station, ovvero all'inizio del ciclo, sceglie due slot a caso tra quelli I e li definisce uno come A, e l'altro come RP. Quando poi si trova nello slot A invia un messaggio di tipo Avertisement in broadcast contenente l'indice dello slot che ha definito come RP. A differenza della Base Station però, un nodo qualsiasi se ha domanda pienamente soddisfatta, spegne la radio durante gli slot I, in quanto non ha necessità di ricevere Advertisement, ottenendo così il risparmio energetico. Si deve prestare attenzione però al fatto che il nodo ha comunque il chip radio acceso negli slot R, e che quindi se viene ricevuto un Advertisement in questi slot il messaggio va scartato, per non alterare il corretto funzionamento del protocollo.

Lo slot definito come RP all'inizio di un ciclo, in un nodo con $D(k) = S(k)$, o nella Base Station, può avere indice maggiore di quello dello slot definito come

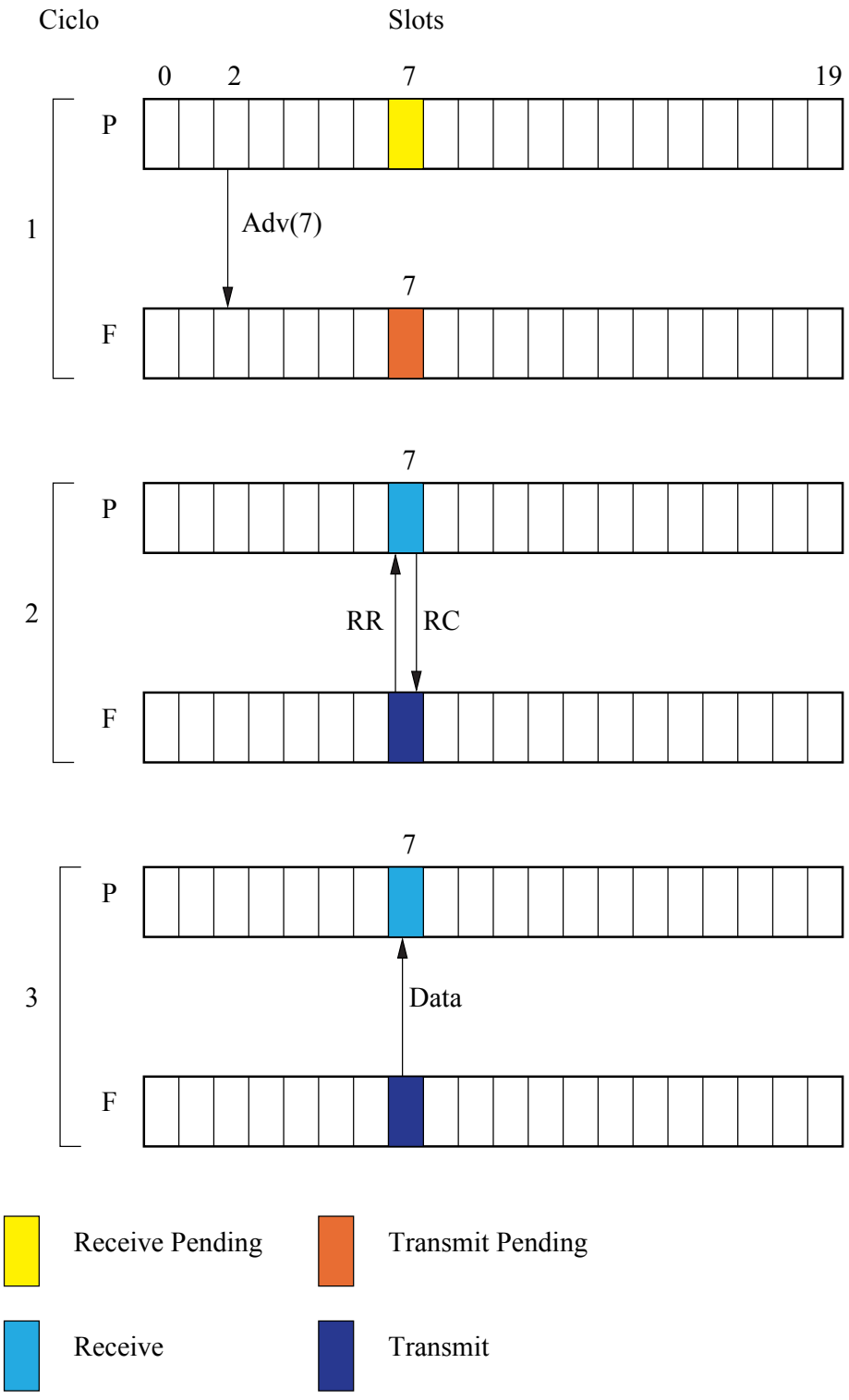


Figura 4.3: Operazione di prenotazione di uno slot tra due nodi.

A, sempre all'inizio del ciclo, come in Figura 4.3. In questo caso se un nodo figlio risponde all'Advertisement di questo nodo, la prenotazione si può concludere nel ciclo stesso. Se invece lo slot RP ha indice minore dello slot A, l'eventuale richiesta di un nodo figlio, viene ricevuta dal genitore nel ciclo successivo, e la prenotazione dura quindi due cicli. Perchè ciò avvenga correttamente si utilizzano due variabili booleane *slotRPreserved* e *firstCycleRP*, che inizialmete valgono FALSE. Quando un nodo definisce uno slot RP pone le due variabili a TRUE. Se nello stesso ciclo trasforma lo slot RP in R, il che significa che la prenotazione è andata a buon fine in quel ciclo, pone nuovamente le variabili a FALSE. Alla fine del ciclo trovando la variabile slotRPreserved a FALSE, un nodo sa che non ci sono prenotazioni in sospeso. Se invece alla fine del ciclo trova le due variabili a TRUE, il nodo sa che c'è una prenotazione cominciata in quel ciclo ma che non si è ancora risolta, e l'unica cosa che fa è porre firstCycleRP a FALSE, a indicare che la prenotazione non si è conclusa in un ciclo. Se alla fine del ciclo successivo la situazione è slotRPreserved=FALSE e firstCycleRP=FALSE, significa che la prenotazione si è conclusa nel secondo ciclo, e non occorre fare niente. Se invece, la situazione alla fine del ciclo successivo è slotRPreserved=TRUE e firstCycleRP=FALSE, la prenotazione non si è conclusa nemmeno nel secondo ciclo, e quindi può essere cancellata ponendo lo slot RP nuovamente a I.

E' utile riuscire ad analizzare gli schedule dei nodi, sia per confermare la correttezza dell'implementazione, sia per studiare il comportamento dell'algoritmo. Si è deciso quindi di memorizzare nella FLASH sul volume Data, assieme agli altri dati visti nel Paragrafo 3.2.5, anche lo schedule. Nella Base Station questo viene fatto nel momento esatto in cui si conclude l'ultima delle prenotazioni, relative alle richieste FPS, dei sensori inizialmente presenti sulla rete. Per far ciò nella Base Station si utilizza la variabile NUM_NODES, che viene inizializzata con il numero iniziale di nodi presenti sulla rete. Se ogni nodo ha domanda iniziale pari a uno, alla fine lo schedule della Base Station ha N_R (numero di slot di ricezione) uguale al numero di nodi presenti nella rete. Inoltre per come funziona il protocollo FPS, la fase delle prenotazioni iniziale si conclude con la definizione dell'ultimo slot R nella Base Station. Considerando queste affermazioni, nel momento in cui la Base Station definisce uno slot come R, dopo che ha già definito come R altri NUM_NODES-1 slot, la fase delle prenotazioni FPS è conclusa. In questo istante quindi la Base Station memorizza lo schedule sulla FLASH, e invia un messaggio di tipo *ChangeMsg* in broadcast nello slot B (questo tipo di slot viene descritto nel Paragrafo 4.5). Quan-

do un nodo riceve questo messaggio esegue delle operazioni tra le quali inoltrare in broadcast il messaggio nel proprio slot B, come la Base Station, e memorizzare nella FLASH lo schedule. Trascorso un intervallo di tempo tutti i nodi avranno ricevuto il messaggio ChangeMsg e memorizzato lo schedule nella FLASH. Se la domanda iniziale dei nodi può essere anche diversa da uno, al posto di NUM_NODES si deve utilizzare la somma delle domandi iniziali.

Nel Paragrafo 3.2.5 si è detto che utilizzando il comando append(), del componente LogStorageC, si può scrivere al massimo 254 byte alla volta. Poiché uno slot dello schedule occupa 1 byte, se lo schedule ha più di 254 slot è necessario memorizzarlo in più pezzi. Un altro limite da considerare viene dal fatto che non è possibile creare messaggi da spedire via radio o seriale (anche USB), maggiori di una determinata grandezza. Dobbiamo tener conto di questo perché gli elementi che vengono memorizzati nella FLASH, vanno poi spediti via USB al PC. Il valore massimo in byte del payolad di un messaggio è dato dalla variabile di TinyOS DTOSH_DATA_LENGHT. Di default questa variabile vale 28 byte, e quindi non è possibile inviare più 28 byte di dati per messaggio. E' possibile modificare il valore di questa variabile in fase di compilazione inserendo nel MakeFile l'istruzione 'PFLAGS += -DTOSH_DATA_LENGHT=X', con X il valore in byte che si vuole assegnare. Il valore massimo di DTOSH_DATA_LENGHT è per il Tmote Sky circa 200 byte. Nella scrittura sulla FLASH si deve quindi dividere lo schedule in parti non più grandi di 200 byte. E' stata scelta una dimensione di 100 byte. Si definisce allora un nuovo tipo di struct, chiamata *SlotStructLog*, che rappresenta un pezzo dello schedule da memorizzare sulla FLASH. A differenza di SlotStruct che contiene N_{slot} slot, SlotStructLog ne contiene quindi solo 100. Nel momento del salvataggio dello schedule, i vari pezzi da 100 slot dello schedule vengono caricati prima sulla struttura SlotStructLog e poi memorizzati sulla FLASH.

4.4 Gestione della collisione dei messaggi radio

La suddivisione del tempo in slot ha funzione primaria di determinare quando accendere o spegnere il chip radio di un nodo. Questa strategia di risparmio energetico possiede però la anche la desiderabile caratteristica di diminuire la probabilità di collisione di messaggi sul canale radio. Difatti un protocollo dello strato MAC molto usato nelle comunicazioni wireless è il TDMA (Time Division Multiple Access), che consiste proprio nel suddividere il tempo in slot, e nel concedere in uno di essi

solamente a un nodo di trasmettere e a uno di ricevere. Gli schedule utilizzati nel protocollo FPS però, derivano solamente da informazioni locali, e non hanno quindi la vista globale della rete. Per questo motivo le collisioni non vengono del tutto evitate, come mostra l'esempio in Figura 4.4.

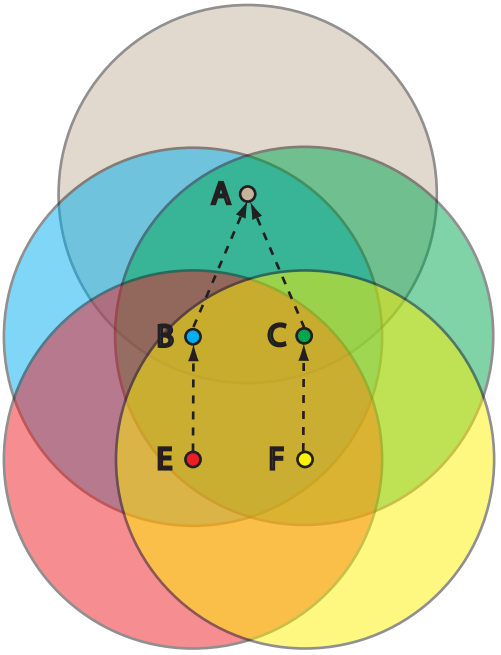


Figura 4.4: Esempio di collisione di messaggi nel protocollo FPS.

Nella figura i nodi sono rappresentati con i cerchi piccoli e le lettere, le linee tratteggiate sono i link che si vengono a formare tra i nodi nella fase di inizializzazione, e i cerchi grandi evidenziano la copertura radio di ogni nodo. I nodi E e F avendo genitori diversi, possono trasmettere un loro messaggio nello stesso slot, e poiché entrambi i genitori sono nel raggio di copertura radio di entrambi i nodi, su di essi può avvenire una collisione di messaggi. In questo caso la trasmissione viene corrotta, e i messaggi scartati. Per risolvere questo problema, gli autori dell'articolo che spiega il protocollo FSP [1], indicano che è sufficiente utilizzare un semplice meccanismo CSMA. Il chip radio CC2420 del Tmote Sky permette l'implementazione del CSMA, attraverso la funzione CCA (Clear Channel Assessment). Il CCA si basa sulla misura del valore di RSSI e su una soglia programmabile. Per sfruttare tale funzione si deve utilizzare il comando del CC2420, STXONCCA. Effettuando la trasmissione con tale comando, invece che con il comando STXON, la trasmissione avviene infatti soltanto se il canale è libero. TinyOS incorpora dei componenti che sfrutta-

no il comando STXONCCA del CC2420. E' possibile infatti per ogni trasmissione effettuata attraverso l'interfaccia *Send* di TinyOS, impostare due diversi tempi di attesa. Il primo, detto "InitialBackoff", è un ritardo del primo tentativo di trasmissione del messaggio. Questo ritardo avviene sempre, e inizia con l'invocazione del comando *send()*. Il secondo tempo di attesa, detto "CongestionBackoff", inizia invece ogni volta che nel tentativo di trasmettere un messaggio, il mote trova il canale occupato. Per impostare i due tempi di attesa si deve gestire la notifica degli eventi *requestCongestionBackoff* e *requestInitialBackoff*, dell'interfaccia *RadioBackoff*. La notifica di questi eventi avviene ogni volta che viene trasmesso un messaggio. Nel gestore degli eventi i valori dei due tempi vengono definiti con i comandi *setCongestionBackoff()* e *setInitialBackoff()*, sempre della stessa interfaccia.

Purtroppo il CSMA da solo non risolve il noto problema dei *nodi nascosti*, che utilizzando l'FPS può verificarsi in una situazione come quella illustrata in Figura 4.5.

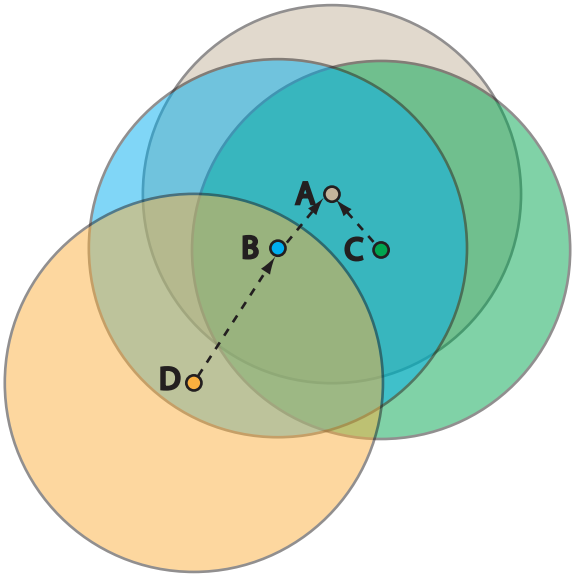


Figura 4.5: Problema dei nodi nascosti nel protocollo FPS.

I nodi D e C avendo padri diversi, possono trasmettere un loro messaggio nello stesso slot, ma a differenza dell'esempio precedente in questa situazione il meccanismo CSMA non interviene in quanto i due nodi non si vedono. Se in uno slot in cui D invia un messaggio a B, C invia un messaggio ad A, le due trasmissioni si sovrappongono in B che riceverà un messaggio corrotto. Questa situazione se si

verifica, purtroppo si ripete ad ogni ciclo.

Un altro svantaggio dello schema CSMA sta nel fatto che se un nodo trova il canale occupato, posticipa la trasmissione in un istante successivo. Se in questo istante il canale viene ritrovato occupato, la trasmissione viene posticipata ulteriormente, e così via. Questo significa che la durata di uno slot deve essere tale da consentire l'invio in sequenza di più messaggi e dei rispettivi Acknowledge, che vengono utilizzati per gestire la qualità dei link, si veda Paragrafo 4.8. E' necessario quindi mantenere uno slot di durata piuttosto lunga per avere la certezza che alcuni messaggi non finiscano nello slot successivo, creando ulteriori problemi. Questa impostazione è però negativa in quanto maggiore è la durata di uno slot, maggiore è la latenza dei messaggi, e soprattutto maggiore è il consumo di energia, in quanto aumenta la frazione di tempo nel quale la radio rimane accesa.

Per risolvere questi problemi si è deciso per il sistema WirMoS di non utilizzare il meccanismo CSMA durante la fase di regime, ma di estendere l'algoritmo FPS con aggiunta della tecnica FDMA (Frequency Division Multiple Access), che consiste nel utilizzo da parte di diverse sorgenti, di differenti frequenze di trasmissione, evitando quindi la sovrapposizione delle loro trasmissioni. La piattaforma Tmote Sky permette di scegliere al momento della trasmissione di un pacchetto uno fra sedici canali disponibili, si veda Paragrafo 2.2. Per realizzare lo schema FDMA un nodo prima di inviare un messaggio di Reservation Confirm, sceglie a caso uno dei sedici canali e inserisce questo valore in un campo del messaggio. Il nodo figlio che riceve il messaggio legge il valore di quel campo, e viene così a conoscenza del canale sul quale trasmettere i campioni di rilevazione. Il canale viene impostato all'inizio di ogni slot utilizzando i comandi *setChannel()* e *sync()* dell'interfaccia *CC2420Config*. Il primo serve per impostare il canale desiderato, mentre il secondo per sincronizzare l'hardware al nuovo cambiamento. Fortunatamente quest'operazione è molto veloce (< 1 ms) e i due comandi possono essere eseguiti uno dopo l'altro in sequenza. Due dei sedici canali disponibili nel Tmote Sky vengono utilizzati in WirMos per la sincronizzazione e le informazioni di controllo, si veda Paragrafo 4.5. E' possibile scegliere quali sono questi due canali, e quali dei rimanenti utilizzare per inviare i campioni di rilevazione. La scelta dei canali d'essere fatta in base a quali sono i canali liberi nel sito dove viene dispiegato il sistema. La banda di frequenze utilizzata dal Tmote Sky è infatti sfruttata anche da altre tecnologie come ad esempio WiFi. E' necessario quindi prima di installare il sistema effettuare con qualche tool un'analisi per ogni canale, della qualità di trasmissione del Tmote Sky. Introducendo

queste modifiche la probabilità di collisione sui campioni di rilevazione diminuisce di un fattore $1/n$, dove n è il numero di canali utilizzati per questi messaggi.

Se nonostante questi accorgimenti due nodi con genitori diversi inviano messaggi che collidono perché trasmettono nello stesso slot, con lo stesso canale, e la loro zona di copertura radio si sovrappone sui destinatari, i nodi cambiano semplicemente il canale su cui trasmettere. Utilizzando questa strategia al posto del CSMA quindi, non solo si ottiene una forte diminuzione delle collisione, ma si risolve il problema dei nodi nascosti, e sopratutto non è necessario imporre gli slot di durata lunga a sufficienza per consentire il posticipo della trasmissione dei messaggi, mantenendo così bassa la latenza e il consumo energetico. Per fare in modo che i nodi rilevino le collisioni si utilizza lo stesso meccanismo di gestione della qualità dei link spiegato nel Paragrafo 4.8. Quando un nodo genitore rileva che avvengono delle probabili collisioni in uno slot, sceglie un altro canale per la ricezione dei messaggi in quel slot e comunica la scelta al figlio inviando un messaggio in uno slot B (si veda prossimo paragrafo).

4.5 Broadcast e sincronizzazione

Utilizzando l'algoritmo FPS così com'è stato spiegato, la comunicazione può avvenire in una sola direzione, dal basso verso la radice dell'albero. Spesso è invece necessario che ci sia la possibilità di comunicare dall'alto verso il basso, ad esempio per far arrivare a tutti i nodi, o a un particolare nodo, dei comandi o delle interrogazioni provenienti dal PC, o per effettuare la sincronizzazione. Per realizzare questo tipo di comunicazione si utilizzano altri due tipi di slot, B (Broadcast) e RB (Receive Broadcast). Ogni nodo ha uno slot dello schedule definito come B e uno slot definito come RB. La Base Station ha solo lo slot B, perché i comandi li riceve via USB, e non ha bisogno di sincronizzazione. Quando un nodo si trova in uno slot B, se ha la necessità invia un messaggio, che può essere un comando, una query, un messaggio di sincronizzazione, etc., in broadcast. Se il messaggio è diretto a un qualche nodo particolare, è comunque necessario inviare il messaggio in broadcast, questo perché i nodi non conoscono i loro discendenti, e quindi non sanno su quale figlio inoltrare la richiesta. Di conseguenza ogni comunicazione che parte dal PC per arrivare a un qualche nodo, coinvolgerà in realtà tutti i nodi, i quali dovranno eventualmente scartare il messaggio, se non sono il destinatario del messaggio trasmesso. Se lo slot definito come B in un nodo ha indice i, anche lo slot definito

come RB di ogni figlio dovrà avere indice i. Per ottenere questo risultato in modo semplice si è deciso di definire come B lo slot che ha per indice il TOS_NODE_ID del nodo stesso, mentre come RB lo slot che ha per indice il TOS_NODE_ID del genitore, quest'ultimo contenuto nella struttura BaseStationPath. Questa scelta oltre che a semplificare la costruzione della rete, evitando ulteriori prenotazioni FPS, evita che ci siano collisioni tra i messaggi inviati in questi slot, in quando essendo il TOS_NODE_ID univoco, non ci saranno due o più nodi aventi lo stesso indice per lo slot B. Inoltre per la trasmissione negli slot B si utilizza un canale separato. In questo modo i messaggi non possono collidere nemmeno con i messaggi conteneti i campioni di rilevazione, o informazioni di controllo del protocollo FPS, quindi non possono mai collidere. Questo risultato è molto importante per l'invio dei messaggi di sincronizzazione che non devono subire ritardi aleatori, e che devono essere persi il meno possibile.

Introducendo gli slot di tipo RB nei quali il nodo mantiene il chip radio acceso, è necessario effettuare una modifica l'algoritmo di base, per evitare che un nodo risponda ad un Advertisement in questi slot quando $D(k) = S(k)$, introducendo un errore nella logica del protocollo FPS. E' quindi sufficiente scartare un messaggio di Advertisement se lo slot corrente è di tipo RB.

Come già anticipato la comunicazione dall'alto verso il basso dell'albero può essere utilizzata anche per la sincronizzazione dei nodi. Il problema della sincronizzazione è presente in tutti i sistemi distribuiti, e nasce dal fatto che la frequenza a cui oscilla il cristallo di un orologio di un dispositivo è sì abbastanza stabile, ma generalmente è leggermente diversa da dispositivo dispositivo. Quindi, in un sistema costituito da n dispositivi, tutti gli n cristalli oscillano a frequenza leggermente diversa. Come conseguenza gli n orologi dopo un po' di tempo vanno fuori sincrono, e la situazione peggiora con il passare del tempo. Il risultato di tutto ciò è che per i vari dispositivi non c'è la possibilità di accordarsi sull'eseguire una determinata operazione, in un certo istante temporale, in quanto quell'istante sarà in generale diverso per tutti. Per per risolvere questo problema sono stati progettati diversi algoritmi di sincronizzazione, tra cui il noto NTP (Network Time Protocol) il più utilizzato in internet.

Anche nel protocollo FPS la sincronizzazione è importante, in quanto si basa sulla suddivisione del tempo in slot, dando per scontato che l'i-esimo slot di un ciclo inizi esattamente nello stesso istante per tutti. Il protocollo consente comunque di mantenere l'intervallo temporale di uno slot abbastanza lungo, richiedendo quindi

una sincronizzazione non estremamente fine. E' necessario tuttavia implementare un meccanismo di sincronizzazione, in quanto il disallineamento degli orologi porta a lungo andare ad un disallineamento degli slot, tale da compromettere la correttezza dell'algoritmo. Per effettuare la sincronizzazione un nodo quando si trova nello slot B come prima operazione invia in broadcast un messaggio di tipo Rensync. Questo messaggio viene inviato istantaneamente ponendo il tempo InitialBackoff a 0 (si veda Paragrafo 4.4). Sono state fatte diverse prove per misurare l'RTT (round trip time) tra due Tmote Sky, trasmettendo un messaggio di 13 byte, che è la dimensione del messaggio contenente un campione di rilevazione. Il risultato di queste prove è stato sempre RTT=10 ms, indipendentemente dalla distanza tra i due mote, quindi il tempo di trasmissione è circa 5 ms. Quando un nodo riceve un messaggio Resync, controlla innanzitutto se proviene dal genitore, e se si, riavvia l'FPSslotTimer con tempo $T_s - 5$, e pone come slot corrente lo slot definito RB, in quanto quest'ultimo ha indice pari a quello dello slot B del genitore.

Quest'ultima operazione si rende necessaria nel caso in cui il messaggio di Resync arrivi ad un nodo in uno slot precedente a quello definito RB, quindi in uno slot con indice diverso dall'indice dello slot corrente del genitore. Difatti a seconda che sia l'orologio del genitore ad essere in anticipo su quello del figlio, o viceversa, possono presentarsi le due situazioni illustrate in Figura 4.6 con frecce nere continue. Se per esempio il figlio ha un ritardo maggiore a 5 ms, il messaggio di sincronizzazione avviato dal genitore nello slot n, arriva al figlio nello slot $n - 1$, come mostra la freccia nera della Figura 4.6. E' probabile che il primo messaggio di sincronizzazione non venga ricevuto nello slot RB, in quanto la fase di setup della rete dura parecchi secondi, e il disallineamento che si crea è quindi grande. La prima sincronizzazione però avviene durante la fase delle prenotazioni FPS iniziale, nella quale tutti i nodi hanno la radio accesa in tutti gli slot, quindi c'è la certezza che il messaggio di sincronizzazione venga ricevuto. Nelle successive sincronizzazioni può accadere che nello slot dove viene ricevuto il messaggio, la radio sia è spenta.

Una soluzione al problema può essere quella di ritardare l'invio del messaggio di sincronizzazione di un tempo pari a $\Delta t > \delta t - 5$, dove δt è il più grande disallineamento temporale che si può formare tra gli orologi di due nodi, tra una sincronizzazione e l'altra. In questo modo il nodo figlio riceverà il messaggio di sincronizzazione sicuramente nello slot RB, nel quale la radio è accesa, ma dovrà tener conto del fatto che lo slot B del genitore è iniziato già da circa $\Delta t + 5$ ms. Le frecce rosse tratteggiate in Figura 4.6 mostrano questa strategia. Per evitare che il messaggio di sincronizza-

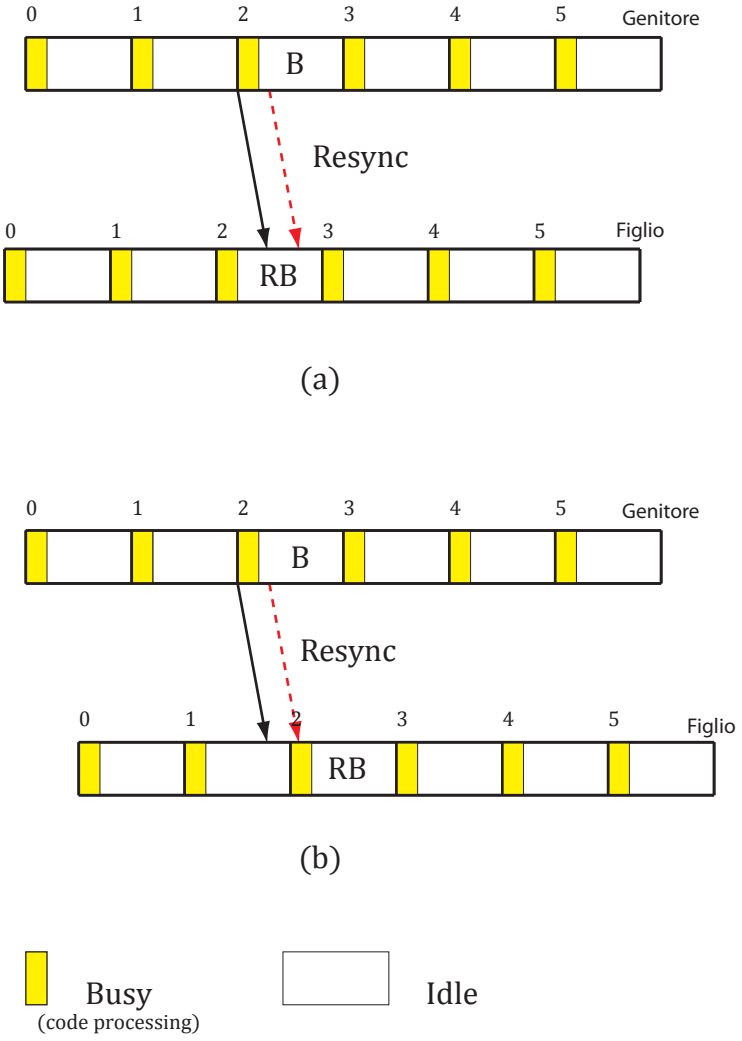


Figura 4.6: Sincronizzazione tra genitore e figlio. (a) L'orologio del genitore e il ritardo su quello del figlio. (b) L'orologio del genitore è in anticipo su quello del figlio.

zione venga ricevuto nello slot successivo a quello RB, quando è il genitore ad essere un ritardo sul figlio, deve essere $T_s > \delta t + 5$.

Quando un messaggio viene ricevuto su uno slot definito RB, il nodo che lo ha ricevuto lo inoltra sullo stesso ciclo o su quello successivo, a seconda del fatto che lo slot definito B abbia indice maggiore o minore di quello RB. Questo significa che un messaggio che parte dalla Base Station può impiegare diversi cicli prima di arrivare a destinazione. La Figura 4.7 mostra il caso peggiore, nel quale un messaggio che parte dal mittente nel ciclo I viene ricevuto al ciclo $i + N_h - 1$ dal destinatario, dove N_h è il numero di hop che separa il mittente dal destinatario, con una latenza di circa $(N_h - 1)T_c$ secondi. Per quanto riguarda la sincronizzazione, questo non significa che

sia necessario avviarla ogni H cicli, in quanto quello che conta è la sincronizzazione tra genitore e figlio, e quindi può essere effettuata con una frequenza indipendente dall'altezza dell'albero, anche ogni ciclo.

Il numero di volte che la radio rimane accesa negli slot B e RB all'interno del periodo di campionamento influisce molto sui consumi, e per questo motivo si è deciso di ridurre al minimo questo valore. Si decide quindi di mantenere la radio accesa in questi slot un numero di volte, che sia il più piccolo valore possibile, sufficientemente grande però per mantenere una sincronizzazione adeguata. Nel prossimo paragrafo viene descritto che per avere un bassa latenza dei messaggi, si deve cercare di minimizzare il valore di T_s. Più piccolo è il valore di T_s, minore però deve essere il disallineamento massimo δt. Dalle prove fatte si è visto che è sufficiente effettuare una sincronizzazione ogni 30 secondi. Questo valore è stato scelto tendo conto anche della possibilità che alcuni messaggi di sincronizzazione vadano persi, e che quindi il disallineamento massimo sia maggiore di quello che si ottiene in 30 secondi. Ovviamente se viene perso un numero di messaggi di sincronizzazione consecutivi maggiore di una certa soglia, la sincronizzazione viene persa, e il sottoalbero che ha per radice il nodo che ha perso la sincronia si sconnette dalla rete. In questa versione del sistema non c'è la possibilità di recuperare la sincronizzazione, che resta una funzione che fa parte degli sviluppi futuri. Per fare in modo che un nodo accenda la radio nello slot RB del ciclo esatto, per ricevere la sincronizzazione, si deve calcolare quanti cicli impiega il messaggio di sincronizzazione, che parte dalla radice, ad arrivare al nodo. Questo calcolo viene fatto inserendo nel messaggio di sincronizzazione, il numero di ciclo nel quale il messaggio viene inviato dalla Base Station. La prima volta che un nodo riceve un messaggio di sincronizzazione calcola l, la latenza in cicli, effettuando la differenza tra il valore del ciclo attuale e il valore memorizzato nel messaggio. In questo modo se si è deciso che la sincronizzazione deve essere avviata ogni n cicli, in nodo accenderà la radio nello slot RB, ogni $n + l$ cicli.

4.6 Latenza dei messaggi

La latenza di un messaggio è l'intervallo di tempo che intercorre da quando il messaggio viene inviato, a quando viene ricevuto sul PC. Nel caso peggiore un messaggio che parte da un nodo al livello l nel ciclo i può arrivare al livello $l - h$ (compiendo quindi h hop) nel ciclo $i + h - 1$, con una latenza di circa $(h - 1)T_c$, similmente a

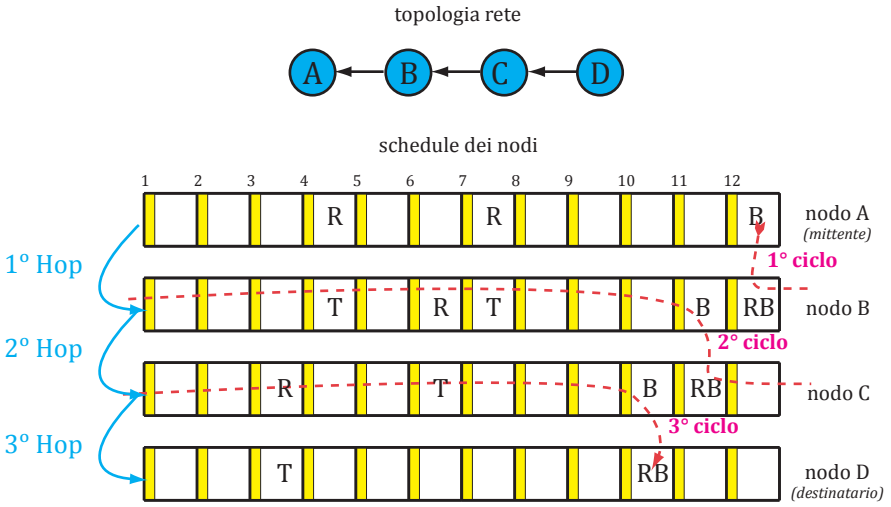


Figura 4.7: Caso pessimo per la latenza dei comandi e delle sincronizzazioni.

quanto visto nel paragrafo precedente per il broadcast. La Figura 4.8 illustra questa situazione.

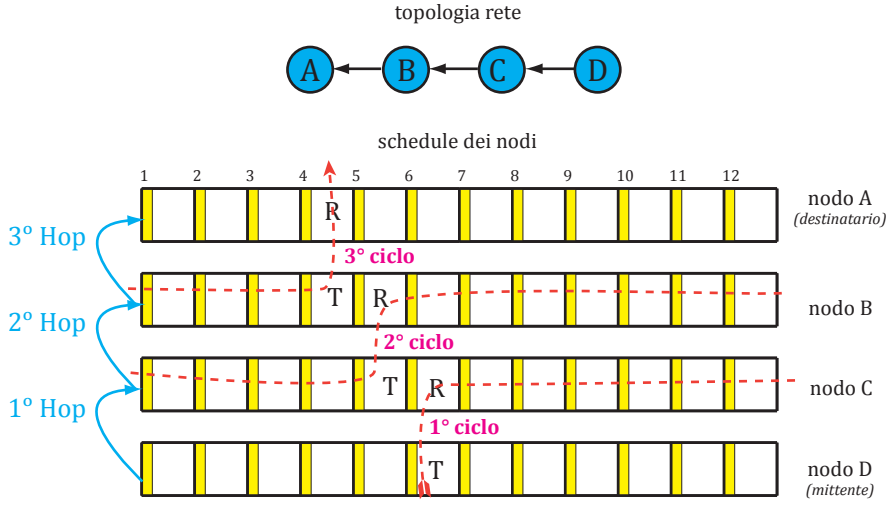


Figura 4.8: Caso pessimo per la latenza delle trasmissioni.

Per questo motivo il valore al caso pessimo (worst case) della latenza, di un qualsiasi messaggio inviato da un nodo della rete con destinatario la Base Station è dato dal'Equazione (4.3):

$$\begin{aligned} L_{wc} &= T_c(H-1) \\ &= T_s N_{slot}(H-1) \end{aligned} \tag{4.3}$$

dove H è l'altezza dell'albero. Per minimizzare L_{wc} si deve minimizzare T_s o N_{slot}. Per quanto riguarda il primo valore, questo deve essere sufficientemente grande da consentire l'invio di due messaggi, il campione di rilevazione e il suo Acknowledge. Poiché il tempo di trasmissione è di 5 ms, si può impostare un valore minimo di T_s di 15 ms, che tiene conto anche del possibile disallineamento che si viene a formare tra una sincronizzazione e l'altra. Per minimizzare N_{slot} si deve innanzitutto capire qual'è il numero minimo di slot, che lo schedule deve avere, per avere la certezza che prima o poi sia possibile per un nodo, portare a termine tutte le prenotazioni FPS. La Figura 4.9 mostra lo schedule di tre nodi, il nodo A genitore e i nodi B e C figli. I nodi B e C hanno domanda rispettivamente 3 e 2, quindi il genitore ha domanda 6. Gli slot T e R dei tre schedule sono disposti nel caso peggiore per il nodo A, ovvero nel caso che richiedere il maggior numero di slot per A per riuscire prima o poi a soddisfare tutte le prenotazioni.

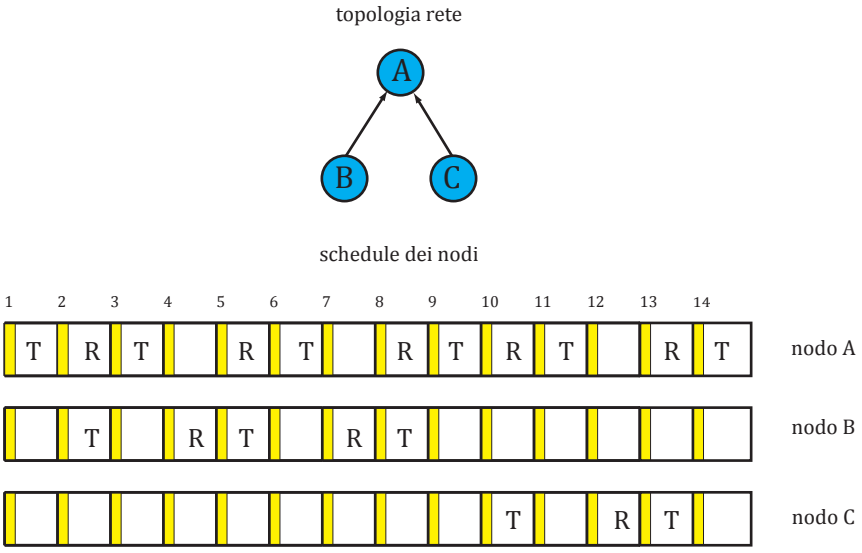


Figura 4.9: Grandezza schedule al caso peggiore.

Come si vede dalla figura, il caso peggiore si ha quando gli slot R degli schedule dei figli, hanno tutti indice diverso da quello degli slot T del genitore. In questo

caso, e se la domanda iniziale di ogni nodo è 1, *il numero di slot minimo che deve avere lo schedule per il nodo j è dato dall'Equazione* (4.4):

$$\begin{aligned} N_{slot}(j) &= 2N_d(j) + (N_d(j) - N_{so}(j)) + 1 \\ &= 3N_d(j) - N_{so}(j) + 1 \\ &= 3N_{dn}(j) + 2N_{so}(j) + 1 \end{aligned} \tag{4.4}$$

dove $N_d(j)$ (Number of Descendant) è il numero di discendenti del nodo j, $N_{so}(j)$ (Number of Sons) è il numero di figli del nodo j, e $N_{dn}(j)$ (Number of Descendant Not sons) è il numero di discendenti del nodo j ma che non sono suoi figli. Osservando la formula 4.4 vediamo che per j serve uno slot per inviare il proprio messaggio, $2N_d(j)$ slot definiti T e R, e $N_d(j) - N_{so}(j)$ slot in più che lo schedule deve avere, e corrispondono alla somma degli slot R dei figli. Nell'esempio $N_{dn}(A) = 3$, $N_{so}(A) = 2$, e quindi $N_{slot}(A) = 14$.

Per lo schedule della Base Station il discorso è simile ma essendo la radice dell'albero non possiede slot di tipo T, quindi vale l'Equazione (4.5).

$$\begin{aligned} N_{slot}(BS) &= N_d(BS) + (N_d(BS) - N_{so}(BS)) \\ &= 2N_d(BS) - N_{so}(BS) \\ &= 2N_{dn}(BS) + N_{so}(BS) \end{aligned} \tag{4.5}$$

Utilizzando uno schedule nell'FPS con un numero di slot $N_{slot} = N_{slot}(j)$, con j indice di un nodo qualsiasi compreso quello della Base Station, e tale che $N_{slot}(j)$ sia massimo, siamo quindi sicuri che la fase di prenotazioni prima o poi termina. L'Equazione (4.6) mostra il valore di N_{dn} di un nodo j a profondità $h > 0$, in funzione dei valori N_{dn} e N_{so} dei nodi in Γ_{h+1} (Γ_z è l'insieme dei nodi a profondità z). Essa dimostra che il valore di $N_{slot}(j)$ di un un nodo $j \in \Gamma_k$ è sicuramente maggiore di quello di un nodo $i \in \Gamma_{k+1}$. In conclusione lo schedule dovrà avere dimensione pari al maggiore tra $N_{slot}(j)$ con $j \in \Gamma_1$ e $N_{slot}(BS)$.

$$
\begin{aligned}
3N_{dn}(j \in \Gamma_h) &= 3 \sum_{i \in \Gamma_{h+1}} (N_{dn}(i) + N_{so}(i)) \\
&= \sum_{i \in \Gamma_{h+1}} (N_{slot}(i) + N_{so}(i) - 1) \qquad (4.6)
\end{aligned}
$$

Una buona approssimazione per eccesso della 4.4 è $3N_d(j)$, quindi per usare un unico valore, dato che $N_d(BS) > N_d(j)\ \forall\ j \neq BS$ possiamo porre, sovradimensionando quando $N_{slot}(BS) > N_{slot}(j)\ \forall\ j \neq BS$, $N_{slot} = 3N_d(BS) \simeq 3N_n$ dove N_n è il numero di nodi massimo della rete.

C'è però un'altra considerazione da fare e riguarda il fatto che quando un nodo definisce uno slot I come RP in una prenotazione FPS, potrebbe non esserci nessun figlio che ha uno slot I, con indice uguale a quello RP del genitore. In questo caso il ciclo corrente è inutile ai fini delle prenotazioni, e la fase in cui i nodi cercano di soddisfare la propria domanda si prolunga. Di conseguenza più più piccolo è il numero di slot degli schedule, minore è la probabilità per un nodo, di scegliere degli slot che sono I anche per qualche figlio, e maggiore è la probabilità che la fase delle prenotazioni si prolunghi. Per questo motivo in certe reti "non buone", ovvero nelle quali ci sono dei nodi figli, che hanno circa lo stesso numero di discendenti del nodo figlio che ha il massimo numero di discendenti, uno schedule con un numero di slot pari a $3N_n$ potrebbe essere troppo piccolo, ritardando troppo la conclusione della fase delle prenotazioni iniziale. A meno di non aggiungere delle opportune contromisure quindi, se l'albero è di questo tipo, oppure non si sa com'è, conviene utilizzare un numero di slot maggiore, ad'esempio $4N_n$.

Utilizzando uno schedule con il numero minimo di slot, ovvero $4N_n$, e T_s di durata minima, ovvero 15 ms, otteniamo il valore minimo della latenza al caso pessimo $L_{wc} = 60N_n(H-1)$. Il grafico in Figura 4.10 mostra questo valore al variare del numero dei nodi della rete, e per diversi valori di H.

4.7 Supercicli

Nel paragrafo precedente si è visto che in FPS per diminuire la latenza si diminuisce il numero di slot dello schedule, e quindi la durata del ciclo FPS, che corrisponde al tempo di campionamento. Questo significa che c'è un maggior consumo in quanto i nodi trasmettono più spesso. Se la diminuzione del tempo di campionamento non è necessaria per la particolare applicazione, diminuire la latenza diventa un

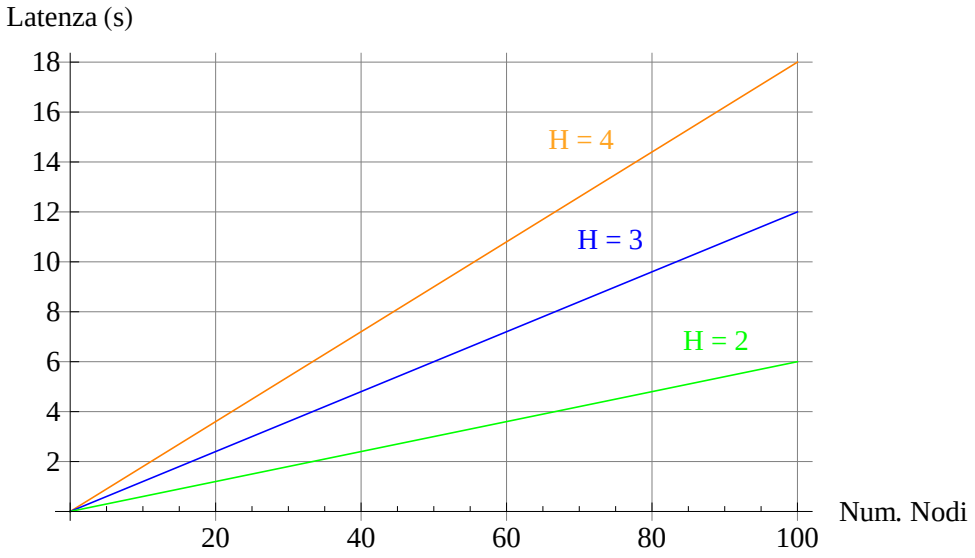


Figura 4.10: Latenza minima al variare del numero dei nodi e dell'altezza dell'albero H.

operazione molto negativa, in quanto produce un aumento dei consumi inutile. Per permettere la diminuzione della latenza, evitando che diminuisca anche il tempo di campionamento, e quindi aumentino i consumi, si introducono i supercicli. Un *superciclo* è un insieme di $\alpha \geq H$ cicli consecutivi dell'algoritmo FPS, dove $T_{sc} = \alpha T_c$ (SuperCycle Time) è il tempo di campionamento del sistema, al posto di T_c. La prima parte del superciclo, è detta *parte attiva*, ed è costituita dai primi H cicli, dove H è l'altezza dell'albero, mentre la restante è detta *parte inattiva*, vedi Figura 4.11.

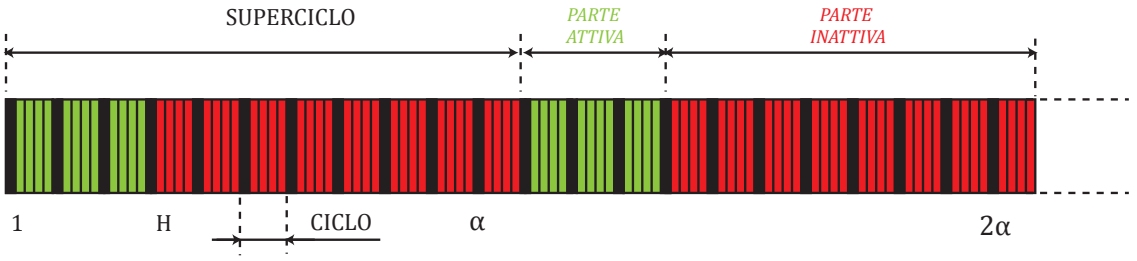


Figura 4.11: Definizioni di superciclo, parte attiva, parte inattiva.

Nella parte attiva di un superciclo avvengo le trasmissioni dei campioni di rilevazione, mentre nella parte inattiva in tutti gli slot di tipo T e R la radio rimane spenta. Considerando la trasmissione dei soli campioni di rilevazione, in un albero

di altezza H, nel primo ciclo della parte attiva trasmettono solo i nodi a profondità H, e ricevono solo i nodi a profondità $H-1$. Nel secondo ciclo della parte attiva trasmettono solo i nodi a profondità $H-1$, e ricevono solo i nodi a profondità $H-2$, e così via. Quindi nel n ciclo della parte attiva trasmettono solo i nodi a profondità $H-n+1$, e ricevono solo i nodi a profondità $H-n$. In uno slot che non è usato per ricevere o trasmettere, si mantiene la radio spenta. In questo modo la radio rimane accesa solo negli slot effettivamente necessari, per ricevere nella Base Station un solo campione di ogni nodo per superciclo. Di conseguenza il numero di slot T e R della parte attiva, nei quali la radio rimane accesa, è uguale numero di slot T e R di un solo ciclo.

Ogni nodo non Base Station ha due variabili ALPHA=α, e TREE_HEIGHT=H, che indicano quando accendere e spegnere la radio negli slot T e R. L'altezza dell'albero viene calcolata dalla Base Station, dopo qualche ciclo dalla fine della fase delle prenotazioni. Il calcolo si ottiene analizzando i messaggi inviati verso il PC dai nodi, che contengono anche il numero di hop che li distanziano dalla radice. Utilizzando questo valore la Base Station pone come altezza dell'albero la massima profondità rilevata. Periodicamente il valore dell'altezza dell'albero viene ricalcolato, dato che ci possono essere variazioni nella topologia della rete. La modifica del tempo di campionamento mantiene, a differenza che nell'FPS originale, invariato il numero di slot per ciclo, e quindi la latenza. Se si aumenta il tempo di campionamento, l'effetto è invece quello di aumentare il numero di cicli nella parte inattiva. Il tempo in cui la radio è accesa in un superciclo quindi non cambia, ma aumenta la durata del superciclo, e quindi diminuisce il consumo medio. In conclusione con questa soluzione è possibile variare a piacere il tempo di campionamento, senza modificare la latenza dei messaggi, e ottenere inoltre una proporzionale variazione dei consumi.

Una volta impostato $T_s \geq 15$ ms e $N_{slot} \geq 4N_n$ nelle applicazioni installate sui sensori, si può scegliere attraverso WirMos-Dashboard il tempo di campionamento desiderato T_{su} (Sample User Time). Poiché questo valore deve essere un multiplo di T_c, il tempo di campionamento effettivo, ovvero la durata di un superciclo è $T_{sc} = Round(T_{su}/T_c)T_c$ che è il multiplo di T_c più vicino a T_{su}.

Quando inizia la fase di regime i supercicli sono disabilitati, quindi il nodo esegue il protocollo FPS nella sua versione originale. Come già accennato, la Base Station nel momento in cui si accorge che è finita la fase delle prenotazioni FPS iniziale, invia un messaggio ChangeMsg con il campo *change* impostato a FALSE ai propri figli. Quando un nodo riceve questo messaggio abilità i supercicli, e inoltra il messaggio ai

propri figli. Dopo un intervallo temporale tutti i nodi hanno ricevuto il ChangeMsg, e abilitato i supercicli. Per essere sicuri che tutti i nodi ricevano il ChangeMsg, ogni nodo invia questo messaggio cinque volte. I nodi inseriscono nel messaggio che contiene il campione di rilevazione, anche l'informazione relativa all'abilitazione dei supercicli. In questo modo l'utente ha la possibilità di controllare in ogni momento se un nodo ha, o non ha, i supercicli abilitati, ed eventualmente forzare la condizione.

4.8 Variazioni della topologia

In un ambiente dove le trasmissioni avvengono senza fili, può accadere che un link che fino a poco tempo prima funzionava perfettamente, un attimo dopo non permetta più la trasmissione di dati. Il motivo sta nella possibilità della nascita di interferenze dovute all'introduzione nell'ambiente da controllare, di campi elettromagnetici, corpi solidi schermanti, etc... .

Per permettere la variazione della topologia in questa situazione, si utilizza una nuova struttura dati chiamata *controlSlots*, un array di interi della stessa dimensione dell'array che rappresenta lo schedule locale. Ogni cella di controlSlots è un contatore che viene incrementato ogni volta che si riceve un messaggio di dati nello slot che ha per indice, l'indice di quella cella. Ogni N_{scc} (Number of SuperCycle for Control) supercicli, viene controllato nell'ultimo slot del primo ciclo il contenuto di ogni cella di controlSlots e poi impostato nuovamente a zero. Se il valore di una cella è minore di $N_{scc} \cdot 0.5$ significa che per il corrispondente slot, in quell'intervallo di tempo, sono stati ricevuti meno del 50% dei messaggi. In questo caso lo slot R e uno slot T qualsiasi vengono definiti nuovamente I, e decrementati di uno $S(k)$ e $D(k)$. Nel frattempo la stessa cosa avviene anche in tutti gli ascendenti del nodo fino alla Base Station, in quanto anche loro avranno rilevato la perdita dei messaggi.

Per fare in modo che anche il nodo che trasmette modifichi il suo stato, se il link con il genitore diventa non affidabile, ogni volta che quest'ultimo riceve un messaggio contenente dati, invia un messaggio Acknowledge in risposta a quelli che riceve. Un contatore, N_{ack} (Number of Acknowledge), mantiene il numero dei messaggi Acknoledge ricevuti in N_{scc} supercicli, mentre il contatore N_{tr} mantiene il numero di messaggi di dati trasmessi sempre nello stesso intervallo. Dopo N_{scc} supercicli viene controllato il valore dei contatori e se $N_{ack} < 0.5N_{tr}$ significa che sono stati ricevuti meno del 50% degli Acknowledge. Quando questo accade tutti gli slot di tipo

T vengono ridefiniti come I, e diminuito di conseguenza il valore dell'offerta $S(k)$. Conclusa questa operazione il nodo sceglie un nuovo genitore nella AdjTable, e la scelta ricade sul nodo che dista il minor numero di hop dalla radice. Una volta che è stato ridefinito il genitore, poiché il nodo si trova nella situazione $D(k) > S(k)$, disabilità i supercicli e si mette in ascolto di messaggi di Advertisement dal nuovo genitore, per cercare di soddisfare la sua domanda. Questa strategia della scelta del genitore si basa su informazioni raccolte nella fase di inizializzazione della rete, perché si assume che i mote non si spostino. Se si vuole effettuare uno spostamento di un nodo, si deve reinizializzare la rete. E' stata fatta questa scelta in quanto se un nodo interno si sposta, anche tutti i figli devono cercare un nuovo genitore. E' necessario in questo caso un meccanismo più complesso di ricostruzione della rete che potrebbe impiegare un tempo troppo elevato, risultando quindi più conveniente in questo caso il reset della rete

Quando la Base Station ridefinisce come I degli slot R e T, trasmette in broadcast nel prossimo slot B dove i figli hanno la radio accesa un messaggio di tipo ChangeMsg con il campo *change*=TRUE. Un nodo non Base Station che riceve nello slot RB un messaggio di questo tipo disabilità i supercicli, abilita gli Advertisement e inoltra il messaggio nel prossimo slot B dove ha la radio accesa. Dopo un certo tempo il broadcast verrà completato, e quindi tutti i nodi si comporteranno allo stesso modo. Il nodo che ha cambiato genitore, può così ricevere da questo gli Advertisement, e soddisfare la sua domanda. Come già spiegato la Base Station riconosce quando tutte le domande sono nuovamente soddisfatte, quindi anche in questo caso può inviare un messaggio ChangeMsg con con il campo *change*=FALSE, che verrà ricevuto da tutti i nodi per abilitare nuovamente i supercicli e disabilitare gli Advertisement. Come meccanismo di sicurezza la Base Station invia comunque il secondo messaggio ChangeMsg se la situazione iniziale non si è ripristinata entro un certo intervallo di tempo dall'invio del primo ChangeMsg.

Se l'utente desidera inserire un nuovo nodo nella rete senza dover effettuare la reinizializzazione, invia attraverso l'applicazione WirMos-Dashboard, un comando alla Base Station per avvertire tutti i nodi di disabilitare i supercicli e abilitare gli Advertisement. Successivamente inserisce un nuovo nodo nella rete, il quale una volta acceso si sincronizza con un nodo qualsiasi alla prima ricezione di un messaggio Resync, e trasmette nell'ultimo slot dello schedule, che è sempre I per tutti i nodi, un messaggio di Ping per $N_P = 40$ cicli. I nodi che ricevono i Ping aggiornano la

PingTable aggiungendo un nuovo record. I Ping sono numerati, e il primo trasmesso è il numero uno. Assumendo che il primo Ping ricevuto da un nodo sia il numero il x (quindi i primi $x-1$ Ping non sono stati ricevuti da quel nodo), dopo $N_P - x$ cicli questo nodo trasmette, se il numero di Ping ricevuti è sufficiente, il record della PingTable al nuovo nodo della rete, come accade nella fase di Linking. La trasmissione del record avviene nello slot B del nodo mittente in modo da evitare collisioni. Oltre che il record viene inviato anche il numero di hop che distano dal mittente alla BaseStation. Quando il nuovo nodo riceve i record della PingTable costruisce la AdjTable, e sceglie come genitore il vicino con il minor numero di hop. Dopo questa fase il nuovo nodo inserito si mette in ascolto degli Advertisement dal genitore per soddisfare la propria domanda. Quando la Base Station si accorge che la domanda del nuovo nodo è stata soddisfatta, comunica a tutti gli altri nodi di abilitare i supercicli e disabilitare gli Advertisement. Anche in questo caso, come meccanismo di sicurezza, se dopo un certo intervallo di tempo la domanda del nuovo nodo non viene soddisfatta, la Base Station avverte comunque di riabilitare i supercicli e disabilitare gli Advertisement.

Capitolo 5

Interfacce sistema-utente

Questo capitolo descrive il software utilizzabile dagli utenti per interagire con il sistema. Per il software da utilizzare nella workstation è stato scelto il linguaggio di programmazione Java in quanto risulta essere quello maggiormente compatibile con TinyOS, e perché per tale linguaggio esiste il maggior numero di funzioni utili per lavorare con questo sistema operativo.

5.1 WirMoS-Dashboard

L'applicazione WirMoS-Dashboard è la principale interfaccia con l'utilizzatore, e permette di svolgere numerose attività riguardanti il sistema, come la visualizzazione dei dati provenienti dai sensori, la loro archiviazione su memoria secondaria, la visualizzazione di grafici relativi ai dati, il controllo della rete, il controllo automatico degli attuatori. WirMoS-Dashboard può essere utilizzata sia per il normale esercizio del sistema, sia nella fase di progettazione del sistema stesso, per identificare al meglio quali sono le esigenze della funzione di controllo.

5.1.1 WirMoS-Dashboard GUI

Per la realizzazione di WirMoS-Dashboard è stato utilizzato l'ambiente di sviluppo open source della Sun, NetBeans IDE 6.5.1. NetBeans assieme ad Eclipse sono gli IDE (Integrated Development Environment) più completi e utilizzati per sviluppare in Java. NetBeans è inlotre stato scelto come IDE ufficiale della Sun, che è proprietaria del linguaggio Java. Per WirMoS-Dashboard è stato scelto di realizzare in NetBeans un progetto di tipo *Java Desktop Application*. Questo modello fornisce un'infrastruttura di base per realizzare un applicazione di tipo desktop. L'infrastrut-

tura contiene il frame (finestra) principale, barra dei menù, barra di stato e la classe principale dell'applicazione che gestisce le funzioni di base del ciclo di vita dell'applicazione. Il nome del progetto creato per WNControl è WirMoS-DashboardPrj. Per la grafica è stata utilizzata la libreria Swing, fa parte di JFC Java Foundation Classes, un framework per realizzare GUI (graphical user interface) che aggiungere le funzionalità grafiche e l'interattività alle applicazioni Java. La libreria Swing include tutti i widget necessari a comporre una GUI, come bottoni, caselle di testo, finestre, etc. . I widget Swing forniscono una GUI più sofisticata rispetto alla precedente libreria grafica AWT (Abstract Window Toolkit). Questi controlli essendo scritti in puro Java, funzionano allo stesso modo su tutte le piattaforme (su cui Java può essere eseguito), al contrario dei controlli AWT, che sono legati al sistema grafico nativo del sistema operativo. NetBeans offre supporto alla creazione di GUI, sia che si utilizzi la libreria AWT o la Swing, attraverso il suo GUI builder, conosciuto con il nome di Matisse. Matisse semplifica la creazione delle GUI, permettendo lo sviluppatore di sistemare su di essa i widget con il drag-and-drop, usando un editor WYSIWYG (What You See Is What You Get), ottenendo così un feedback immediato, e senza dover mandare in esecuzione il programma per vedere il risultato della programmazione. Tutte le caratteristiche dei widget con Matisse vengono specificate attraverso delle finestre user-friendly, che NetBeans visualizza per semplificare e velocizzare la creazione della GUI. Un altro vantaggio importante che il GUI builder offre, è il supporto per gli eventi, che caratterizzano il software *event-driven*, come le applicazioni che dispongono di interfaccia grafica. Matisse permette infatti di gestire in modo veloce e semplice, gli eventi relativi a un componente dell'interfaccia, a partire da una lista di eventi disponibili per il particolare widget, e producendo in automatico una buona parte del codice relativo.

La classe principale di WirMoS-Dashboard è *WirMoS-DashboardPrjView* creata in automatico dall'IDE. Questa classe contiene la gestione dei principali eventi che vengono notificati quando l'utilizzatore interagisce con l'applicazione. La classe contiene anche il metodo *initComponent()*, anch'esso creato in automatico, che inizializza i vari componenti della GUI, in base alle impostazioni che sviluppatore ha determinato costruendo in modo visuale l'interfaccia. Ci sono però alcune impostazioni che è necessario, o più conveniente effettuare direttamente sul programma. Per questo motivo è stato creato il metodo *myInitComponent()*, che contiene tutte le impostazioni dei componenti non generate automatico. Il frame principale di WirMoS-Dashboard, contiene oltre alla barra dei menù, una barra degli strumenti,

con la quale è possibile avviare le operazioni principali in modo più semplice, cliccando su un'apposita icona, relativa all'operazione da effettuare. Il restante spazio del frame è suddiviso in due parti attraverso un'oggetto *javax.swing.jSplitPane*, la parte destra contiene dei controlli per i comandi utente, mentre la parte sinistra visualizza dati, relativi ai comandi impartiti nella parte destra. E' possibile modificare la porzione del frame principale occupata dalle due parti, spostando con il mouse il divisore al centro. La parte destra è un oggetto di tipo *javax.swing.jTabbedPane*, un pannello che permette di contenere dei tab. La parte sinistra ospita oggetti di tipo *javax.swing.jInternalFrame* che sono dei frame da utilizzare all'interno di oggetti *javax.swing.jFrame*.

5.1.2 Modellazione WSN e mote

Una WSN in WirMoS-Dashboard è rappresentata dalla classe *WSN*, la cui variabile d'istanza principale è *motes* che è un array di oggetti *Mote*. Questa variabile contiene le informazioni sui sensori che appartengono alla WSN. Ad ogni WSN è inoltre collegata un'immagine (formato jpg, bmp etc.), che rappresenta la pianta dell'area coperta dalla WSN. Su quest'immagine verranno inseriti dall'utente i simboli dei sensori in base alla loro reale posizione.

Per creare una nuova WSN l'utente deve selezionare 'Crea WSN' dal menù WSN oppure cliccare sulla relativa icona nella barra degli strumenti. Viene così reso visibile un frame che contiene tre text-box. Nel primo dei tre l'utente deve inserire il nome che vuole assegnare alla WSN, nel secondo deve inserire il percorso dell'immagine, che rappresenta la pianta della WSN da caricare sul programma, e nel terzo l'identificativo della Base Station. Questo identificativo deve essere uguale al TOS_NODE_ID della Base Station usato nell'applicazione WirmosBStationP.nc. Un pulsante 'Browse' permette di navigare tra le cartelle memorizzate nel hard disk della workstation per localizzare l'immagine da caricare. Premendo il pulsante 'Finish' su questo frame viene creata una nuova WSN, ovvero un istanza della classe *WSN*, e salvata nella memoria secondaria della workstation, utilizzando la classe *java.io.ObjectOutputStream*. Dopo questa operazione viene visualizzato un nuovo tab sulla parte sinistra della GUI, che contiene l'immagine della pianta della WSN creata. Sulla pianta viene inserito automaticamente un simbolo rappresentante la Base Station (rettangolo di colore giallo con all'interno l'identificativo del sensore), che l'utente può posizionare in base alla posizione fisica del mote. Nella parte sinistra della GUI vengono invece visualizzati i comandi per modificare la WSN nel

frame interno intitolato 'Modifica WSN'.

All'interno della cartella di installazione del programma c'è una cartella di nome `wsn` che a sua volta contiene delle cartelle create in automatico dall'applicazione, una per ogni WSN creata. Ognuna di queste cartelle ha per nome il nome della relativa WSN, e contiene un file con estensione .wsn che memorizza l'oggetto di tipo *WSN*, una copia dell'immagine che rappresenta la pianta della WSN, la cartella TQuality che contiene dati relativi alla qualità delle trasmissioni, e una cartella per ogni sensore che verrà creato per questa WSN.

Per caricare una WSN già creata, si deve utilizzare la voce 'Apri WSN' del menu WSN oppure la relativa icona nella barra degli strumenti. Cliccando su uno di questi elementi si apre una finestra che permette di navigare tra le cartelle, che si trovano all'interno della cartella `wsn`. Si deve quindi aprire la cartella con il nome della WSN da caricare, selezionare il file con estensione .wsn, e premere il pulsante 'Apri WSN'. Per recuperare dalla memoria secondaria della workstation l'oggetto WSN si utilizza la classe *java.io.ObjectImputStream*.

Un sensore wireless in WirMoS-Dashboard è rappresentato dalla classe *Mote* che mantiene tutte le informazioni necessarie per la sua gestione tra cui:

- TOS_NODE_ID
- Posizione sulla mappa
- Valore di temperatura, umidità, luminosità attuali
- Valore tensione batterie e durata batterie attuali
- Indicazione temporale dell'ultimo campione ricevuto
- L'informazione sullo stato di attivazione dei supercicli
- Numero di Acknowledge ricevuti nell'ultimo periodo di controllo (non in tutti i campioni)

Il pulsante 'Aggiungi mote' nel frame interno 'Modifica WSN' permette di aggiungere un nuovo sensore alla WSN corrente. Per aggiungere un nuovo sensore si deve inserire l'identificativo del sensore nella test-box affiancata al pulsante. Questo identificativo deve corrispondere al TOS_NODE_ID dell'applicazione WirmosNodeP.nc sul corrispondente Tmote Sky. Quando viene aggiunto un nuovo sensore sulla mappa, viene creato un nuovo simbolo di sensore (rettangolo azzurro con all'interno l'identificativo del sensore) che, come per la Base Station, l'utente deve posizionare con il mouse nella posizione corretta. Il pulsante 'Elimina mote' nel frame interno

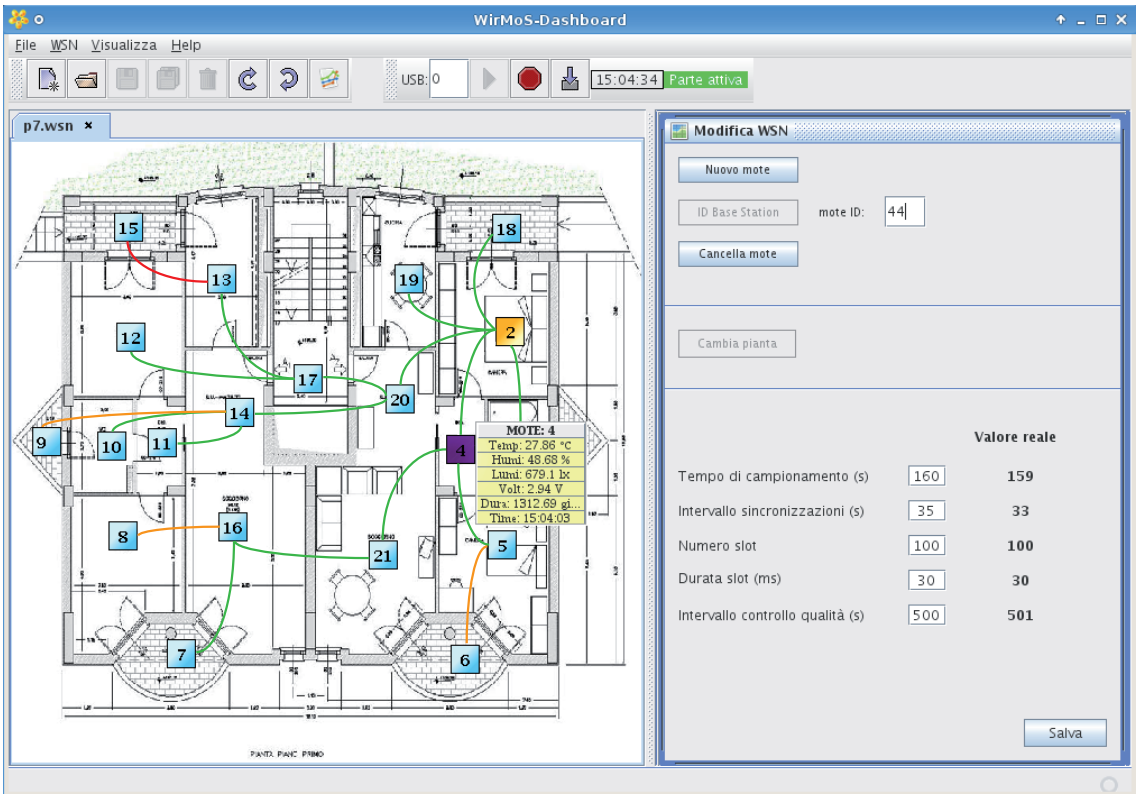


Figura 5.1: Mappa della WSN in WirMoS-Dashboard.

'Modifica WSN' permette di eliminare un nuovo sensore alla WSN corrente. Per eliminare un sensore si deve inserire l'identificativo del sensore nella test-box affiancata al pulsante. Il pulsante 'Cambia ID BS' nel frame interno 'Modifica WSN' permette di modificare l'identificativo della Base Station alla WSN corrente. Per cambiare l'identificativo si deve inserire il nuovo identificativo nella test-box affiancata al pulsante. Il pulsante 'Cambia mappa' nel frame interno 'Modifica WSN' permette cambiare la mappa alla WSN corrente. Ogni volta che si effettua una modifica sulla WSN, per memorizzare le modifiche si deve effettuare un salvataggio della WSN utilizzando 'Salva WSN' nel menù WSN oppure cliccando sull'icona con il disco nella barra degli strumenti. La Figura 5.1 mostra uno screenshoot dell'applicazione con visualizzata una pianta di una WSN in connessione.

Per modificare la posizione del sensore basta cliccare sopra il suo simbolo con il tasto sinistro del mouse, quindi mantenendo premuto il tasto spostare il simbolo sulla posizione desiderata, e infine rilasciare il tasto (drag-and-drop). Quando il puntatore del mouse si trova sopra un simbolo di sensore, questo si colora di rosso ad

indicare che è possibile effettuare il trascinamento. Per rendere i simboli dei sensori interattivi ognuno di essi viene realizzato attraverso un'istanza della classe *MotePanel* che estende la classe *javax.swing.jPanel*. In questo modo è possibile estendere la *classe java.awt.event.MouseAdapter* e *java.awt.event.MouseMotionAdapter*. Con la prima gestiamo gli eventi che vengono segnalati quando il puntatore del mouse entra o esce all'interno dell'area occupata da un simbolo. Con la seconda gestiamo gli eventi, che vengono segnalati quando viene premuto il tasto sinistro del mouse sopra l'area occupata dal simbolo, e mantenendo premuto il tasto viene il spostato puntatore del mouse (drag), e gli eventi che vengono segnalati quando viene rilasciato il tasto (drop). Le due classi vengono inserite all'interno della classe *WirMoS-DashboardPrjView* (come classi interne), in questo modo è possibile utilizzare all'interno dei gestori degli eventi, le variabili d'istanza della classe principale dell'applicazione. Quando viene aggiunto un sensore sulla pianta, viene creato un oggetto di tipo *Mote* e un oggetto di tipo *MotePanel*, e su quest'ultimo si effettua la registrazione dei gestori degli eventi creati attraverso i metodi *addMouseListener()* e *addMouseMotionListener()*. Viene inoltre creata all'interno della cartella della WSN, una nuova cartella che ha per nome l'identificativo del nuovo sensore; questa cartella conterrà i file che memorizzano i dati che il sensore invierà alla workstation. La creazione di una nuova cartella in Java si effettua con il metodo *mkdir()* della classe *java.IO.File*.

5.1.3 Comunicazione con i sensori

5.1.3.1 Connessione Workstation-Mote

TinyOS mette a disposizione delle classi Java per permettere ad un mote di comunicare con un'applicazione Java e viceversa; di seguito viene brevemente illustrato uno dei modi per realizzare questa comunicazione.

L'applicazione MIG di TinyOS semplifica il lavoro dello sviluppatore, in quanto converte in automatico il formato binario dei messaggi ricevuti dall'applicazione Java, nelle strutture dati inserite dai mote nei messaggi al momento della trasmissione, e fa il contrario con i messaggi che l'applicazione Java trasmette ai mote. E' possibile invocare l'applicazione MIG, nel momento in cui si effettua la compilazione del codice nesC, che va installato sul mote che deve trasmettere dati all'applicazione

Java, o ricevere da essa. La chiamata al MIG[1] viene inserita nel MakeFile, che si deve trovare all'interno della cartella dell'applicazione nesC. Si deve effettuare una chiamata a MIG per ogni tipologia di messaggio che si vuole inviare o ricevere. E' necessario inoltre che il tipo AM [27], per ogni tipologia di messaggio, venga dichiarato nel file .h con il nome AM_[*nomeStruttura*], dove [*nomeStruttura*] è il nome della struttura dati che viene inserita nel messaggio, scritto tutto in maiuscolo e sostituendo gli spazi bianchi con l'underscore. Effettuando la compilazione viene così creato un file .java per ogni chiamata al MIG effettuata nel MakeFile. Queste classi vanno posizionate nella cartella dei sorgenti del progetto Java.

Una possibilità per fare in modo che una classe Java riceva i pacchetti trasmessi da un mote, è quella di implementare l'interfaccia *net.tiyos.message.MessageLister*. Per utilizzare le classi e le interfacce Java di supporto, offerte da TinyOS, si deve importare all'interno del progetto la libreria tinyos.jar che si trova in [*cartella TinyOS*]/support/sdk/java. Per effettuare questa operazione in NetBeans, si deve aprire sulla finestra Project il nodo relativo al progetto nel quale si vuole effettuare l'importazione, quindi cliccare con il tasto destro su 'Libraries' e selezionare la voce 'Add JAR/Folder'. A questo punto basta recuperare il file indicando il percorso della libreria, e premere 'Open'. All'interno della classe che implementa l'interfaccia MessageListener (classe listener), si deve creare un oggetto di tipo *net.tiyos.message.MoteIF*. Al costruttore di questa classe si deve passare una *sorgente di pacchetto*, che è l'astrazione base di TinyOS per la comunicazione Workstation-Mote. Una sorgente di pacchetto è un mezzo di comunicazione attraverso il quale un'applicazione può ricevere pacchetti da un mote, e spedire pacchetti a un mote. Esempi di sorgenti di pacchetto sono le porte seriali, i socket TCP, lo strumento SerialForwarder. Per comunicare con il Tmote Sky si può usare la sorgente di pacchetto seriale, anche se la comunicazione avviene via USB. Questo perché USB e seriale sono molto simili nelle funzionalità, anche se sono molto differenti in termini di cavi e connettori. Quando si deve connettere un'applicazione Java ad un mote si deve sempre specificare qual è la sorgente di pacchetto, assieme al numero della porta a cui è connesso il mote, e la velocità di trasmissione dei dati. Nei tools di TinyOS tutte queste informazioni vengono raggruppate in un'unica stringa come ad esempio: "serial@/dev/ttyUSB0:telosb". Questa stringa serve per la connessione attraverso la sorgente di pacchetto seriale, tra un'applicazione installata sulla workstation e un mote che si trova collegato sulla porta /dev/ttyUSB0 (Linux). La

[1] Si veda tutorial TinyOS [27].

sottostringa 'telosb', è una variabile d'ambiente che fornisce il valore della velocità di trasmissione dei dati per le piattaforme TelosB e Tmote Sky. Per creare la sorgente di pacchetto da passare al costruttore dell'oggetto MoteIF, si utilizza il metodo *net.tinyos.BuildSource.makePhoenix()* fornendo ad esso la stringa di connessione come primo parametro, e *net.tinyos.util.PrintStreamMessenger.err* come secondo parametro. Dopo aver creato l'oggetto MoteIF, si deve registrare ad esso ogni tipologia di messaggio che la classe listener vuole ricevere o inviare, attraverso il metodo *registerListener()*. Il metodo ha due parametri, il primo deve essere un'istanza della classe generata dal MIG, relativa al tipo di messaggio che si vuole registrare, mentre il secondo deve essere un riferimento alla classe listener. Come ultimo e più importante passo si deve definire il corpo del metodo *messageReceived()* sulla classe listener, che viene invocato quando l'applicazione Java riceve un pacchetto dal mote. L'implementazione di questo metodo per WirMoS-Dashboard viene descritta nel prossimo paragrafo. Per inviare un messaggio ad un mote invece, si deve utilizzare il metodo *send()* della classe MoteIF.

Nell'applicazione WirMoS-Dashboard è possibile gestire una comunicazione con i mote utilizzando i controlli che ci sono nel frame interno 'Comunicazione sensori' o nell'omonima barra degli strumenti. Per visualizzare il frame si deve selezionare la voce 'Comunicazione sensori' nel menu Visualizza. Per connettere WirMoS-Dashboard con un mote collegato ad una porta USB della workstation, che sarà la Base Station, si deve inserire il numero della porta nel campo USB, e premere il pulsante 'Connetti'. Con quest'operazione viene creato l'oggetto MoteIF, e vengono registrate le tipologie di messaggio. Da quel momento l'applicazione è pronta per ricevere o trasmettere dati, al mote connesso alla porta USB fornita. Premendo il pulsante 'Avvia WSN' viene inviato un messaggio alla Base Station, che avvia la procedura di inizializzazione della rete. Per disconnettere WirMoS-Dashboard dalla Base Station basta cliccare sul pulsante 'Disconetti'.

5.1.3.2 Ricezione dei dati

La classe listener di WirMoS-Dashboard è WirMoS-DashboardPrjView, che contiene quindi il metodo messageReceived(). Questo metodo si occupa della gestione della ricezione di tutti messaggi, inviati dal mote connesso alla porta USB. Descriviamo in questo paragrafo le operazioni che il metodo esegue, quando viene ricevuto un messaggio di tipo *DataMsg*, contenete i dati forniti dai sensori dei mote.

La prima operazione è la trasformazione del numero di ciclo FSP, inserito nel campo *cycle* del messaggio ricevuto, in una indicazione temporale da assegnare al messaggio stesso. Il primo messaggio di un ciclo che arriva alla workstation, è quello della Base Station che viene inviato nel primo slot. A questo messaggio, che ha latenza trascurabile, viene data l'indicazione temporale dell'orologio della workstation, al momento della ricezione. Inoltre viene assegnato alla variabile *actualCycle* il valore del campo *cycle* del messaggio. Questa variabile indica a WirMoS-Dashboard qual'è il ciclo attuale del sistema. Quando invece arriva un messaggio di un nodo non Base Station, si effettua la differenza tra la variabile *actualCycle*, e il valore del campo *cycle* del messaggio. Il valore ottenuto è zero se il messaggio inviato dal nodo arriva alla workstation nello stesso ciclo in cui è stato trasmesso, come accade per la Base Station, altrimenti è maggiore di zero. La differenza tra i due valori viene moltiplicata per T_c ms, ottenendo il valore di quanti ms prima dell'inizio del ciclo attuale, il messaggio arrivato è stato trasmesso. A questo punto è sufficiente effettuare la differenza tra l'indicazione temporale del ciclo corrente (assegnata al messaggio della Base Station), e il ritardo calcolato, per ottenere l'indicazione temporale per il messaggio ricevuto.

La seconda operazione che il metodo esegue, è quella di creare alla ricezione del primo messaggio della giornata di un particolare nodo, un nuovo file di estensione .his (history file) che ha per nome la data del giorno attuale. I file .his di un particolare nodo si trovano all'interno della cartella /wsn/[*nome WSN*]/[*id nodo*]. Ognuno di questi file contiene tutti i dati ricevuti in una giornata da un particolare nodo.

Per ogni giornata inoltre viene creato il file 'TQuality_[*data*].txt' nella cartella /wsn/[*nome WSN*]/TQuality. Su questo file vengono memorizzate le informazioni relative alla qualità delle trasmissioni. L'utente può scegliere l'intervallo di tempo sul quale effettuare l'analisi. L'analisi si ripete ciclicamente, ovvero quando termina un analisi riparte subito un'altra analisi. Nel momento in cui l'analisi termina viene memorizzato per ogni nodo il numero di messaggi non arrivati alla workstation nell'ultimo intervallo di analisi, e la percentuale di messaggi ricevuti dalla workstation nell'ultimo intervallo di analisi, rispetto al numero massimo di messaggi che la workstation può ricevere per un nodo, in un intervallo di analisi. Nel file troviamo quindi una lista di elementi, ognuno dei quali contiene l'orario nel quale l'analisi si è conclusa, e la lista dei nodi della rete con affianco il numero di messaggi persi e la percentuale di messaggi ricevuti nell'ultimo intervallo. Queste statistiche mostrano per ogni nodo quanti messaggi non sono stati ricevuti nella workstation, ma non

danno l'indicazione su che link il messaggio è andato perso. Per questo motivo ogni nodo periodicamente controlla il numero di Acknowledge ricevuti, e inserisce nel prossimo suo campione di rilevazione anche il numero di messaggi di Acknowledge non ricevuti. In questo modo la workstation può conoscere la qualità di ogni link in entrambe le sue direzioni.

L'ultima operazione di messageReceived() è quella di elaborare i dati ricevuti, visualizzarli sulla GUI e memorizzarli nel file .his . Viene fatta innanzitutto la conversione nelle unità di misura desiderate, dei valori di umidità, temperatura, luminosità, e tensione di alimentazione forniti dai vari sensori, e contenuti nei vari campi del messaggio ricevuto. A questo punto se qualche controllo è attivato, viene determinato quali attuatori attivare e quali disattivare, attraverso il metodo *controlActuator()* e inviato i comandi all'activator. La funzione di controllo di WirMoS-Dashboard verrà spiegata nel dettaglio nel prossimo capitolo. Successivamente viene calcolata la stima della durata delle batterie, attraverso l'Equazione (6.1) e il valore della tensione delle batterie rilevato. I dati vengono poi salvati nel file .his e sull'oggetto Mote relativo al mittente del messaggio. Infine viene ridisegnato il pannello che contiene la pianta con i sensori, mostrando se abilitato, i nuovi dati ricevuti.

5.1.4 Visualizzazione dei dati

In WirMoS-Dashboard esistono tre modi per visualizzare i dati inviati dei sensori. Il primo consiste nel mostrare i dati direttamente sulla pianta, nell'istante in cui vengono ricevuti. Accanto al simbolo del sensore viene visualizzato il valore di temperatura, umidità, luminosità, la tensione attuale delle batterie, la stima della durata delle batterie, e l'indicazione temporale di quando i dati sono stati generati. Inoltre vengono mostrati i link che formano la topologia ad albero della WSN, attraverso degli archi che collegano i simboli dei sensori. Gli archi vengono colorati visualizzati con colori diversi in base alla qualità dei link (rosso per qualità pessima, arancione per qualità media, verde per qualità ottima). Per questa modalità di visualizzazione, è possibile scegliere nel menu Visualizza una delle seguenti tre opzioni di visualizzazione:

- *Mostra dati tutti sensori*: i dati ricevuti dai sensori sono sempre visualizzati sulla mappa.
- *Mostra dati sensore*: vengono visualizzati solo i dati del sensore attualmente selezionato.

- *Nascondi dati sensore*: i dati ricevuti dai sensori non vengono mai visualizzati.

La prima opzione può essere utilizzata quando sulla mappa ci sono pochi sensori, e quindi è possibile visualizzare i dati di tutti sensori senza creare confusione sull'immagine. La seconda opzione si utilizza invece quando ci sono tanti sensori sulla mappa, e quindi è conveniente visualizzare solo i dati del sensore interessato. Infine la terza opzione è utile quando si deve spostare i simboli dei sensori sulla posizione corretta, ed è necessario avere la visuale completa della mappa. Qualsiasi sia l'opzione che si scelga, gli archi che rappresentano i link tra i sensori vengono sempre mostrati.

Il secondo modo per la visualizzazione dei dati, è creazione di grafici di temperatura, luminosità, e umidità dei dati memorizzati nei file .his. Per visualizzare un grafico si deve selezionare la voce 'Visualizza grafico' nel menu WSN, oppure selezionare il pulsante con l'icona del grafico nella toolbar principale. Dopo aver effettuato questa operazione sulla parte sinistra della GUI viene visualizzato il frame interno 'Visualizza Grafico'. Per visualizzare un grafico si deve anzitutto selezionare la WSN, a cui appartiene il sensore di cui vogliamo conoscere i dati. Nel frame interno un menu a tendina permette di scegliere una WSN, scegliendola in una lista che mostra tutte quelle contenute nella cartella `wsn`. Una volta selezionata la WSN, si deve scegliere un sensore tra quelli visualizzati dal menu a tendina 'Seleziona mote'. Il menu mostra solo i sensori appartenenti alla WSN selezionata. Infine si deve scegliere l'intervallo temporale dei dati di cui si vuole realizzare il grafico, impostando sempre attraverso gli appositi menù a tendina il giorno, l'istante iniziale e l'istante finale, e premere il pulsante 'View Graph'. Se nell'intervallo scelto ci sono dei dati, verrà creato sulla parte sinistra della GUI un nuovo tab contenente tre grafici xy, nei quali l'asse x è l'asse temporale per tutti i grafici, mentre l'asse y rappresenta rispettivamente i valori di temperatura, umidità e luminosità, Figura 5.2. Per realizzare i grafici è stata collegata al progetto la libreria open source JFreeChart [28]. JFreeChart è un libreria molto completa per realizzare grafici di vario tipo come grafici lineari, a torta, a barre, temporali etc.. . I grafici che vengono visualizzati da WirMoS-Dashboard sono dei *time series chart* , dei grafici temporali, ovvero grafici lineari, con i valori delle ascisse che sono date al posto dei numeri.

L'ultimo modo per visualizzare i dati dei sensori è accessibile a partire dal menù WSN selezionando la voce 'Leggi storia mote'. Cliccato su tale voce è possibile navigare tra le cartelle, a partire dalla cartella `wsn` per scegliere la WSN, il sensore e quindi il file .his di interesse. Dopo aver selezionato il file e premuto il pulsante 'Apri storia', sulla parte sinistra della GUI viene visualizzato un nuovo tab contenente una tabella con sei colonne. Questa tabella mostra i valori di temperatura, umidità, luminosità, tensione rimanente sulle batterie, stima durata batterie indicazione temporale di ogni campione, e il numero

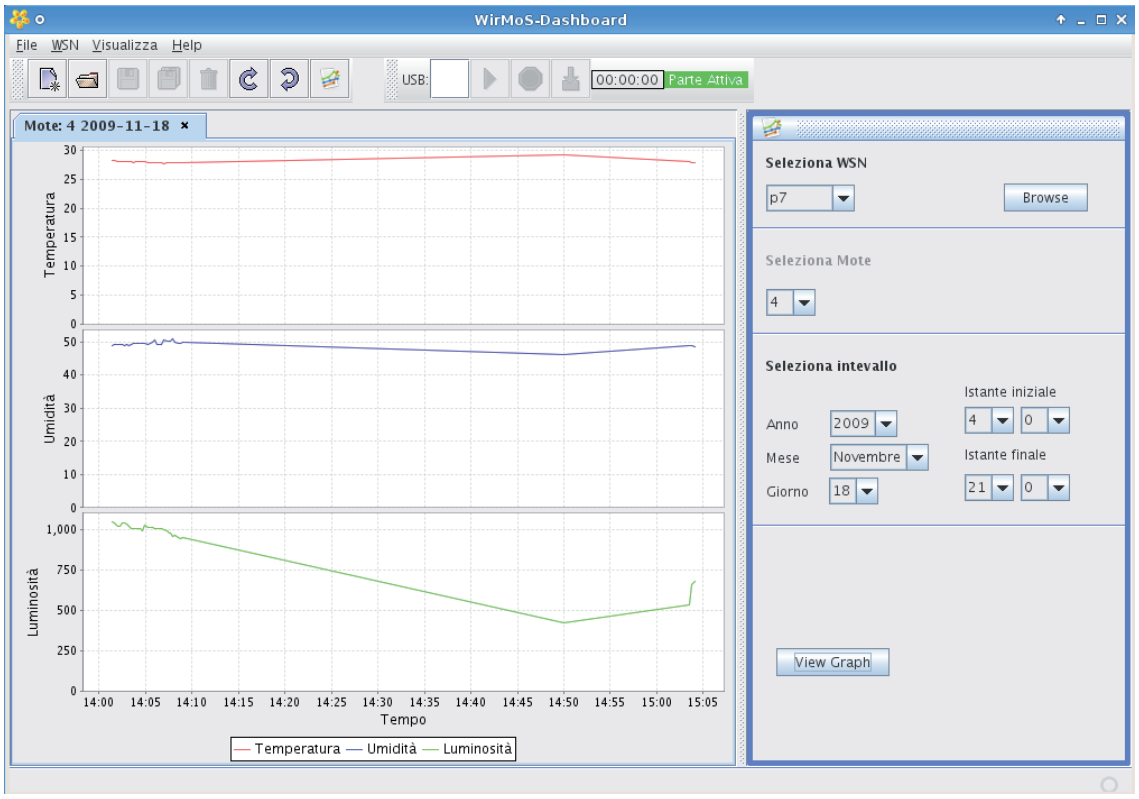


Figura 5.2: Grafici nello storico dati dei sensori in WirMoS-Dashboard.

di messaggi di acknowledge persi nell'ultimo intervallo di controllo, per tutti i campioni presenti nel file .his .

5.2 Lettura dati di logging

Per leggere le informazioni di logging memorizzate nella FLASH di un mote, durante la fase di logging, e quando viene trasmesso o ricevuto un messaggio (si veda paragarfo Paragrafo 3.2.5), si utilizza l'applicazione nesC WirmosReadNodeP.nc, che legge i dati dalla memoria e li invia alla workstation, e l'applicazione Java WirMoS-ReadNode, che riceve i dati, li elabora, e li memorizza su dei file.

5.2.1 WirmosReadNodeP.nc

Questa applicazione va installata sul mote quando si vuole conoscere i dati di logging. Per leggere i dati è importante non resettare il sensore con l'applicazione WirmosBStationP.nc o WirmosNodeP.nc installata sul sensore, in quanto durante l'avvio dell'applicazione la memoria viene cancellata. Per leggere la memoria, l'applicazione WirmosReadNodeP.nc utilizza il componente LogStorageC, lo stesso utilizzato per scrivere, in quanto implementa

anche l'interfaccia *LogRead*. Poiché sono stati creati due volumi della flash è necessario creare due istanze del componente, e un alias per ognuno. Si è scelto *LogReadMessage* per il volume Message e *LogReadData* per il volume Data. La prima operazione da fare per leggere i dati, è quella di posizionare per ogni volume, il puntatore di lettura all'inizio della memoria. Questo viene effettuato nel boot dell'applicazione utilizzando il comando *seek()* dell'interfaccia LogRead. La lettura inizia quando l'applicazione riceve dalla workstation un messaggio, che accende il led giallo del Tmote Sky e avvia il Timer0. Nel Timer0 vengono invocate le varie letture utilizzando il comando *read()* dell'interfaccia LogRead. Sia la scrittura con l'interfaccia LogWrite, che la lettura con l'interfaccia LogRead, si effettua per elemento. La lettura con LogRead è sequenziale, ovvero dopo che è stato letto un elemento, il puntatore di lettura si sposta automaticamente sul successivo elemento del volume. Ogni elemento è costituito da un particolare tipo di dato. Nel caso in esame sono state utilizzate *struct* di vario tipo definite nel file .h. Quando gli elementi vengono scritti nei due volumi, dalle applicazioni WirmosBStationP.nc e WirmosNodeP.nc, per ogni scrittura viene specificato il tipo di struct. Nel comando read() il primo parametro è un puntatore ad una zona di memoria RAM, dove posizionare i dati contenuti nell'elemento da leggere. Per riservare questa zona di memoria, si deve instanziare una struct dello stesso tipo della struct contenuta nell'elemento. E' necessario pertanto, quando si utilizza il comando read(), conoscere qual'è il tipo dato dell'elemento che si va a leggere. Come secondo parametro si deve passare la dimensione dell'area di memoria allocata, per far sapere quanti sono i byte da leggere. La lettura di un elemento si conclude con la segnalazione dell'evento *readDone*, che avrà un gestore per volume. Dopo la notifica dell'evento, è possibile recuperare i dati contenuti nell'elemento, accedendo ai campi dell'istanza della struct creata. L'evento inoltre restituisce il puntatore alla zona di memoria dove sono stati piazzati i dati, e il numero byte letti. Quest'ultimo valore è zero quando si tenta una lettura, dopo che sono stati letti tutti gli elementi del volume, e quindi può essere utilizzato per creare una condizione di fine lettura del volume. WirmosReadNodeP.nc legge prima tutti gli elementi memorizzati nel volume Message, e poi quelli nel volume Data (si veda Paragrafo 3.2.5). Subito dopo la lettura di un elemento, questo viene inviato all'applicazione WirMoS-ReadNode, e l'evento sendDone avvia la lettura successiva dopo 50 ms. Quando sono stati letti e inviati tutti gli elementi del volume Message, l'applicazione legge le varie struct contenute nel volume Data: i dati di ricezione, i dati di trasmissione, i dati generali (struct VarData), tutti i record della PingTable, tutti i record della AdjTable, e i vari pezzi dello schedule FPS. E' stato detto che è necessario sapere il tipo di dato dell'elemento che si va a leggere con il comando read(), per questo motivo l'applicazione deve conoscere quanti sono i record della PingTable, quanti sono i record della AdjTable e quanti sono i pezzi dello schedule, per sapere dove sono posizionati i vari elementi all'interno volume Data. Questi dati si trovano nella struttura VarData.

Come ultima operazione d'applicazione WirmosReadNodeP.nc invia alla workstation un messaggio di fine lettura dei dati, e accende il led rosso del Tmote Sky.

5.2.2 WirMoS-ReadNode

Il compito dell'applicazione WirMoS-ReadNode è quello di ricevere via USB i dati trasmessi dall'applicazione WirmosReadNodeP.nc, installata su un qualche sensore connesso ad una porta USB, di elaborarli, e di memorizzarli su dei file. L'applicazione è stata scritta, come per WirMoS-Dashboard, utilizzando un progetto di tipo *Java Desktop Application* di NetBeans, ma la GUI in questo caso è molto semplice. Quando l'applicazione viene avviata, si apre una finestra che contiene dei controlli per l'utente. Inserendo sulla casella di testo 'porta USB' il numero della porta USB, sulla quale è collegato il mote da leggere, e cliccando sul pulsante 'Connetti', viene creata una connessione con il mote nello stesso modo dell'applicazione WirMoS-Dashboard. Cliccando sul pulsante 'Avvia Lettura' viene inviato un messaggio all'applicazione WirmosReadNodeP.nc, che alla ricezione comincia ad estrarre i dati dalla FLASH e spedirli alla workstation. Alla ricezione del primo messaggio da parte WirMoS-ReadNode di ogni volume, viene creato un nuovo file di testo sulla cartella `SensorData`, contenuta nella cartella di installazione del programma. Il nome dei file è del tipo '[*idSensore*]_msg_[*dataCreazioneFile*]', per il file che contiene gli elementi del volume Message, e '[*idSensore*]_data_[*dataCreazioneFile*]' per il file che contiene quelli del volume Data. I file possono essere letti tramite un qualsiasi editor di testi.

Il primo file contiene una lista di informazioni sui messaggi trasmessi o ricevuti dal mote. I dati sono disposti in ordine cronologico. Nella prima parte del file, relativa all'inizializzazione della rete, ogni riga fornisce l'indicazione temporale in millisecondi del messaggio trasmesso/ricevuto, il TOS_NODE_ID del sensore mittente del messaggio (se questo valore è l'id del mote da cui provengono i dati, si tratta di un messaggio inviato, altrimenti ricevuto), e il tipo di messaggio trasmesso/ricevuto. L'indicazione temporale corrisponde all'intervallo di tempo trascorso dall'inizio della fase di Discovery, ovvero dall'invio del primo messaggio Discovery, fino alla trasmissione/ricezione del messaggio. Nella seconda parte del file, relativa alla fase di regime del sistema, oltre a questi dati è presente anche il numero di slot FPS nel quale il messaggio è stato trasmesso/ricevuto. Per i messaggi di questa parte del file, l'indicazione temporale corrisponde all'intervallo di tempo trascorso dall'inizio del ciclo, nel quale il messaggio è stato trasmesso/ricevuto, fino all'istante di ricezione/trasmissione. Ogni inizio di ciclo viene segnalato su file indicando il numero di ciclo, e il numero di slot di trasmissione e ricezione dello schedule in quel momento. Durante il ciclo vengono anche memorizzati gli eventi relativi allo spegnimento e l'accensione della radio. Ogni inizio dell'ultimo slot di un ciclo, viene segnalato sul file indicando il valore della domanda e dell'offerta in quel momento, e l'indicazione temporale.

Ad esempio per un nodo con ID 24 e T_s=30 mS:

```
*****************************************************************
CICLO: 80/1 numRT: 7 Time: 0
-----------------------------------------
Time: 230 Slot: 7 Type: RadioStart
Time: 246 Slot: 8 mote: 8 Type: Resync
Time: 271 Slot: 9 Type: RadioStop
-----------------------------------------
-----------------------------------------
Time: 351 Slot: 11 Type: RadioStart
Time: 367 Slot: 12 mote: 36 Type: Data
Time: 391 Slot: 13 Type: RadioStop
-----------------------------------------
-----------------------------------------
Time: 711 Slot: 23 Type: RadioStart
Time: 721 Slot: 24 *MOTE*: 24 Type: Resync
Time: 751 Slot: 25 Type: RadioStop
-----------------------------------------
-----------------------------------------
Time: 861 Slot: 28 Type: RadioStart
Time: 878 Slot: 29 mote: 32 Type: Data
Time: 901 Slot: 30 Type: RadioStop
-----------------------------------------
-----------------------------------------
Time: 1251 Slot: 41 Type: RadioStart
Time: 1268 Slot: 42 mote: 28 Type: Data
Time: 1291 Slot: 43 Type: RadioStop
-----------------------------------------
ULTIMO SLOT:  Demand: 4 Supply: 4 Time: 1471
*****************************************************************
```

Nel secondo file vengono forniti in primo luogo i dati di ricezione: il numero totale di messaggi ricevuti nella fase di inizializzazione, il numero di messaggi Discovery, Ping, Linking, Acknowledge, e Routing ricevuti. L'informazione successiva è l'indicazione del numero di messaggi del volume Message, persi perché al momento della memorizzazione, la memoria FLASH era già impegnata a memorizzare un messaggio precedente, come

spiegato nel Paragrafo 3.2.5 . Vengono poi presentate le informazioni di trasmissione: il numero totale di messaggi trasmessi nella fase di inizializzazione, il numero di messaggi Linking, Acknowledge, e Routing trasmessi. Proseguendo nel file troviamo nell'ordine, la lunghezza della PingTable e della AdjTable, il genitore del nodo nella topologia ad albero, il numero di hop che distano dalla Base Station, i record della PingTable e della AdjTable, e lo schedule FPS. Per ogni slot dello schedule viene indicato lo stato del nodo: idle, trasmit, receive, etc. . L'ultima sezione è un conteggio dei dati ricevuti dall'applicazione WirMoS-ReadNode. Viene segnalato innanzitutto il numero di messaggi totali ricevuti dall'applicazione, e successivamente il numero di messaggi del volume Messagge ricevuti, suddivisi per tipo.

Capitolo 6

Controllo attuatori e prestazioni

6.1 Controllo del modello di una serra

Il sistema WirMoS permette, dopo aver raccolto i dati provenienti dai sensori, di comandare degli attuatori per forzare le variabili misurate ad un valore desiderato. Il tipo di controllo dipende dalla specifica applicazione nella quale il sistema viene impiegato, ed è realizzato attraverso attuatori posizionati sul sito di interesse, e attraverso una funzione di controllo implementata all'interno di WirMos-Dashboard. E' utile utilizzare WirMoS stesso nella fase di progettazione del controllo, perché la flessibilità dei sensori wireless permette di testare al meglio tipo, e posizione degli attuatori, nonché la funzione di controllo di WirMos-Dashboard.

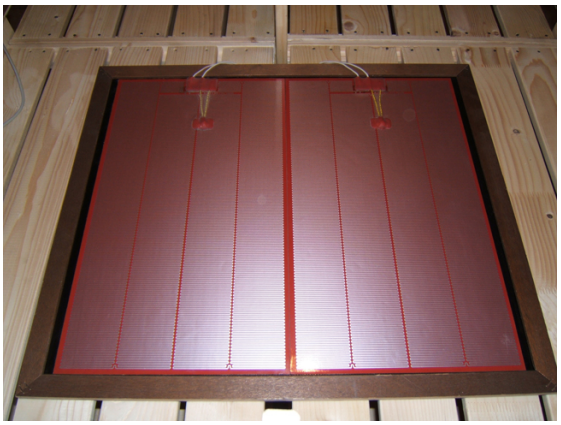
Il lavoro svolto comprende la realizzazione del controllo per un modello di una serra di larghezza 1.5 m, altezza 1.1 m, e profondità 1.0 m, visibile in Figura 6.1. L'interno del modello è costituito due piani sostenuti da montanti in legno. Il tetto, e due lati del modello, sono costituiti da lastre di plexiglas trasparente, allo scopo di far passare le radiazioni solari provenienti dall'esterno. Gli altri due lati sono costituiti da due pannelli di materiale isolante, e il fondo da una lastra di legno, allo scopo di migliorare il controllo della temperatura.

Per creare l'aumento della temperatura il modello utilizza due resistenze elettriche incollate su una lastra di vetro temperato, di dimensione 70 x 50 cm, Figura 6.2(a). Le resistenze sono alimentate a 220 V e assorbono una potenza di 400 W ciascuna. Per diminuire la temperatura viene utilizzata una coppia di ventole alimentate a 12 V, poste sulla parte superiore di un lato della serra, Figura 6.2(b). Le due ventole estraggono l'aria calda dalla serra per immetterla nell'ambiente esterno attraverso una serie di fori posizionati sotto le ventole. Questo sistema serve per mitigare un aumento eccessivo della temperatura dovuto all'effetto serra, abbassando la temperatura interna fino a raggiungere quell'esterna. Per aumentare l'umidità il modello utilizza l'atomizzazione ultrasonica. Questa tecnica consiste

Figura 6.1: Modello serra.

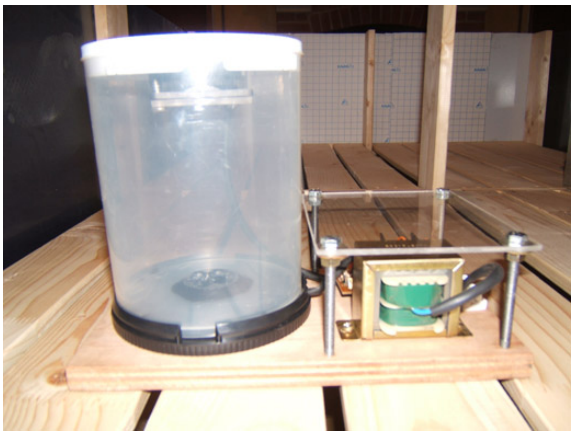
nel far oscillare ad una determinata frequenza un trasduttore piezoelettrico immerso nell'acqua, provocando la generazione di vapore. Il vantaggio dell'atomizzazione ultrasonica è che funziona a freddo, e quindi non va a modificare la temperatura interna della serra. Il modello utilizza un contenitore dove viene inserita l'acqua, alla base del quale è stato posto il trasduttore piezoelettrico. Questo trasduttore è dotato anche di un sensore di presenza di acqua, necessaria al raffreddamento del piezo. Poiché il vapore prodotto tende a ricadere sulla superficie dell'acqua, sul coperchio del contenitore è stata posta un'altra ventola alimentata a 12 V, che spinge l'aria all'interno del contenitore per far uscire il vapore dai fori praticati sul coperchio, Figura 6.2(c). Infine per controllare il livello di illuminazione il modello utilizza delle lampade alimentate a 12 V, e con assorbimento 20 W ciascuna, Figura 6.2(d).

Per rilevare il valore di temperatura, umidità, e livello di luce del modello si utilizza un nodo della rete, che chiamiamo *gHouseMote*. Per comandare gli attuatori viene impiegato invece un mote a parte, l'activator. La piattaforma Tmote Sky è dotata di un connettore di espansione che permette il collegamento a periferiche, attraverso un bus I^2C. I^2C è un sistema di comunicazione seriale bifilare a bassa velocità, utilizzato tra circuiti integrati, e inventato dalla Philips . Generalmente questo bus viene utilizzato per collegare periferiche a bassa velocità a schede madri, sistemi embedded, e cellulari. L'activator nel nostro caso viene collegato ad una scheda dotata dell'integrato PCF8574 e otto relè, Figura 6.3(a). L'integrato PCF8574 permette di pilotare gli otto relè utilizzando il protocollo I^2C, e i relè di comandare dispositivi a bassa tensione. Ventole e luci sono collegati direttamente a

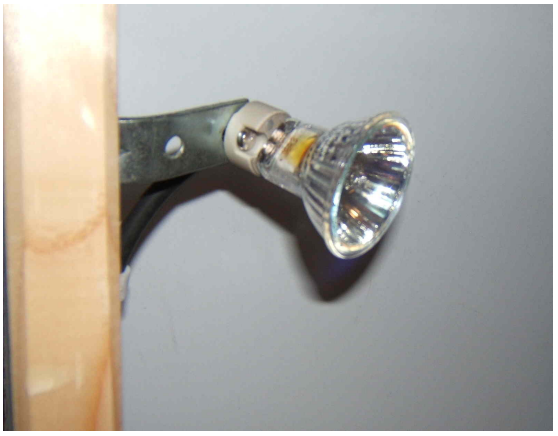


(a) Riscaldatore.

(b) Ventole raffreddamento.



(c) Umidificatore.

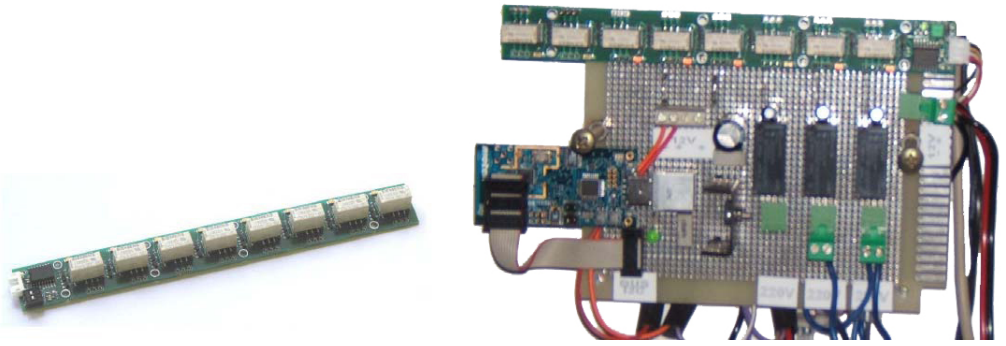


(d) Luci.

Figura 6.2: Attuatori modello serra.

questi relè. Il riscaldatore e l'umidificatore però sono alimentati a 220 V, e di conseguenza per questi si utilizzano dei relè alimentati a 12 V, ma con contatti che supportano una tensione che può raggiungere i 220 V. Questi relè sono attivati dai relè sulla scheda con PCF8574. Activator, scheda con otto relè e PCF8574, e relè di potenza, vengono connessi assieme utilizzando la scheda visibile in Figura 6.3(b).

Per attivare il controllo della serra si deve cliccare sulla voce 'Modello Serra', nel menu Visualizza, di WirMoS-Dashboard, quando è attiva una connessione ad una WSN. Questa operazione visualizza sulla parte destra della GUI il frame 'Controlla Serra'. Su questo frame è possibile selezionare quale mote della WSN è il gHouseMote, e qual'è la porta USB sulla quale è collegato l'activator. Una volta scelta la porta si deve cliccare sul pulsante 'Connetti', per creare una connessione con l'activator. Effettuata la connessione è possibile attivare i controlli di temperatura, umidità, e luce. Per ogni controllo è presente un pulsante per attivarlo o disattivarlo, una barra con cursore (*slider*) per impostare il valore



(a) Scheda 8 relè con PCF8574. (b) Scheda completa controllo attuatori.

Figura 6.3: Schede utilizzate per il controllo del modello della serra.

desiderato della variabile da controllare, e l'indicazione del valore impostato. Una volta attivato il controllo, è sufficiente spostare a destra o a sinistra il cursore, per scegliere il valore desiderato. Per disattivare un controllo basta cliccare nuovamente sul pulsante affianco allo slider. La Figura 6.4 mostra uno sreenshoot di WirMoS-Dashboard, con visualizzato il frame 'Controlla Serra'.

Quando almeno uno dei tre controlli e attivato, e viene ricevuto un messaggio dal gHouseMote, oltre alle normali operazioni descritte nel capitolo precedente, vengono effettuate delle operazioni aggiuntive, ovvero viene calcolata l'attuazione da trasmettere all'activator. L'attuazione consiste nell'informazione di quali attuatori devono essere attivi, e quali devono essere spenti.

Il calcolo dell'attuazione viene eseguito dal metodo *controlActuator()* della classe WirMoS-DashboardPrjView.java di WirMoS-Dashborad. Il metodo decide se attivare o disattivare il riscaldatore, e le ventole di raffreddamento, in base alla funzione mostrata in Figura 6.5. Quando la temperatura salendo supera $t_d + v$, dove t_d è la temperatura desiderata e v un valore di soglia, vengono attivate le ventole. Se la temperatura successivamente scende fino al valore t_d, le ventole vengono fermate. Quando la temperatura scende al di sotto della soglia $t_d - v$ viene attivato il riscaldatore. Se la temperatura successivamente sale fino al valore t_d, il riscaldatore viene spento. Per il controllo dell'umidità il metodo controlActuator() utilizza la funzione in Figura 6.6. Se l'umidità scende sotto il valore $h_d - z$, dove h_d è l'umidità desiderata e z un valore di soglia, viene attivato l'umidificatore. Se l'umidità successivamente sale fino al valore h_d, l'umidificatore viene spento. Per il controllo della luminosità il metodo controlActuator() utilizza la funzione in Figura 6.7. Se la luminosità scende sotto il valore l_d, ovvero il valore minimo di luminosità desiderato, vengono accese

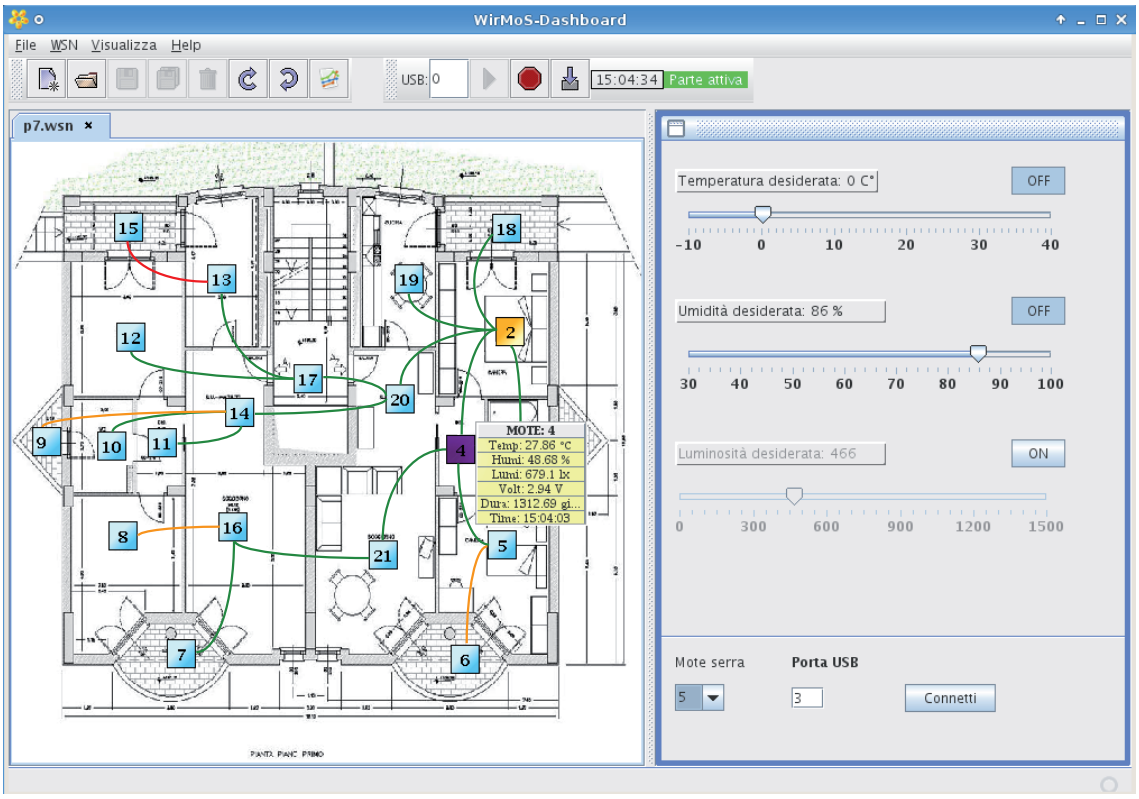


Figura 6.4: Controllo delle variabili monitorate con WirMoS-Dashboard.

le luci interne del modello. Quando la luminosità supera il valore $l_d + w$, dove w è un valore di soglia, le luci vengono spente.

Dopo aver calcolato l'attuazione richiesta, questa viene inserita in un messaggio di tipo *ActuationMsg*, e il messaggio inviato all'activator. L'activator è collegato ad una porta USB della workstation, e su di esso è installata l'applicazione WirmosActivatorP.nc. In alternativa l'activator può ricevere i comandi via radio, collegando alla workstation un'altro mote con installata l'applicazione WirmosBridgeP.nc. Quest'ultima semplicemente inoltra per l'activator sul canale radio i messaggi di tipo ActuationMsg inviati dalla workstation. WirmosActivatorP.nc comanda attraverso il bus I^2C l'integrato PCF8574 e quindi gli otto relè. Per far ciò l'applicazione utilizza il componente *Mps430I2CC* di TinyOS, che implementa l'interfaccia *I2CPacket*. Per pilotare i relè è sufficiente utilizzare il comando *write()* di tale interfaccia, fornendo il valore dell'attuazione. Il dato che il comando si aspetta di ricevere è un intero ad otto bit. Se l'n-esimo bit del valore dell'attenuazione convertito in binario è 0, l'n-esimo relè viene alimentato, se è 1 non viene alimentato. Questo vale per tutti gli otto bit, in quanto la scheda comanda otto relè.

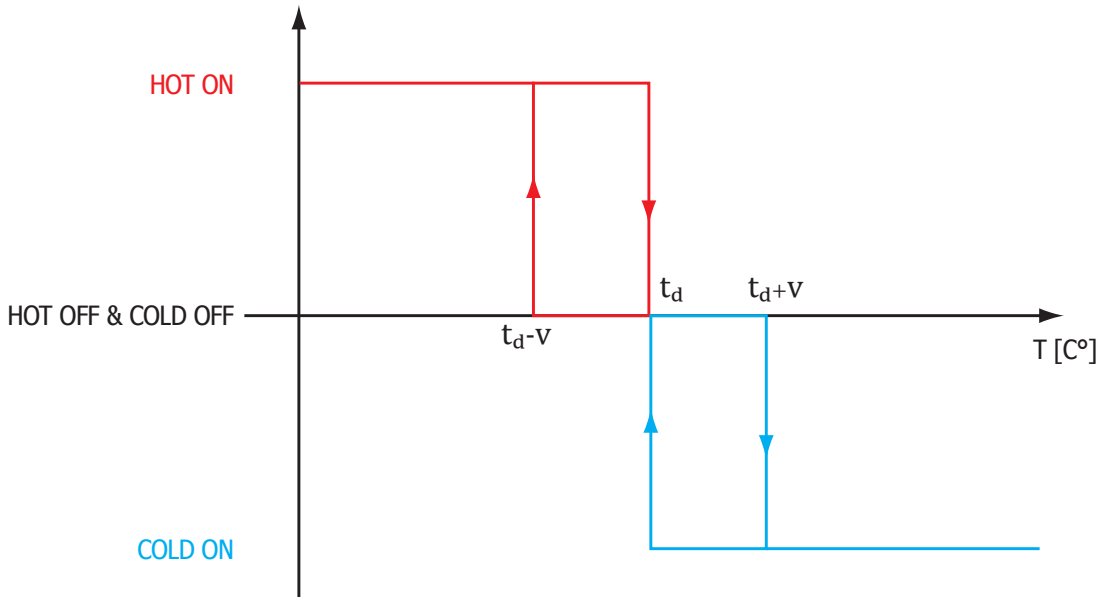


Figura 6.5: Funzione di controllo della temperatura.

6.2 Prestazioni e prove

In questa sezione descriviamo le prestazioni del sistema, tempi e consumi, attraverso prove e misurazioni effettuate sul campo, e considerazioni teoriche.

Un parte delicata dell'inizializzazione della rete è la fase di Pinging. In questa fase tutti i nodi inviano consecutivamente in broadcast un certo numero di messaggi di Ping (almeno 40). Il traffico che si viene a creare è elevato e sono probabili diverse collisioni. Queste non devono essere in numero tale da compromettere l'utilità della fase (verificare la qualità dei link che si vengono a formare tra i nodi), quindi per limitarle il protocollo di WirMoS utilizza in questa fase sia il meccanismo CSMA, sia distribuisce le trasmissioni in modo casuale in un intervallo di tempo sufficientemente lungo. La situazione è tanto più critica quanti più nodi si trovano sulla stessa zona di copertura radio. Per testare l'efficacia del meccanismo realizzato nella situazione più critica, sono state effettuate delle prove con una rete a stella di 10 nodi (base station con ID 19), posizionanti in modo tale che ogni sensore possa raggiungere con i suoi messaggi tutti gli altri sensori. Per ogni prova si è costruito una tabella come la seguente:

Nelle prove fatte i nodi inviano 40 ping, quindi se non venissero persi messaggi, ogni nodo dovrebbe ricevere un totale di 360 ping. Nella seconda colonna della tabella si riporta il numero di ping non arrivati su un totale di 360, e nell'ultima riga viene visualizzata la media. Il tempo di attesa tra una trasmissione e l'altra è compreso nell' intervallo [0 ms , 150 ms], mentre il tempo di attesa della trasmissione quando il CSMA trova il canale

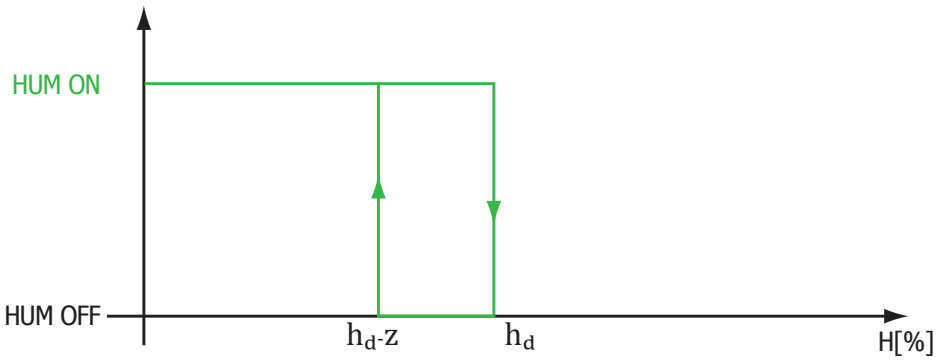


Figura 6.6: Funzione di controllo dell'umidità.

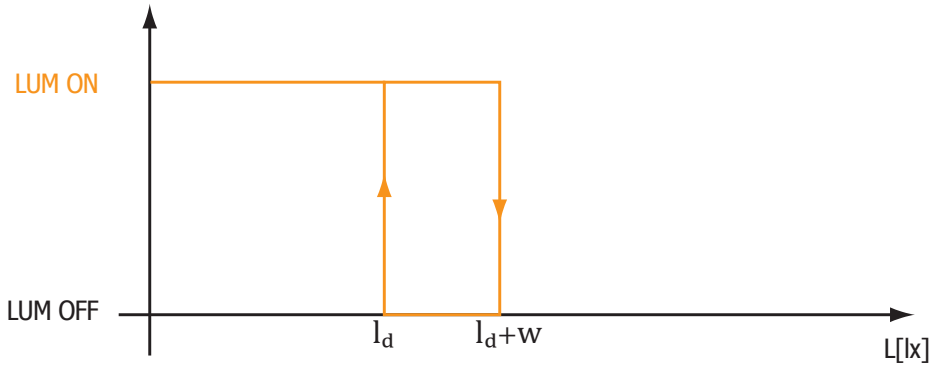


Figura 6.7: Funzione di controllo della luminosità.

occupato è di 7 ms. Dopo aver effettuato diverse prove è stata calcolata la media delle medie, ottenendo un valore medio di 28 messaggi persi su 360 (7,8%). La terza colonna riporta il numero di vicini affidabili che un nodo si ritrova ad avere una volta conclusa la fase di Linking, ovvero il numero di record della AdjTable. Come si può osservare dalla tabella, nonostante venga perso un certo numero di messaggi, ogni nodo è collegato da un link affidabile con tutti gli altri nodi. Questo accade perché il numero di messaggi trasmessi da un nodo, e ricevuti da un altro, è sempre maggiore o uguale di 32 su 40, che è la soglia impostata al di sotto della quale non si considera un link affidabile. Tale risultato si è verificato in ogni prova confermando la validità della fase, nella quale WirMoS riesce a rilevare tutti i link affidabili effettivamente presenti, anche nella situazione più critica. In ogni caso in una condizione del genere, essendo i sensori tutti nella stessa zona, per un nodo è indifferente quale altro nodo utilizzare come genitore, quindi anche se non vengono individuati proprio tutti i link affidabili non è comunque un fatto grave.

Nel Paragrafo 4.6 è stato detto che per mantenere minima la latenza è necessario

Mote ID	N° ping non arrivati	N° vicini
15	30	9
16	30	9
17	31	9
18	29	9
20	29	9
21	29	9
22	26	9
23	28	9
24	29	9
media	*29*	*9*

Tabella 6.1: Esempio di dati raccolti nel test della fase di Pinging.

minimizzare il numero di slot ponendolo a $N_{slot} = 3N_n$, e che però per far in modo che la fase delle prenotazioni non duri troppo è conveniente porre $N_{slot} = 4N_n$. Quest'ultima considerazione è utile nel caso in cui ci sono nodi figli, che hanno circa lo stesso numero di discendenti del nodo figlio che ha il massimo numero di discendenti. Infatti questi sono nodi che hanno uno schedule con molti slot di tipo R e T, e i loro genitori, durante una prenotazione, hanno una probabilità di scelta di uno slot che è I anche per un figlio, che è la più bassa tra tutti i nodi.

Per provare che utilizzare uno schedule con un numero di slot pari a $4N_n$ è una scelta adeguata, sono state effettuate delle prove con una rete la cui topologia è quella di Figura 6.8.

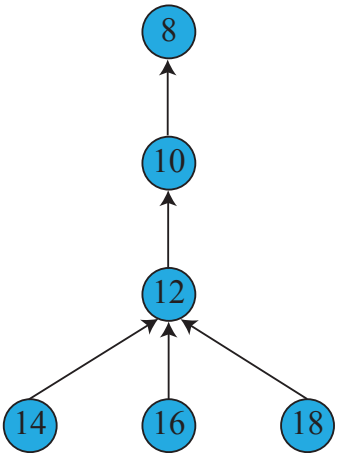


Figura 6.8: Rete utilizzata per testare il vincolo sul numero minimo di slot dello schedule.

Il nodo 10 è il nodo figlio che ha il massimo numero di discendenti $N_d(10) = 4$, e il nodo 11 ha circa lo stesso numero di discendenti $N_d(10) = 3$. Il numero di nodi è 6, quindi

$4N_n = 24$, ma in realtà si pone a 26 il numero di slot dello schedule perché il primo e l'ultimo slot sono di servizio, e devono essere sempre I. Per le prove sono state effettuate le seguenti ulteriori impostazioni: $T_s = 30$ ms, quindi $T_c = T_s N_{slot} = 0,78$ secondi, $\alpha = 20$ quindi $T_{sc} = \alpha T_c = 15,6$ secondi. Sono state effettuate diverse prove con questi valori registrando ad ogni prova la durata della fase delle prenotazione, e alla fine è stata calcolata la media. Il risultato medio ottenuto è di 30 cicli FPS, ovvero una durata di 23,4 secondi. Altre prove sono state effettuate mantenendo le stesse impostazioni, ma utilizzando un schedule di 150 slot. Anche in questo caso la durata media della fase delle prenotazioni è stata di 30 cicli, dimostrano che utilizzare uno schedule di $4N_n$ slot non solo è sufficiente per portare a termine la fase delle prenotazioni, ma anche non la rallenta.

6.2.1 Consumi

Nell'algoritmo FPS originale, ogni nodo sceglie ad ogni ciclo FPS due slot I, e li definisce come A e RP. In questi due slot il nodo tiene la radio accesa, e nello slot A invia un messaggio Advertisement. Queste operazioni servono per permettere la variazione della domanda e dell'offerta dei nodi, a causa del cambiamento della topologia della rete, ovvero la modifica del genitore da parte di un nodo se decade la qualità del link, o l'aggiunta di nodi. In molte applicazioni però le variazioni avvengono di rado, e si è deciso quindi di abilitare l'invio degli Advertisement, e quindi tenere la radio accesa negli slot A e RB, solo quando avviene la variazione, per limitare ulteriormente i consumi. Anche in questo caso all'inizio della fase di regime, la trasmissione dei messaggi di Advertisement è abilità, per permettere l'esecuzione delle prenotazioni FPS. Quando la fase delle prenotazioni finisce e viene ricevuto il ChangeMsg, l'invio degli Advertisement viene sospeso, e quindi non vengono più creati slot di tipo A, RP, o TP.

Fatte queste considerazioni possiamo valutare il consumo medio di un nodo. Per far ciò si deve individuare un intervallo di tempo trascorso il quale gli eventi si ripetono allo stesso modo. Indichiamo con γ il numero di cicli che compongono questo intervallo. $\alpha = T_{sc}/T_c$ è il numero di cicli in un superciclo. Definiamo inoltre le seguenti sigle:

β: Ogni β cicli viene effettuata la sincronizzazione
$\beta = \lfloor \frac{T_{re}}{T_c} \rfloor$

γ: Ogni γ cicli gli eventi si ripetono
$\gamma = m.c.m(\alpha, \beta)$ minimo comune multiplo tra α e β

$N_{RT}(j)$: Numero di slot R e T nello schedule del un nodo j
$N_{RT}(j) = N_R + N_T$

$n_{rt}(j)$: Numero di slot R o T del nodo j, con radio accesa, in γ cicli

$n_{rt}(j) = N_{RT}(j)\frac{\gamma}{\alpha}$

n_{re}: Numero di slot usati per la sincronizzazione in γ cicli
$n_{re} = 2\frac{\gamma}{\beta}$

Δ: Numero di slot in γ cicli
$\Delta = \gamma N_{slot}$

La Tabella 6.2, tratta dal datasheet del Tmote Sky, mostra le condizioni operative tipiche della piattaforma. Quello che ci interessa ora sono i consumi, che dipendono dallo stato in cui il mote si trova. Verifiche sperimentali su un sensore Tmote Sky versione certificata FCC, hanno confermato i valori. In particolare si è riscontrato un consumo con la radio accesa di 19 mA in trasmissione (a potenza massima, valore 31), 19.2 mA in ricezione, 18.8 mA con radio nello stato idle e 6 µA con la radio spenta. Ora abbiamo tutti gli elementi necessari per calcolare il consumo medio (Average Consumption) a regime di un nodo j in un superciclo, attraverso l'Equazione (6.1).

$$AC(j) = \frac{19T_s(n_{rt}(j) + n_{re}(j)) + 6 \cdot 10^{-3}T_s(\Delta - n_{rt}(j) - n_{re}(j))}{\Delta \cdot T_s} \quad (6.1)$$

Nell'equazione si assume 6 µA il cosumo medio del sensore con la radio spenta, e 19 mA con la radio accesa. Con le stesse considerazioni è possibile determinare la formula del duty cycle di un nodo j, ovvero la frazione di tempo che la radio è accesa rispetto al tempo totale:

$$\delta(j) = \frac{n_{rt}(j) + n_{re}(j)}{\Delta} \quad (6.2)$$

Se utilizziamo i valori minimi di T_s e N_{slot}, che sono 15 ms e $4N_n$ rispettivamente, dove N_n è il numero massimo di nodi della rete, otteniamo la minima latenza. Se aumentiamo

	MIN	NOM	MAX	UNIT
Supply voltage	2.1		3.6	V
Supply voltage during flash memory programming	2.7		3.6	V
Operating free air temperature	-40		85	°C
Current Consumption: MCU on, Radio RX		21.8	23	mA
Current Consumption: MCU on, Radio TX		19.5	21	mA
Current Consumption: MCU on, Radio off		1800	2400	µA
Current Consumption: MCU idle, Radio off		54.5	1200	µA
Current Consumption: MCU standby		5.1	21.0	µA

Tabella 6.2: Condizioni tipiche di funzionamento piattaforma Tmote Sky.

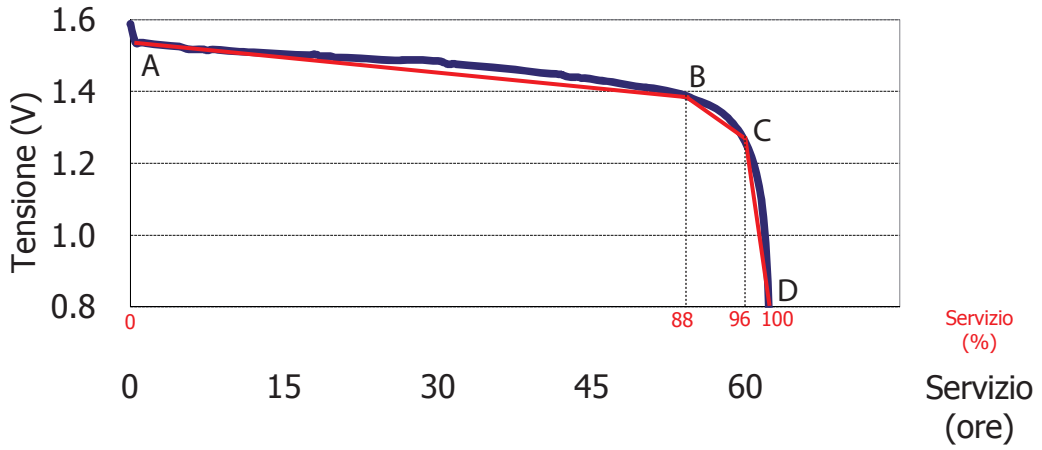


Figura 6.9: Curva di scarica delle batterie Energizer L91.

N_{slot}, per aumentare la latenza, con i supercicli aumenta solo T_c e quindi diminuisce α, ma il tempo in cui la radio rimane accesa nel superciclo e la durata di questo, rimangono invariati. Pertanto il consumo medio non varia. Inoltre mentre nell'FPS di base aumentare T_s non ha effetti sul consumo medio in un ciclo, in quanto aumenta anche la sua durata in proporzione, aumentare T_s, e quindi la latenza con i supercicli ha come risultato l'aumento dei consumi. Questo è dovuto al fatto che aumenta il tempo in cui la radio resta accesa in un superciclo, ma la sua durata rimane invariata. Per questi motivi è meglio che N_{slot}, T_s e quindi la latenza abbiano il minor valore possibile, rispettando però i vincoli descritti.

Conoscendo il consumo medio di un nodo, e la capacità delle batterie che lo alimentano, è possibile fornire una stima del tempo di vita. Per il sistema WirMos si è deciso di utilizzare le batterie al litio Energizer L91 [26], che hanno una capacità di 3000 mAh. La curva blu di Figura 6.9 è la curva di scarica di queste batterie, tratta dal data sheet. Dalla figura si osserva che se la tensione di una batteria di questo tipo è di 1V, la batteria è praticamente scarica. Poiché le due batterie che alimentano il mote sono in serie, quando la tensione è circa 2V la piattaforma smette di funzionare. Fortunatamente la tensione minima di alimentazione di un Tmote Sky è 1.8V, pertanto questo mote è in grado di sfruttare tutta la capacità delle batterie. Altre piattaforme invece, hanno bisogno di almeno 2.7 V, quindi di 1.35 V per batteria, riuscendo così a sfruttare solo una frazione della carica totale. Poiché un nodo della rete j viene alimentato da batterie con capacità 3000 mAh, la sua durata in giorni è $3000/(24 \cdot AC(j))$.

Per informare l'utente sulla durata rimanente delle batterie di un nodo, si trasmette insieme alle letture dei vari sensori ambientali, anche l'informazione della tensione interna del mote, utilizzando il componente *Msp430InternalVoltageC*. Il valore del sensore di tensione dopo esser stato diviso per 4096 e moltiplicato per 3, fornisce la tensione di alimentazione

del mote in Volt, e quindi la tensione fornita dalle batterie. Per ricavare la percentuale di tempo di vita di un mote già speso, si utilizza la curva rossa di Figura 6.9, che è un'approssimazione della curva di scarica reale delle batterie Energizer L91 (curva blu). La curva si riferisce ad una corrente assorbita costante di 50 mA, ma variando l'intensità di corrente la capacità della batteria e la curva rimangono simili. La curva è composta da segmenti i cui estremi sono i punti del piano cartesiano A=(0%,1.55V), B=(88%,1.38V), C=(96%,1.27V) e D=(100%,0V). Il primo valore di ogni punto è la percentuale di tempo di vita già speso p, mentre il secondo è la tensione corrispondente v. Utilizzando questi punti si ricava l'equazione per calcolare p dato v, che corrisponde quindi al grafico inverso di quello visualizzato in figura 6.9:

$$p(v) = \begin{cases} -\frac{400}{127}v + 100, & \text{per } 0 \leq v \leq 1.27 \\ -\frac{800}{11}v + \frac{2072}{11}, & \text{per } 1.27 \leq v \leq 1.38 \\ -\frac{8800}{17}v + \frac{13640}{17}, & \text{per } 1.38 \leq v \leq 1.55 \end{cases}$$

Poiché il Tmote Sky utilizza due batterie in serie Wirmos-DashBoard deve utilizzare le seguenti equazioni:

$$p(v) = \begin{cases} -\frac{200}{127}v + 100, & \text{per } 0 \leq v \leq 2.54 \\ -\frac{400}{11}v + \frac{2072}{11}, & \text{per } 2.54 \leq v \leq 2.76 \\ -\frac{4400}{17}v + \frac{13640}{17}, & \text{per } 2.76 \leq v \leq 3.1 \end{cases}$$

Dopo aver ricavato $p(v)$, si può calcolare $L_t(j)$ (Life Time), la stima del tempo di vita rimanente del nodo j in giorni, utilizzando l'equazione 6.3.

$$L_t(j) = \frac{3000}{24AC(j)} - \frac{3000p(v)}{2400AC(j)} \tag{6.3}$$

Nel sistema realizzato l'utente ha possibilità di scegliere in qualsiasi momento la frequenza di campionamento che desidera. Il comando viene impartito nell'applicazione WirMos-Dashboard, la quale calcola T_{sc} e α, e trasmette alla Base Station un messaggio di tipo ChangeMsg con il campo *alpha* impostato al valore trovato. La Base Station inoltra il messaggio ai nodi a profondità uno. Quando un nodo non Base Station riceve nello slot RB un messaggio di questo tipo, modifica la variabile ALPHA, cambiando quindi il tempo di campionamento, e inoltra il messaggio nel prossimo slot B. Dopo un certo tempo il broadcast verrà completato, e quindi tutti i nodi si comporteranno allo stesso

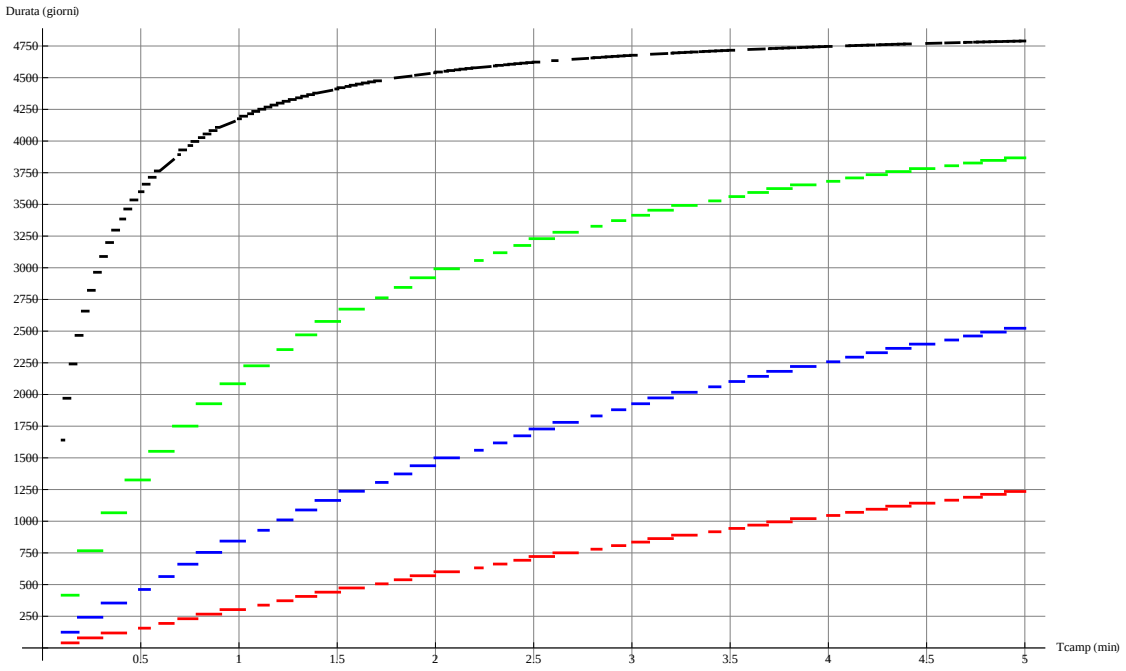


Figura 6.10: Durata nodi al variare del tempo di campionamento, di una con rete con topologia ad albero ternario completo, di altezza H=4 (121 nodi). La curva rossa indica la durata dei nodi a profondità 1, la curva blu dei nodi a profondità 2, la curva verde dei nodi a profondità 3, e la curva nera dei nodi a profondità 4.

modo. L'utente dopo aver modificato il tempo di campionamento della rete, può valutare il cambiamento nella stima della durata di funzionamento di ogni nodo, e decidere in base a tali valori se mantenere le nuove impostazioni, o modificarle ulteriormente.

La Figura 6.10 mostra il valore della durata in giorni di un nodo, al variare del tempo di campionamento in minuti impostato dall'utente, utilizzando l'Equazione (6.1), T_s=15 ms, T_{re}=30 s, e capacità delle batterie 3000 mAh. Le quattro curve si riferiscono ad una rete con topologia ad albero ternario completo di altezza 4 (121 nodi). Ogni curva mostra la durata di un nodo ad una profondità diversa. Le curve sono discontinue in quanto il tempo di campionamento effettivo deve essere un multiplo di T_c. La Figura 6.11 mostra invece le curve relative ad una rete con topologia ad albero binario completo di altezza 3 (31 nodi), utilizzando gli stessi parametri del caso precedente.

Per testare la validità della formula 6.1, che fornisce il consumo medio di corrente di un nodo della rete, è stata effettuata una misurazione dell'assorbimento di corrente alimentando un Tmote Sky con un source meter Keithley 2612. Lo scopo della misura è stato quello di rilevare l'eventuale neccesità di modificare la formula, per adattarla meglio alla realtà, e in particolare verificare il peso dei picchi di corrente che si verificano all'accensione della radio, sul calcolo del consumo medio. Durante la misurazione sul mote era in esecuzione l'applicazione WirmosNode.nc a regime, con T_s=30 ms. In Figura 6.12 si può

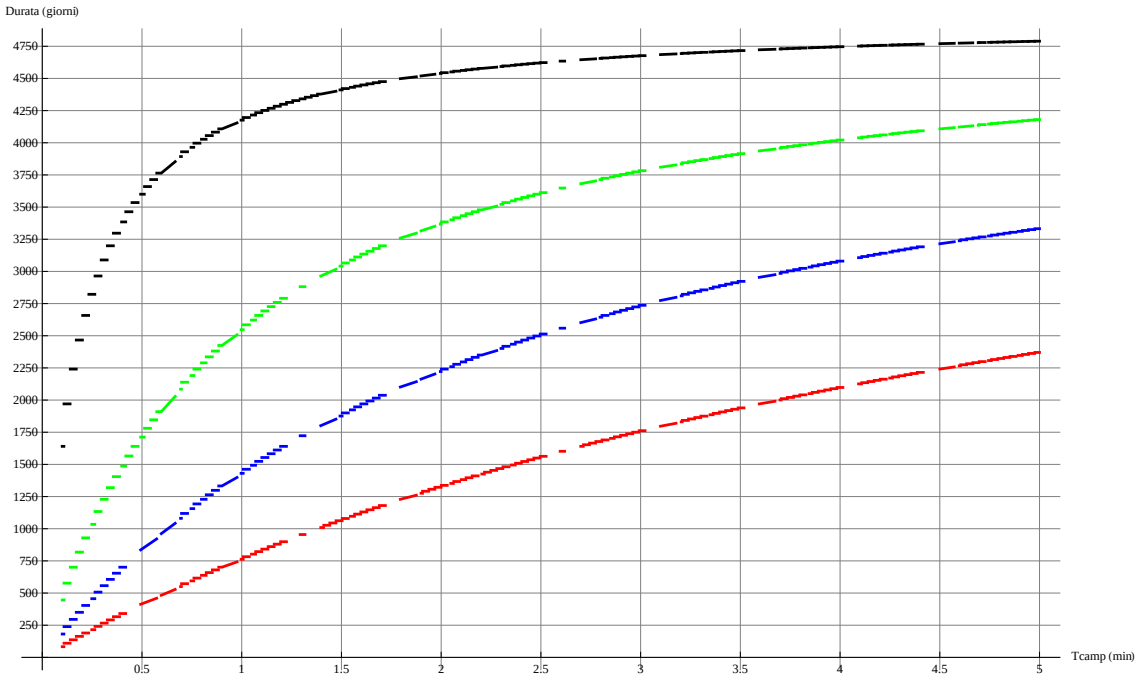


Figura 6.11: Durata nodi al variare del tempo di campionamento, di una con rete con topologia ad albero binario completo, di altezza H=4 (31 nodi). La curva rossa indica la durata dei nodi a profondità 1, la curva blu dei nodi a profondità 2, la curva verde dei nodi a profondità 3, e la curva nera dei nodi a profondità 4.

vedere un grafico relativo ad una parte dell'intervallo di tempo utilizzato per la misura. Il grafico mostra l'assorbimento di corrente della piattaforma al variare del tempo. Il segnale misurato è un onda quadra con un livello alto di circa 19 mA (radio accesa), e livello basso di circa 6 µA (radio spenta). Quest'ultimo non è visibile nel grafico in quanto ha scala in mA, e non viene nemmeno misurato dallo strumento che rileva a radio spenta solo rumore. Negli intervalli in cui la radio è accesa è indicato anche il numero di slot. La Figura 6.12 mostra un particolare del grafico relativo allo slot 13. All'inizio dello slot, cosi come prima del successivo, c'è una caduta del segnale di corrente provocata probabilmente dalla resicnronizzazione dell'hardware dopo un cambio del canale sul quale trasmettere/ricevere. La misurazione effettuata dimostra che se si trascura il tempo di salita e di discesa dell'onda quadra, e i picchi di corrente che si presentano all'accensione della radio, non si introduce un errore significativo nel calcolo dell'assorbimento medio di corrente di un nodo. Pertanto si ritiene che la formula 6.1 possa essere considerata sufficientemente attendibile.

6.2.2 Test sul controllo del modello serra

Per testare il controllo della temperatura del modello della serra, attraverso WirMoS-Dashboard è stata impostata una temperatura desiderata di 29 ℃, e una finestra v di 1

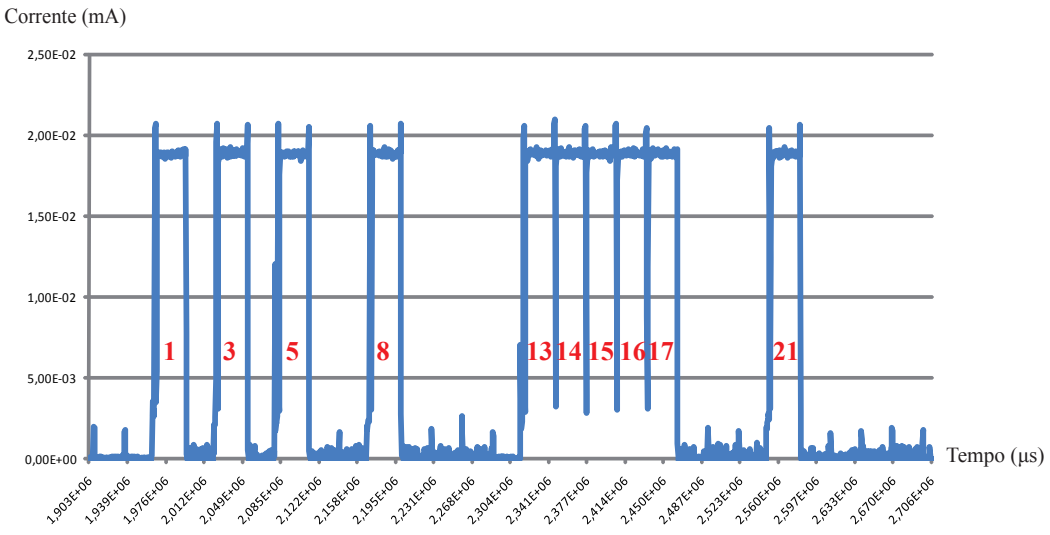


Figura 6.12: Assorbimento di corrente di WirmoNode.nc a regime, al variare del tempo.

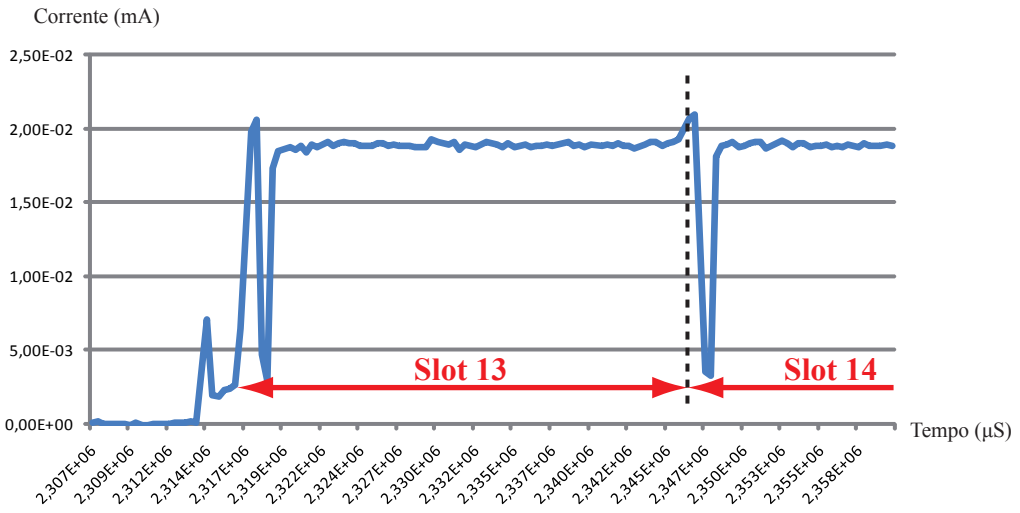


Figura 6.13: Particolare del grafico in Figura 6.12.

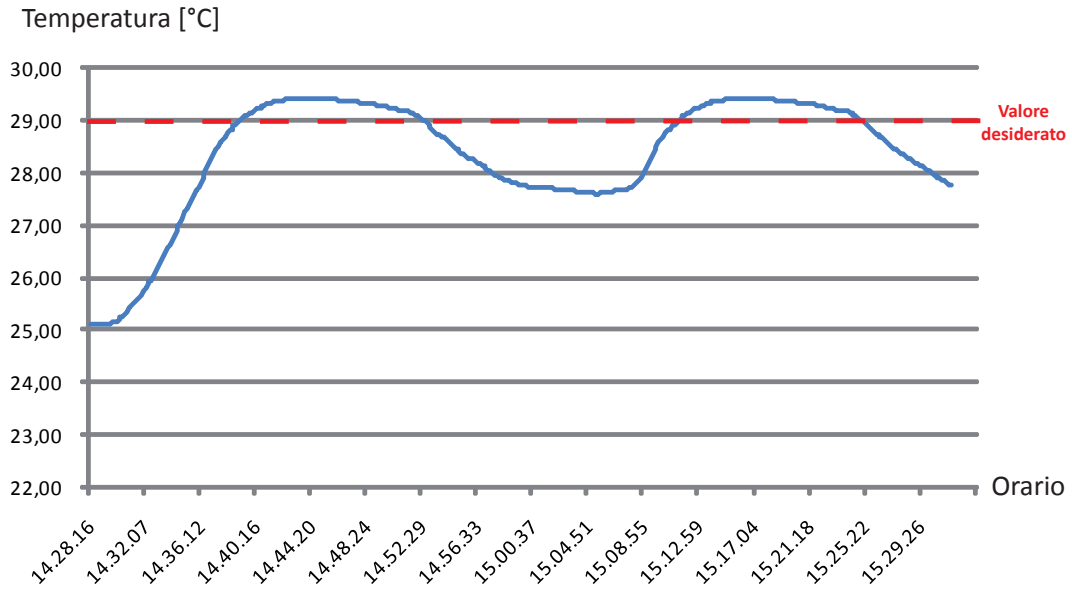


Figura 6.14: Prova controllo temperatura.

°C. La Figura 6.14 mostra il valore della temperatura della serra al variare del tempo, nella prova effettuata. La temperatura iniziale, alle ore 14:18:26, è di 25 °C. In questo istante viene attivato il controllo di temperatura, e impostata una temperatura desiderata di 28 °C. Essendo la temperatura iniziale sotto la soglia $t_d - v$=28 °C, il riscaldatore entra subito in funzione facendo salire la temperatura interna. Da questo istante in poi il riscaldatore si spegne quando la temperatura supera i 29 °C, e si attiva quando scende sotto il 28 °C.

Per testare il controllo dell'umidità del modello della serra, attraverso WirMoS-Dashboard è stata impostata un' umidità desiderata del 60%, e una finestra z = 5%. La Figura 6.15 mostra il valore dell'umidità della serra al variare del tempo, nella prova effettuata. L'umidità iniziale, alle ore 14:09:49, è del 48%. In questo istante viene attivato il controllo di temperatura, e impostata una temperatura desiderata del 60%. Essendo l'umidità iniziale sotto la soglia $t_d - z$ =55 %, l'umidificatore entra subito in funzione facendo salire l'umidità interna. Da questo istante in poi l'umidificatore si spegne quando l'umidità supera il 60 %, e si attiva quando scende sotto il 55 %.

Per testare il controllo della luminosità del modello della serra, attraverso WirMoS-Dashboard è stata impostata una luminosità minima desiderata di 300 lx, e una finestra w = 100 lx. La Figura 6.16 mostra il valore della luminosità della serra al variare del tempo, nella prova effettuata. Nell'istante 14:30:20 viene eliminata ogni fonte di luce nell'ambiente che contiene il modello dello serra. Il sensore gHouseMote rileva una luminosità inferiore a quella minima, e quindi WirMoS-Dashboard attiva le luci interne, portando la luminosità ad un valore maggiore di quello minimo desiderato. Nell'istante 14:31:38 viene ripristinata

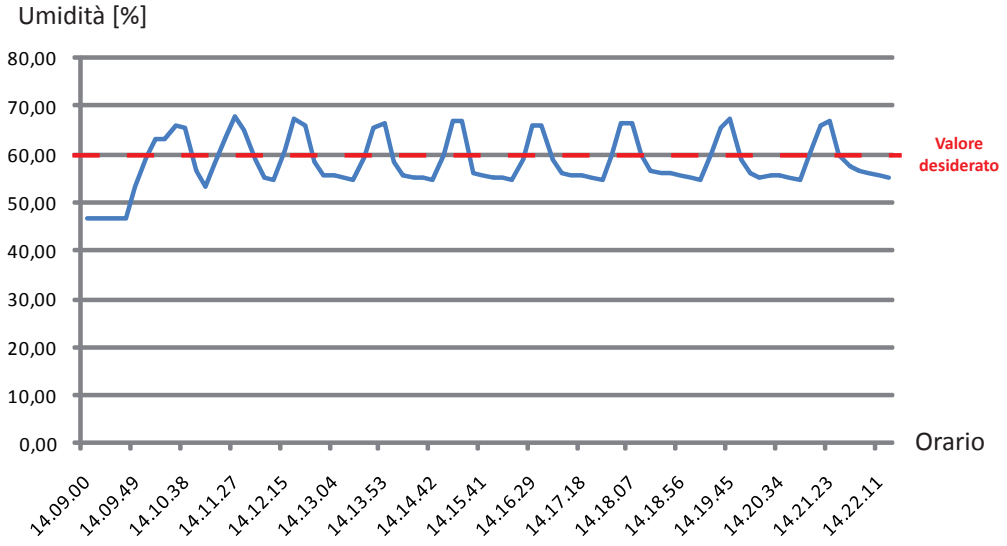


Figura 6.15: Prova controllo umidità.

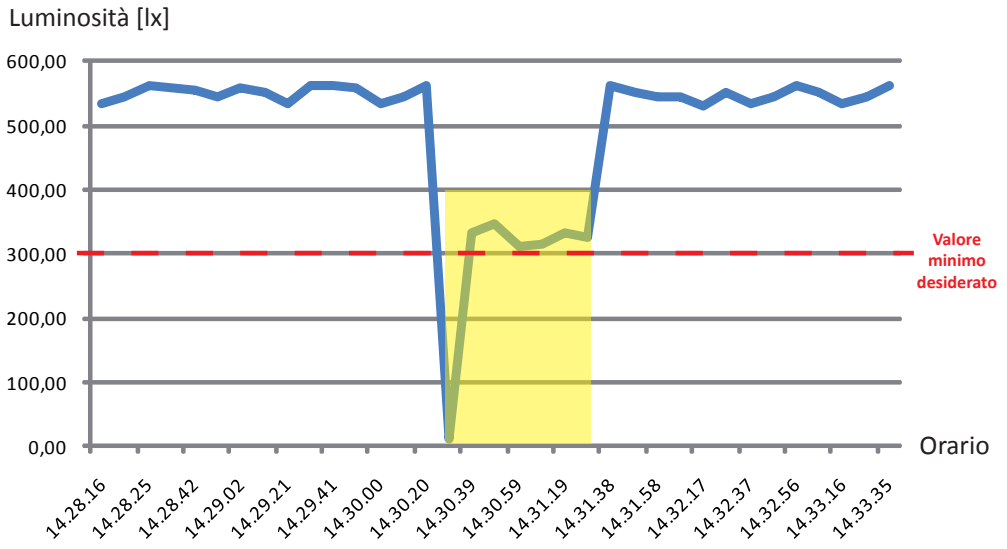


Figura 6.16: Prova controllo luminosità.

la situazione iniziale. Il sensore rileva ora una luminosità maggiore di $l_d + w = 400$ lx, e quindi WirMoS-Dashboard spegne le luci interne. Nella figura l'intervallo nel quale le luci del modello sono accese, è evidenziato da una fascia color giallo.

Conclusioni e sviluppi futuri

Lo scopo del lavoro presentato è stato la creazione di un sistema di monitoraggio e controllo ambientale open source, per reti di sensori wireless (WSN). In una prima fase del lavoro svolto sono state acquisite le competenze necessarie per la realizzazione del progetto, riguardanti in particolare gli strumenti open source utilizzati: il sistema operativo TinyOS e le piattaforme wireless Tmote Sky. Successivamente è stata fatta una ricerca sullo stato dell'arte dei protocolli più comuni per le WSN. Il livello fisico e parte del livello MAC, del modello standard di stack di protocolli per le WSN, sono già implementati nel dispositivo Tmote Sky e in TinyOS. Per quanto riguarda i livelli superiori si è deciso di realizzare un nuovo protocollo, in quanto attualmente non esistono protocolli open source che soddisfano tutte le specifiche di progetto: topologia multihop, router alimentati a batteria, basso consumo energetico, possibilità di variare il tempo di campionamento mantenendo inalterate le latenze, possibilità di interrogare i nodi, controllo della qualità delle trasmissioni con eventuale variazione della topologia, memorizzazione di dati per lo studio del comportamento del protocollo.

Come base di partenza per il protocollo del sistema WirMoS, è stato utilizzato il lavoro contenuto nell'articolo "Flexible Power Scheduling for Sensor Network" [1]. Il protocollo FPS, descritto nell'articolo, permette di ottenere il risparmio energetico in una rete di sensori organizzata ad albero, suddividendo il tempo in slot. In ogni slot un sensore trasmette un messaggio, oppure riceve un messaggio, oppure non fa nulla. In quest'ultimo caso può spegnere il chip radio, portando il consumo praticamente a zero. Questo meccanismo utilizzato anche da diversi altri protocolli, risulta vincente in quanto i nodi per la maggior parte del tempo restano inattivi. Il protocollo FPS però soffre di alcune limitazioni che non permettono il soddisfacimento delle già citate specifiche di progetto, e che sono state eliminate nel protocollo di WirMoS. Innanzitutto in FPS si da per assunto che i nodi siano logicamente già organizzati ad albero, la cui radice riceve i campioni di tutti i nodi e li invia al punto di raccolta dati. Il protocollo di WirMoS invece parte da zero, creando in una prima fase di setup della rete, una topologia ad albero a cammino minimo, formata da link affidabili, ovvero link nei quali la probabilità di perdita di pacchetto è bassa in entrambe le direzioni (verso la radice e verso le foglie). Un secondo problema importante di FPS, sta nel fatto che se si vuole aumentare il tempo di campionamento per diminuire i

consumi, otteniamo un indesiderato aumento delle latenze. Con WirMoS, grazie ai supercicli, la variazione del tempo di campionamento non comporta la variazione delle latenze. Sono stati inoltre studiati e implementati degli accorgimenti per diminuire il più possibile consumi e latenze. In primo luogo sono stati determinati quali sono i valori minimi dei parametri dell'FPS (durata slot, numero di slot, distanza tra le sincronizzazioni), che consentono di ottenere ottenere questo risultato, mantenendo la correttezza e la reattività del protocollo. In secondo luogo FSP utilizza il meccanismo del CSMA per risolvere le collisioni tra messaggi, che possono verificarsi quando più nodi trasmettono sullo stesso slot. WirMoS sostituisce il CSMA con la multiplazione a divisione di frequenza. Quando due nodi decidono su che slot trasmettere, decidono anche uno tra i 16 canali disponibili sul Tmote Sky. Questa modifica abbassa di un fattore 16 la probabilità di collisione, e soprattutto permette di risolvere l'eventuale collisione che si verifica se due o più nodi trasmettono sullo stesso slot e sulla stessa frequenza, semplicemente cambiando il canale di trasmissione. In questo modo non è necessario posticipare la trasmissione dei messaggi come accade con il CSMA, ed è quindi possibile mantenere slot di durata breve, con conseguenti bassi consumi e latenze. La multiplazione a divisione di frequenza ha un altro importante vantaggio, ovvero permette di utilizzare canali separati per le informazioni di controllo e le sincronizzazioni. Per i messaggi di sincronizzazione significa che non ci sono collisioni con altri tipi di messaggi. Questo, unito ad un altro accorgimento di WirMoS che garantisce l'assenza di collisioni tra messaggi di sincronizzazione di nodi diversi, permette di ottenere una sincronizzazione accurata, che è necessaria se si vuole mantenere gli slot brevi per avere bassi consumi e basse latenze.

Oltre alla progettazione e implementazione del protocollo sulle applicazioni da installare sui nodi, è stato realizzato un software (WirMoS-Dashboard) da installare su una workstation, il cui compito è quello di interfaccia tra la rete e gli utilizzatori, e di governare eventuali attuatori. WirMoS-Dashboard permette di creare, memorizzare e modificare le reti a cui connettersi, utilizzando una pianta del sito controllato, sulla quale vanno posizionati i sensori nel punto dove si trovano. Quando l'applicazione è connessa ad una rete i campioni che vengono ricevuti possono essere visualizzati in tempo reale. Per ogni sensore è possibile conoscere il valore delle variabili monitorate, la stima della durata delle sue batterie, l'istante in cui è stato generato il campione, e altre informazioni tra le quali quelle relative alla qualità dei link con gli altri nodi. Sulla pianta vengono tracciati in automatico, e aggiornati, i link tra i vari sensori, e vengono visualizzati con colore diverso in base in base alla percentuale di pacchetti persi. I campioni di ogni sensore vengono salvati sulla memoria della workstation, e attraverso WirMoS-Dashboard è possibile recuperarli, e visualizzare quelli di interesse in forma tabellare o di grafici. L'utente può, utilizzando degli opportuni controlli di WirMoS-Dashboard, modificare dei parametri della rete mentre questa in esecuzione, come ad esempio il tempo di campionamento. Se la workstation è con-

nessa ad Internet è possibile accedere ad una rete da un qualsiasi luogo remoto, sfruttando semplicemente un browser. Infine WirMoS-Dashboard permette di controllare eventuali attuatori connessi alla rete, in modo manuale, oppure in modo automatico attraverso delle funzioni di controllo preinstallate.

L'ultima fase del lavoro svolto consiste nelle prove del sistema per la verifica del corretto funzionamento, e la rilevazione delle performance. Ciò che ci interessa maggiormente è capire qual'è il risparmio energetico che si ottiene utilizzando il protocollo di WirMoS. Le considerazioni fatte in questa fase, supportate da prove e misurazioni sul campo, evidenziano risultati soddisfacenti. Ad esempio in una rete con topologia ad albero ternario completo di altezza 4, che ha 121 nodi, i nodi a profondità 1, che sono quelli che consumano di più, hanno una durata di 1,6 anni, mentre quelli a profondità 4 hanno durata di 12,5 anni. Se non si utilizza il protocollo un sensore smette di funzionare dopo 6 giorni. Per testare la parte di controllo, il sistema WirMoS viene utilizzato per mantenere ad un valore constante desiderato, temperatura, umidità, e luminosità di un modello di una serra di 1.5x1.1x1 metri.

WirMoS può essere certamente esteso e migliorato ulteriormente. Per quanto riguarda in particolare i consumi, questi possono essere ridotti utilizzando l'aggregazione. Questa strategia consiste nell'utilizzare un unico messaggio di un nodo genitore, per trasmettere i campioni ricevuti da più figli. Il risparmio energetico si ottiene perché l'aumento della grandezza dei messaggi non costringe ad aumentare la durata degli slot, e permette invece di diminuire il numero slot di tipo T, nei quali la radio è accesa. Un ulteriore aumento del tempo di vita dei sensori, è sicuramente ottenibile attraverso l'utilizzo di pannelli fotovoltaici da installare su ogni sensore, come già accennato nel Capitolo 1. E' possibile trovare in commercio sistemi che sfruttano questa soluzione, e in letteratura articoli che ne dimostrano l'efficacia, sopratutto se i sensori vanno posizionati all'aperto. Altri miglioramenti di WirMoS che potrebbero essere effettuati riguardano la robustezza del sistema, soprattutto nella fase di regime. Attualmente la qualità delle trasmissioni è garantita dalla scelta iniziale di link affidabili, e da un eventuale variazione della topologia quando i pacchetti persi diventano molti. In ambienti molto disturbati questo potrebbe non essere sufficiente, a mantenere costantemente sotto un adeguato livello, il numero di pacchetti persi. Una soluzione può essere quella di creare n copie dello stesso messaggio, che dovranno percorrere percorsi diversi per arrivare alla radice dell'albero. Questa ridondanza permette di ottenere che un campione di rilevazione non venga ricevuto dalla workstation, solo quanto tutte le sue n copie sono andate perse. Il prezzo da pagare per ottenere questa robustezza è però un aumento del consumo medio dei nodi.

Bibliografia

[1] B. Hohlt, L. Doherty, E. Brewer, *"Flexible Power Scheduling"*, Proceedings of the 3rd international symposium on Information processing in sensor networks (IPSN'04), Berkeley, California, USA, 26-27 April 2004, pp. 205-214.

[2] M.A. Labrador, P.M. Wightman, *"Topology Control in Wireless Sensor Networks"*, Springer 2009.

[3] L. Ruiz-Garcia, P. Barreiro, J. Rodriguez-Bermejo, J.I. Robla, *"Monitoring the intermodal, refrigerated transport of fruit using sensor networks"*, Spanish Journal of Agricultural Research, Vol. 2, No. 5, pp. 142-156, 2007.

[4] P.S. Pandian, K.P. Safeer, Pragati Gupta, D.T. Shakunthala, B.S. Sundersheshu, V.C. Padaki, *"Wireless sensor network for wearable physiological monitoring"*, Journal of Networks, Vol. 3, No. 5, pp. 21-29, May 2008.

[5] T. Bokareva, W. Hu, S. Kanhere, B. Ristic, N. Gordon, T. Bessell, M. Rutten, S. Jha, *"Wireless sensor networks for battlefield surveillance"*, Proceedings of Land Warfare Conference (LWC'06), Brisbane, Queensland, Australia, October 2006.

[6] J. M. Kahn, R. H. Katz, K. S. J. Pister, *"Next century challenges: Mobile networking for smart dust"*, Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking (MOBICOM'99), Seattle, Washington, USA, 1999, pp. 271-278.

[7] C. Edordu, L. Sacks, *"Self organising wireless sensor networks as a land management tool in developing countries: A preliminary survey"*, Proceedings of the 2006 London Communications Symposium, Communications Engineering Doctorate Centre, London, UK, September 2006.

[8] J. A. Stankovic, Q. Cao, T. Doan, L. Fang, Z. He, R. Kiran, S. Lin, S. Son, R. Stoleru, A. Wood, *"Wireless sensor networks for in-home healthcare: Potential and challenges"*, Proceedings of High Confidence Medical Device Software and Systems (HCMDSS'05) Workshop, Philadelphia, Pennsylvania, USA, 2-3 June 2005.

[9] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, J. Anderson, *"Wireless sensor networks for habitat monitoring"*, Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA'02), Atlanta, Georgia, USA, September 2002, pp. 88-97.

[10] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, M. Welsh, *"Monitoring volcanic eruptions with a wireless sensor network"*, Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN'05), January 2005.

[11] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, W. Hong, *"A macroscope in the redwoods"*, Proceedings of the 3rd international conference on Embedded networked sensor systems (SENSYS'05), San Diego, California, USA, November 2005, pp. 51-63.

[12] *"Crossbow eKo solution"*, http://blog.xbow.com/xblog/eko.

[13] X. Jiang, J. Polastre, D. Culler, *"Perpetual Environmentally Powered Sensor Networks"*, Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN'05), Los Angeles, California, USA, 2005, Article No. 65.

[14] *"TinyOS Home Page"*, http://www.tinyos.net/.

[15] G.J. Pottie, W.J. Kaiser, *"Wireless Integrated Network Sensors"*, Communications of the ACM, Vol. 43, No. 5, May 2000, pp. 51-58.

[16] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, D. Culler, *"An analysis of a large scale habitat monitoring application"*, Proceedings of the 2nd international conference on Embedded networked sensor systems, (SENSYS'04), Baltimore, Maryland, USA, November 2004, pp. 214-226.

[17] *"MsstatePAN stack home page."*, http://www.ece.msstate.edu/~reese/msstatePAN/.

[18] Joel K. Young, *"Untangling the Mesh-The Ins and Outs of Mesh Networking Technologies"*, white paper, http://www.digi.com/learningcenter/literature/whitepapers.jsp.

[19] Niek Van Dierdonck, *"Understanding low-power wireless network standards"*, EE Times-Asia, 04 Mar 2008.

[20] R.S. Pressman, *"Principi di ingegneria del software 4ed"*, McGraw-Hill 2005.

[21] D. Gay, P. Levis, D. Culler, E. Brewer, *"nesC 1.3 Language Reference Manual"*, http://nescc.sourceforge.net, June 2009.

[22] D. Gay, P. Levis, R. von Behren, *"The nesC Language: A Holistic Approach to Networked Embedded Systems"*, In Proceedings of the ACM SIGPLAN 2003 Conference on

Programming Language Design and Implementation (PLDI'03), San Diego, California, USA, June 2003, pp. 1-11.

[23] P. Levis, *"TinyOS Programming"*, TinyOS tutorial, www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf, 27 October 2006.

[24] *"Eyes project home page"*, http://www.eyes.eu.org/index.htm.

[25] *"Arch Rock home page"*, http://www.archrock.com/.

[26] *"Energizer home page"*, http://www.energizer.com/.

[27] *"Tutorials TinyOS"*, http://docs.tinyos.net/index.php/TinyOS_Tutorials.

[28] *"JFreeChart home page"*, http://www.jfree.org/jfreechart/.

Elenco delle figure

Elenco delle tabelle

Ringraziamenti

Ringrazio il professore Luca Schenato per la sua disponibilità, Cinzia, i miei genitori, Claudio Lora, Augusto Tazzoli, e tutti quelli che in modo più o meno diretto hanno contribuito alla realizzazione di questo lavoro.