View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Electronic Thesis and Dissertation Archive - Università di Pisa

UNIVERSITÀ DEGLI STUDI DI PISA
Facoltà di Scienze, Matematiche, Fisiche e Naturali
Corso di Laurea in Tecnologie Informatiche

Realizzazione di uno strato di rete per lo stack MaD-WiSe

RELATORE:

Prof. Stefano Chessa

CONTRORELATORE:

Prof. Fabrizio Baiardi

Candidato:

Francesco Gigliucci

ANNO ACCADEMICO 2005-2006

Indice

INTRODUZIONE **3**

1 MAD-WISE **4**

2 AMBIENTE DI SVILUPPO **7**

2.1 NesC 7
2.2 TinyOS 8

3 SPECIFICHE **9**

3.1 Interfaccia Network – Stream System 9
3.2 Interfaccia Network – Query Executor 12
3.3 Interfaccia MAC - Network 12
3.4 Pacchetti di livello MAC 13
3.5 Pacchetti di livello network 14
3.6 Protocollo di controllo 16
3.7 Protocollo di funzionamento a regime della rete 19

4 IMPLEMENTAZIONE **22**

4.1 Overview 22
4.2 Struttura del network 23
4.3 Strutture dati 28
4.4 Tasks 31
4.5 Energy Efficiency 32
4.5.1 Interfaccia EEtoNET *35*
4.5.2 Algoritmo OffTimer *37*

5 CONCLUSIONI **40**

INDICE TABELLE **42**

INDICE FIGURE **43**

BIBLIOGRAFIA **44**

Introduzione

Il lavoro svolto consiste nella realizzazione di uno strato di rete affidabile per reti di sensori. Lo stack protocollare in cui si va ad inserire (chiamato MaD-WiSe) è composto da un livello applicativo di gestione di query, un livello di trasporto nel quale si organizzano le connessioni, un livello di rete su cui sono implementate le comunicazioni tra i nodi. Le interrogazioni verso il database/rete di sensori avvengono tramite un'interfaccia utente: l'applicazione le gestisce e attraverso il livello di rete fa interagire i nodi. I servizi offerti dal livello di rete sono realizzati sfruttando l'interfaccia radio presente su ogni nodo.

Al fine di garantire un'applicazione reale del MaD-WiSe, il network fornisce supporto al routing *multi-hop*. Un server invia comandi ad un nodo speciale chiamato *sink*, in base ai dati ricevuti dall'interfaccia utente e questo inoltra i messaggi verso gli altri nodi: la stessa gestione dei pacchetti è comunque identica in tutta la rete, che può quindi estendersi anche oltre il raggio di azione della ricetrasmittente di un singolo sensore. I protocolli definiti per sincronizzare i nodi con il server e per far funzionare efficacemente l'applicazione sono implementati tenendo in considerazione diverse topologie possibili.

Un'ulteriore funzionalità è implementata con il modulo *energy efficiency*: dato che la gestione dell'energia è molto critica, è stato costruito un meccanismo per inserire dei periodi di radio sleep durante il ciclo di vita dei sensori. In particolare l'algoritmo fornito si propone di estendere questa durata mantenendo inalterate le funzioni riguardanti la corretta esecuzione dell'applicazione in ogni momento.

1 MaD-WiSe

Il MaD-WiSe è un'applicazione per reti di sensori che permette di eseguire query verso una rete tramite un'interfaccia (*gui*) utilizzando un linguaggio SQL-like: l'insieme dei nodi viene trattato come se fosse un database che risponde alle richieste di un server e che comunica via radio i dati rilevati. L'interfaccia attraverso la quale un utente comunica con la rete permette di scrivere le query con un semplice editor, avviarle e bloccarle con un pulsante e analizzare i dati ricevuti dai sensori; inoltre è possibile visualizzare il piano di esecuzione che viene creato dall'applicazione per ottenere il risultato.

In figura 1 sono mostrati i tre ambienti che fanno parte dell'esecuzione:

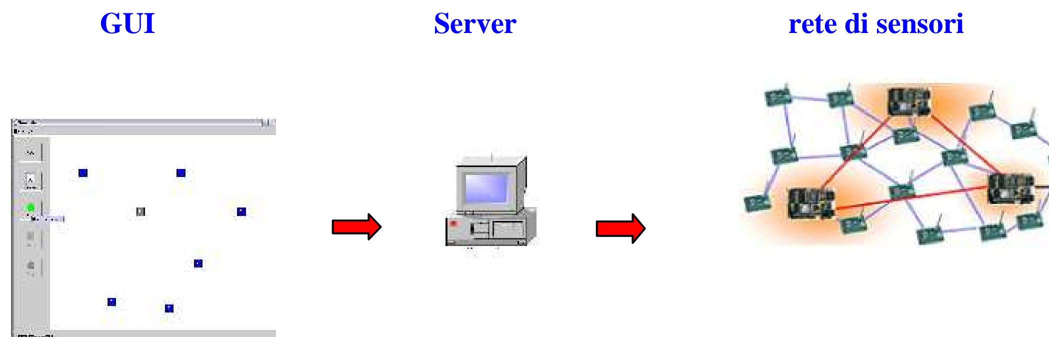

Figura 1: Struttura dell'applicazione MaD-WiSe

I comandi vengono inviati dalla *gui* alla rete passando per un server: le informazioni necessarie per svolgerli vengono inviate ai sensori secondo un protocollo stabilito che prepara inoltre le rotte di comunicazione per garantire un corretto andamento della query.

Lo stack protocollare del MaD-WiSe è costituito da diversi livelli come mostrato in figura 2. Al livello più alto (livello applicazione) troviamo il Query Executor che sfrutta i servizi offerti sia dallo Stream System (livello di trasporto) che direttamente dal Network (livello di rete), su due diverse interfacce. Il livello di rete è implementato sulle

funzionalità offerte dal MAC che prevede funzionalità per inviare e ricevere dati tra i nodi accedendo alla radio.

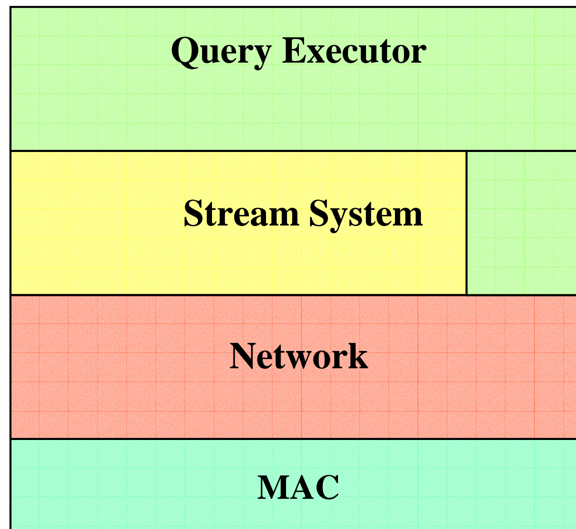

Figura 2: Stack del MaDWiSe

Il Query Executor è una configurazione composta da due moduli chiamati Query Manager e Query Processor ed è il livello applicazione. Questi due moduli si occupano rispettivamente delle operazioni di analisi dei comandi ricevuti dal server e della gestione e dell'invio dei dati rilevati verso di esso, eseguendo di fatto una parte della query.
Al di sotto di questo si trova uno strato di trasporto, Stream System, che offre un servizio orientato alla connessione per inviare i dati lungo la rete. A questo livello si gestisce lo scambio tra i nodi della rete degli stream di dati prodotti dai sensori.
Il livello di rete fornisce funzionalità sia allo Stream System che al Query Executor: per il primo mette a disposizione servizi per aprire comunicazioni orientate alla connessione e per la trasmissione dei dati, per il secondo offre un servizio per l'invio dei comandi ai nodi della rete: quest'ultimo non è orientato alla connessione e viaggia secondo un routing alternativo, in quanto precede la costruzione delle rotte su cui viaggeranno i dati.

MaD-WiSe funziona come un database di sensori che rilevano dati e li inviano all'utente. Il server fa da tramite impostando il corretto formato dei pacchetti per comunicare con la rete e programmando opportunamente i sensori coinvolti, al fine di distribuire l'esecuzione della query fra i vari nodi. Nello specifico si deve fare in modo che il flusso di dati rilevati (*stream*) viaggi da un nodo ad un altro fino a raggiungere il server che proietta i dati sull'interfaccia utente. Lo stream dei dati viene aperto dal nodo al quale è richiesta la rilevazione verso il nodo a cui i dati vengono trasmessi.

Il server programma tutte le rotte che devono essere costruite per portare le informazioni all'utente e invia ai nodi i comandi necessari. Per esempio la disposizione di effettuare una nuova connessione tra nodi arriva dal server tramite un comando (*RemoteOpen*) inviato al nodo con il quale si deve effettuare l'operazione; così si apre lo stream.

Uno stream cessa di esistere quando una query viene fermata: in questo caso il server invia un altro tipo di comando (*Close*) ai nodi per determinare la chiusura della connessione.

Gli altri comandi che il server invia preparano le funzionalità richieste dall'utente sui nodi e fanno parte del protocollo di funzionamento a regime della rete.

2 Ambiente di sviluppo

2.1 NesC

MaD-WiSe è scritto in NesC, una variante del linguaggio C, sviluppata per la programmazione dei sensori. In questo linguaggio si separa il concetto di costruzione del codice dal concetto di composizione di una applicazione. I componenti del NesC possono essere di due tipi: *moduli* o *configurazioni*: i moduli forniscono il codice dell'applicazione implementando una o più interfacce, le configurazioni sono utilizzate per collegare staticamente i componenti, specificando quali sono le interfacce usate e quali moduli le forniscono. Questo aumenta l'efficienza a tempo di esecuzione e permette di definire i collegamenti ad alto livello tra i moduli. Le interfacce sono l'unico punto di accesso ai componenti e i moduli sono assemblati tra di loro come specificato nelle configurazioni.

In NesC sono previste tre tipi di funzioni: *comandi*, *eventi* e *task*.

I *comandi* comportano l'esecuzione di una azione e sono funzioni che possono essere chiamate da un modulo che usa l'interfaccia. Il modulo che li implementa definisce il codice delle funzioni e le espone come pubbliche a quelli che vorranno utilizzarle.

Gli *eventi* sono gestiti da handler che si occupano della segnalazione di un'azione completata o di qualcosa che è accaduto; sono eseguiti in modo atomico e solitamente sono associati ad interruzioni di basso livello, come la rilevazione di un dato da un sensore o il completamento di un'operazione richiesta. Per questo motivo un evento che viene segnalato da un livello inferiore mentre un comando viene invocato in senso opposto ovvero da un livello superiore.

I *task* sono procedure che vengono eseguite in sequenza secondo una politica FIFO e senza interferire tra di loro: si occupano di azioni delicate che non devono modificare

strutture dati in contemporanea. I task possono essere interrotti soltanto da eventi e in tal caso riprendono l'esecuzione una volta che questi sono terminati.

2.2 TinyOS

TinyOS è un sistema operativo per reti wireless di sensori creato dall'università di Berkeley in California: è interamente scritto in NesC, si basa su un modello a eventi per garantire efficienza e risparmio di risorse e mette a disposizione componenti per gestire l'hardware. E' composto da un insieme di task e processi cooperativi che operano sotto le rigide limitazioni di memoria tipiche di queste tipologie di reti e a differenza dei normali sistemi operativi, non ha un kernel, non è multi-thread e non ha un gestore della memoria; tuttavia permette di acquisire dati dai dispositivi posti su un nodo e gestire l'invio e la ricezione di pacchetti via radio.

TinyOS è gestito a moduli: ogni componente può implementare i comandi forniti da un'interfaccia e esporli verso un altro componente, oppure usare i comandi implementati da un altro modulo. Le applicazioni dispongono di una configurazione che specifica i collegamenti tra i moduli e le interfacce: in questo modo è possibile creare un qualsiasi insieme di moduli collegati tra loro in base alle esigenze del programmatore.

Un componente che fornisce un'interfaccia deve avere un'implementazione di tutti i comandi contenuti al suo interno mentre un componente che usa un'interfaccia deve implementare un gestore per tutti gli eventi in essa dichiarati: questi eventi vengono segnalati dal componente che offre tale interfaccia.

3 Specifiche

Il livello network deve garantire l'invio e la ricezione di messaggi tra i nodi della rete di sensori. E' organizzato in due moduli: *NetToSSM* che definisce il routing e *EnergyEfficiencyM* che determina la politica di risparmio energetico. Al livello sottostante si appoggia sul livello MAC, che si occupa della gestione della radio.

Le interfacce con i livelli superiori sono *ConnectionOriented* e *ConnectionLess* e sono utilizzate rispettivamente dal livello dello Stream System (strato di trasporto) e dal livello del Query Executor (strato applicazione).

3.1 Interfaccia Network – Stream System

L'interfaccia *ConnectionOriented* è fornita dal livello di rete per gestire una connessione tra due nodi qualsiasi: grazie ai comandi che sono implementati è possibile aprire e chiudere un canale e inviare dati lungo esso. Un *canale* è una rotta fissata per un cammino tra due nodi e viene memorizzato su tutti i nodi da cui transita. In figura 3 si può vedere come le strutture dati dei singoli nodi tengano traccia del canale che vedono passare mentre altri nodi, eventualmente vicini, possano non essere parte integrante del cammino stabilito. Ogni nodo può tenere più canali diversi che transitano da lui.

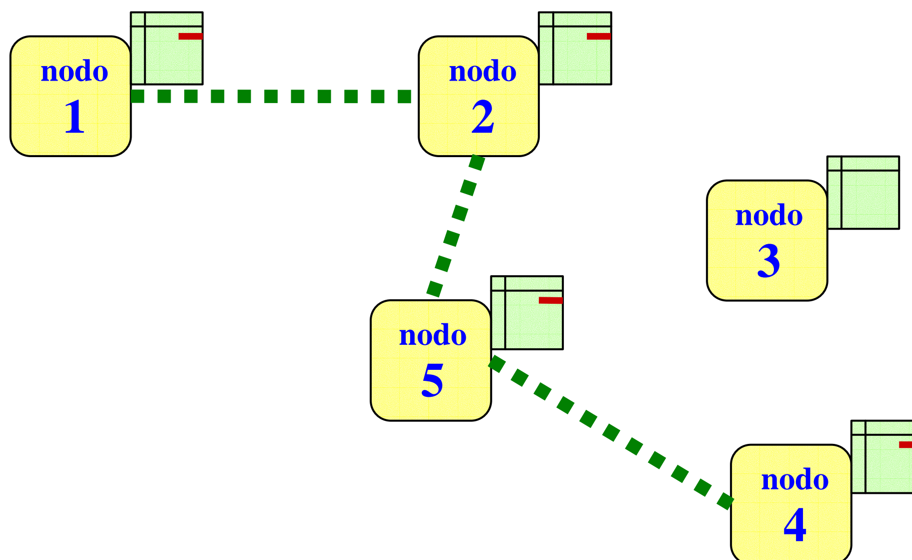

Figura 3: Schema di un canale nella rete

La primitiva per aprire un canale si chiama *connect*: questa prevede di memorizzare le informazioni relative al cammino che sarà stabilito in una tabella sia sul nodo sorgente che sul nodo destinatario. L'invocazione viene effettuata a livello Stream System dal modulo che si occupa del livello di trasporto dei pacchetti: i dati rilevati dai sensori viaggiano sulle rotte determinate da questa procedura.

L'operazione di cancellazione di tali cammini avviene invece con l'invocazione della primitiva *disconnect*: con questa invocazione viene inviato un pacchetto che provoca la rimozione dell'entry relativa al canale nella tabella delle connessioni lungo tutto il percorso del canale: su ogni nodo il canale viene così eliminato e può essere sovrascritto da un altro.

Nel periodo che intercorre tra queste due operazioni l'applicazione funziona regolarmente e i canali che esistono sono le rotte stabilite in precedenza: i pacchetti vengono trasmessi lungo questi canali e seguono un percorso fisso; le comunicazioni, da parte dei nodi, dei dati richiesti dalla query sono periodiche, in base ad un parametro specificato dall'utente, e sono realizzate per mezzo della primitiva *send*. Non è previsto un ack per le

trasmissioni dati sui canali in quanto i dati sono rinnovati continuamente e non avrebbe senso rispedire un dato vecchio di cui si è persa traccia.

Oltre a questi tre comandi l'interfaccia prevede la segnalazione di due eventi allo Stream System: *receive* e *connectDone*. Quando un pacchetto viene ricevuto dalla radio di un nodo, l'handler dell'evento gestisce opportunamente i vari tipi di messaggio esistenti nella rete e schedula le operazioni necessarie; nel caso la gestione del pacchetto debba arrivare anche al livello superiore si segnala l'evento *receive* e lo Stream System lo gestisce in base al protocollo. L'altro evento, *ConnectDone*, viene segnalato nel momento in cui viene spedito un messaggio di tipo connect: l'handler dell'evento *sendDone*, generato quando il pacchetto viene trasmesso via radio, provvede a segnalare anche la *connectDone*. Questa soluzione temporanea è richiesta in quanto non è ancora disponibile un meccanismo di acknowledgement per la creazione di un canale. Un possibile sviluppo futuro del network prevede l'introduzione di una maggiore affidabilità per quanto riguarda la creazione delle connessioni: in tale scenario l'evento *connectDone* viene generato al momento in cui un nodo mittente riceve l'acknowledgement per un pacchetto di connect spedito precedentemente.

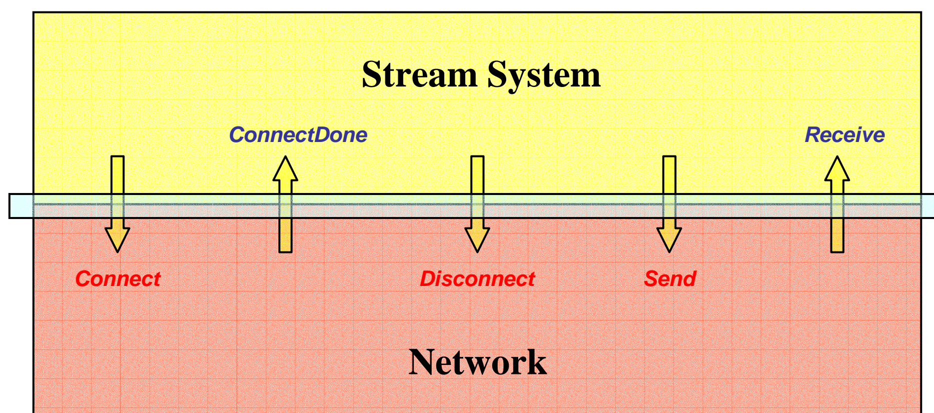

Figura 4: Interfaccia ConnectionOriented (Network – Stream System)

3.2 Interfaccia Network – Query Executor

L'interfaccia *ConnectionLess* è fornita dal livello di rete direttamente verso il livello applicazione, dal quale è utilizzata per inviare comandi per impostare e eseguire una query. L'interfaccia fornisce un'unica primitiva che permette di inviare un pacchetto dal server verso i nodi coinvolti: il percorso da seguire è indicato da tabelle di rotta inviate dal server nella fase di inizializzazione della rete. La primitiva è chiamata *asysend*, trattandosi di un invio asincrono rispetto all'invio dei dati sui canali ed è invocata sia dal server verso i nodi della rete (quelli a cui viene richiesta l'esecuzione della query), sia fra i nodi stessi per inoltrare le disposizioni arrivate.

In corrispondenza dell'invio di questi messaggi, l'interfaccia prevede la segnalazione di un evento di ricezione chiamato *asyreceive*: questo evento è segnalato dal network verso il livello superiore e il messaggio viene gestito dal Query Executor a livello applicazione.

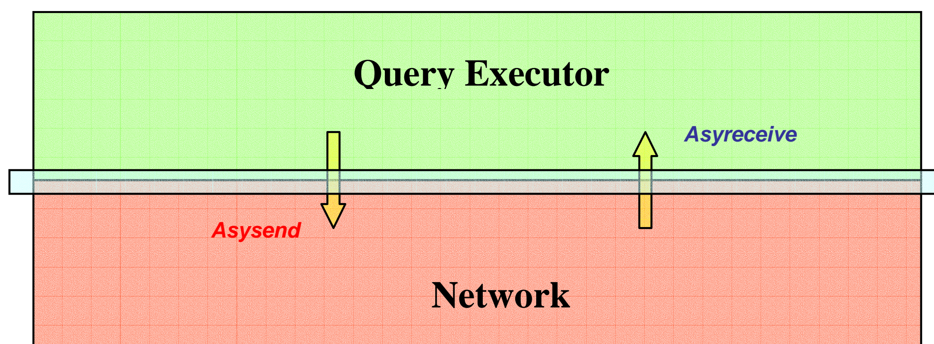

Figura 5: Interfaccia Connectionless (Network – Query Executor)

3.3 Interfaccia MAC - Network

Il livello network sfrutta le funzionalità offerte dal livello MAC per inviare e ricevere messaggi dalla radio. Queste operazioni sono realizzate rispettivamente tramite l'invocazione di un comando *send* e l'implementazione di un handler per l'evento *receive*.

Il primo fa parte dell'interfaccia SendMsg e il secondo dell'interfaccia *ReceiveMsg*: entrambe sono offerte dal *GenericComm*, un modulo che si occupa della gestione dei messaggi a livello MAC.

Il comando *send* viene invocato per inviare un messaggio verso un altro nodo: l'implementazione del comando al livello sottostante consente di spedire tale messaggio sulla radio. Una volta inviato, il livello MAC segnala al network l'evento *sendDone* a completamento dell'operazione.

La funzione di gestione dell'evento *receive* analizza l'arrivo di un messaggio: al momento della ricezione sulla radio (livello MAC) il modulo *GenericComm* provvede a segnalare l'evento al network che gestisce il pacchetto a seconda delle necessità.

3.4 Pacchetti di livello MAC

Il pacchetto che viaggia sulla radio è composto da un header di livello MAC e da un payload contenente il messaggio di livello network.

L'header è formato da 5 bytes: un campo *address* di 2 bytes e tre altri campi, *type*, *group* e *length*, tutti di 1 byte come indicato schematicamente in tabella 1.

Nome campo	Lunghezza	Descrizione
address	16 bit	Indirizzo di livello MAC: indica il nodo della rete a cui verrà inviato il pacchetto (prossimo hop)
type	8 bit	Tipo del pacchetto (tutti i pacchetti inviati sono di tipo RADIO, viaggiano cioè tramite le radio dei nodi)
group	8 bit	Questo campo verrà utilizzato per sviluppi futuri
length	8 bit	Lunghezza del payload del pacchetto: indica quanto è grande il pacchetto di livello Network, contenuto nel payload
data	variabile	Payload del pacchetto di livello MAC: contiene il pacchetto di livello Network.

Tabella 1: header di livello MAC

Il campo *address* corrisponde all'indirizzo di livello MAC a cui è destinato il pacchetto: con un'estensione di 16 bit si possono indirizzare fino a 65536 nodi di una rete. Il campo *type* indica se il pacchetto è di tipo radio. I pacchetti che vengono inviati nella rete devono essere tutti di questo tipo. Il campo *group* non viene utilizzato dall'applicazione e può essere sfruttato per sviluppi di applicazioni in futuro. Il campo *length* indica la dimensione del payload del pacchetto ovvero la lunghezza del messaggio che trasporta.

L'array di bytes *data* è il payload del pacchetto e corrisponde ad un messaggio di livello network.

3.5 Pacchetti di livello network

Il pacchetto di livello network è formato da un header e da un payload.

L'header è una struttura di 4 bytes composta da un campo *type*, un campo *ack*, due campi indirizzo *dest* e *src*, infine un numero progressivo *seqno*. In tabella 2 è mostrato lo schema.

Nome campo	Lunghezza	Descrizione
type	5 bit	Tipo di pacchetto. Si possono riferire fino a 32 tipi differenti. (Vedi tabella successiva)
ack	1 bit	Acknowledgement: indica se si tratta di un pacchetto di conferma
foo	2 bit	Questi due bit sono lasciati liberi per sviluppi futuri
dest	8 bit	Indirizzo finale del pacchetto: sul nodo corrispondente a questo valore il pacchetto si fermerà. E' possibile riferire fino a 256 nodi in una rete
src	8 bit	Indirizzo del nodo che ha creato il pacchetto. E' possibile riferire fino a 256 nodi sorgente.
seqno	8 bit	Contatore progressivo: insieme al sorgente identifica univocamente un pacchetto e serve a tenere traccia di eventuali pacchetti doppi ricevuti sui nodi.

Tabella 2: header di livello Network

Il campo *type* occupa 5 bit dell'header e indica il tipo di livello network del messaggio: in base a questo dato si riconosce la funzione di un pacchetto al momento della sua ricezione. E' possibile specificare fino a 2^5 = 32 tipi diversi. Attualmente sono definiti i seguenti tipi (la funzione di ognuno di questi verrà spiegata in seguito):

ASYSEND	0
CONNECT	1
SEND	2
DISCONNECT	3
SETCLOCK	4
HELLO	5
ND_START	6
BUILD_TREE	7
EE_START	8
EE_STOP	9
NEIGHBOR_REPORT	10
ROUTINGTABLE	11
ASYSEND_WITH_ACK	12

Tabella 3: tipi di messaggio di livello Network

Il campo *ack* segnala se il pacchetto è un acknowledgement: se il bit è a 1 allora il pacchetto è un *ack*, una conferma della ricezione corretta di un pacchetto precedente. Gli ultimi 2 bit del primo byte non sono attualmente in uso e possono servire per definire nuove funzionalità in futuro. Il byte successivo indica la destinazione del messaggio: al contrario del livello MAC, nel quale questo campo corrispondeva al prossimo hop da effettuare, il campo *dest* indica la meta finale del cammino del pacchetto nella rete.

Analogamente il byte *src* indica il mittente, il nodo che ha effettivamente originato il pacchetto. Infine il campo *seqno* è un numero progressivo, generato su ogni nodo indipendentemente e stampato in ogni pacchetto per tracciare una cronologia degli invii da parte di ogni nodo: lo scopo di questo accorgimento è quello di evitare che nella rete si ricevano pacchetti già ricevuti, occupando inutilmente risorse.

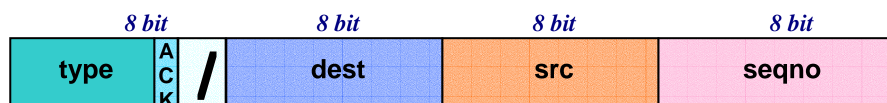

Figura 6: header di livello Network

Nelle figure che seguono l'header di livello network è indicato come un semplice blocco unico: in realtà, come appena descritto, i 4 bytes sono suddivisi come mostrato in figura 6.

3.6 Protocollo di controllo

Nella fase di inizializzazione tutte le radio sono accese e pronte per comunicare coi nodi vicini. Il server esegue un protocollo di controllo inviando messaggi nella rete per analizzarne la topologia e creare tabelle di routing per ogni nodo.

In figura 7 è mostrato il primo messaggio inviato: è un messaggio broadcast di tipo SETCLOCK nel quale il server scrive un timestamp (valore di 64 bit) e alla ricezione del quale ogni sensore corregge il suo clock interno in base al dato ricevuto. Grazie a questa sincronizzazione, che si ripete periodicamente, gli orologi dei singoli nodi si mantengono sincronizzati tra loro e con il server rendendo possibile effettuare in seguito il protocollo di energy efficiency. La sincronizzazione tra i nodi non è stretta in quanto la latenza del messaggio introduce una piccola differenza di temporizzazione. Dato che MaD-WiSe è progettato per applicazioni a basso rate questo errore di sincronizzazione non è considerato un problema.

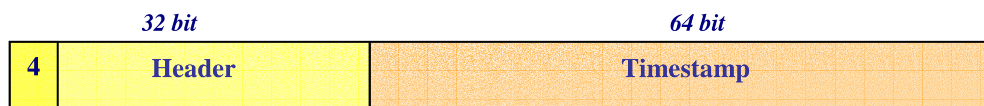

Figura 7: messaggio di tipo SETCLOCK

Il protocollo vero e proprio comincia con l'invio di un pacchetto broadcast di tipo ND_START, composto soltanto dall'header: alla ricezione è sufficiente l'analisi del tipo del messaggio perchè ogni nodo faccia partire la fase di neighbor discovery per conoscere i propri vicini.

Figura 8: messaggio di tipo ND_START

Questo pacchetto provoca l'avvio di un timer di durata casuale alla cui scadenza vengono programmati invii periodici (di default sono tre) di pacchetti contenenti i vicini del nodo: questi pacchetti hanno tipo HELLO e contengono una lista di identificatori dei nodi dai quali il nodo può ricevere via radio. Ogni volta un nodo invia un HELLO in broadcast e nella lista *Neighbors[]* inserisce gli identificatori dei nodi che hanno emesso i pacchetti HELLO ricevuti nel frattempo. Al momento della ricezione di tali pacchetti il nodo aggiorna una tabella di identificatori di nodi vicini: questa indica una prima topologia di rete, anche se limitata localmente al nodo.

Figura 9: messaggio di tipo HELLO

Dopo un periodo di tempo prefissato, il server invia al sink un pacchetto broadcast di tipo BUILDTREE: anche questo non ha un payload e in base al tipo il nodo costruisce un messaggio da inviare verso il server con le informazioni locali che possiede sulla rete.

Figura 10: messaggio di tipo BUILD_TREE

Alla ricezione ogni nodo costruisce un pacchetto di tipo NEIGHBOR_REPORT da inviare verso il server: esso contiene la lista degli identificatori dei nodi vicini con i quali esiste un link attivo e bidirezionale. Il formato è analogo a quello del pacchetto di tipo HELLO. Non esistendo ancora una rotta sicura e valida, si invia verso il server attraverso il nodo "parent", ovvero il primo da cui era stato ricevuto un BUILDTREE.

Completata questa fase il server riceve da tutti i nodi collegati una lista di link affidabili sulla rete e può creare una tabella di routing per ognuno di essi. La tabella è calcolata usando l'algoritmo di Dijkstra per i cammini minimi su un grafo.

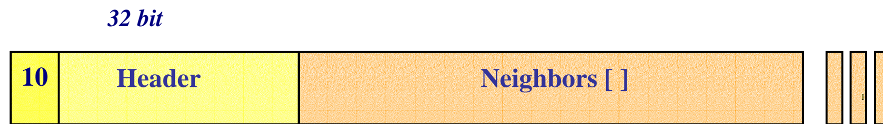

Figura 11: messaggio di tipo NEIGHBOR_REPORT

Il pacchetto successivo è la ROUTINGTABLE e il server lo invia ai singoli nodi separatamente, attendendo poi un ack di risposta e schedulando l'eventuale ritrasmissione; questo succede perché senza la tabella delle rotte un nodo non saprebbe inoltrare i pacchetti ai suoi vicini e i canali non potrebbero funzionare.

L'array RT contiene una lista di coppie:

`<identificatore destinazione, prossimo_hop>`.

Da questo momento ogni nodo conosce un percorso verso qualsiasi destinazione e può inoltrare i messaggi usando il campo `prossimo_hop`.

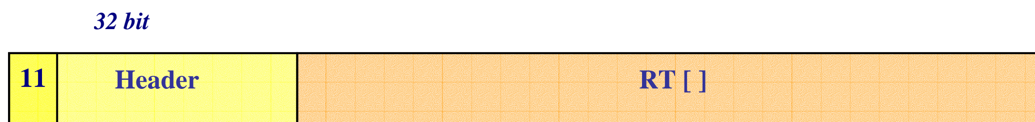

Figura 12: messaggio di tipo ROUTINGTABLE

L'operazione conclusiva del protocollo di controllo è l'invio di un EE_START in broadcast. Pure questo pacchetto non ha un payload e il comando che attiva viene riconosciuto soltanto dal tipo del messaggio. Si avvia così il meccanismo di risparmio energetico e ogni nodo analizza gli istanti in cui la radio deve stare accesa a causa di invii/ricezioni di dati sui canali che gestisce.

Figura 13: messaggio di tipo EE_ START

Da questo momento il server e la rete sono sincronizzati e i sensori aspettano soltanto la richiesta di esecuzione di una query per costruire le eventuali connessioni e inviare i dati rilevati.

3.7 Protocollo di funzionamento a regime della rete

All'avvio ogni nodo alloca un canale speciale nella propria tabella delle connessioni: questo garantisce che periodicamente e per un piccolo periodo di tempo la radio rimanga sicuramente accesa, anche se è attivo il protocollo di risparmio energia.

Quando l'utente invoca i comandi di esecuzione di una query, il protocollo di funzionamento della rete si attiva: esso prevede che il server metta in coda i comandi da inviare ai sensori e li invii non appena possibile, in base al tempo di accensione della radio.

Essendo garantita una finestra temporale fissa in tutti i casi arriva un istante in cui il server può inviare ai nodi coinvolti un pacchetto: in quell'occasione è previsto l'invio di un EE_STOP. Questo messaggio serve a sospendere il protocollo di *energy efficiency* e far sì che la radio riceva tutti i comandi che ogni nodo deve ricevere. Come già detto la costruzione dei canali risulta particolarmente critica: il mancato arrivo anche di una sola connect, data l'assenza di un meccanismo di ack, causerebbe il fallimento dell'intera esecuzione. Il pacchetto EE_STOP è di formato analogo all'EE_START ma causa l'operazione inversa sospendendo quello che l'altro pacchetto aveva attivato. Le radio rimangono quindi accese per ricevere istruzioni sull'esecuzione della query e il server può programmare i nodi inviando loro i comandi necessari.

Figura 14: messaggio di tipo EE_STOP

Dopo l'invio dell'EE_STOP, il server invia comandi separati ai singoli nodi tramite pacchetti ASYSEND_WITH_ACK: questo tipo di messaggio prevede la ricezione di un acknowledgement prima di procedere con l'invio del successivo. Infatti è fondamentale che l'ordine di arrivo dei messaggi sia quello imposto affinché le disposizioni inviate

nella rete producano il risultato corretto e si creino i flussi per il trasferimento dei dati da un sensore ad un altro e poi verso il server.

Lo stesso formato di pacchetto è utilizzato per il tipo ASYSEND, analogo al precedente ma senza necessità di invio dell'acknowledgement: questo tipo è usato dal *sink* per inoltrare i dati ricevuti dai sensori coinvolti verso il server.

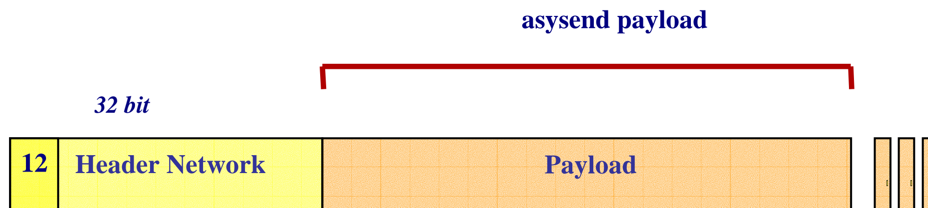

Figura 15: messaggio di tipo ASYSEND_WITH_ACK

Il server programma ogni nodo diversamente per eseguire la query: esistono diversi tipi nell'header di livello applicazione dei pacchetti: in un primo momento, in base al piano di esecuzione derivato per la query, viene inviato un comando per costruire un canale tra due nodi che devono scambiarsi dati.

A questo punto il Query Executor invoca l'operazione di connect: alla ricezione di questo tipo di pacchetto, dopo aver trasmesso l'*ack*, il nodo alloca le strutture dati e invia un pacchetto di tipo CONNECT al nodo destinatario.

Il pacchetto è formato da un header e da un payload come mostrato in figura 16: quest'ultimo è diviso in un *identificatore* di canale di 8 bit, *streamrate* che indica il valore temporale con il quale la query richiede i dati, *size* che indica la dimensione del payload e *data*, il payload vero e proprio, nel quale è contenuto il pacchetto di livello superiore.

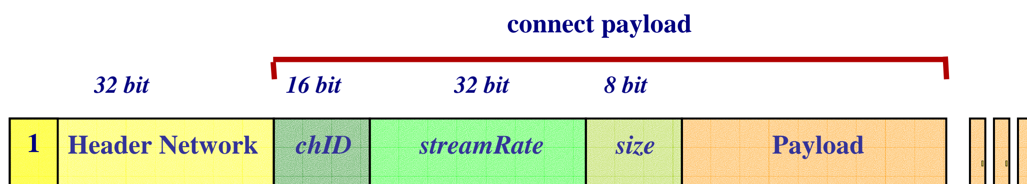

Figura 16: messaggio di tipo CONNECT

Il nodo che lo riceve alloca anch'esso le strutture dati necessarie e il canale rimane attivo fino alla sua cancellazione: al momento dello stop di una query, comandato dall'utente tramite la *gui*, il server invia un comando di chiusura canale verso il nodo che deve iniziare tale operazione. Tale comando è inviato in un pacchetto di tipo ASYSEND_WITH_ACK. Il nodo che lo riceve crea un pacchetto di tipo DISCONNECT, contenente soltanto l'*identificatore* del canale e il payload, e lo spedisce verso l'altro capo del canale. Il formato è mostrato in figura 17.

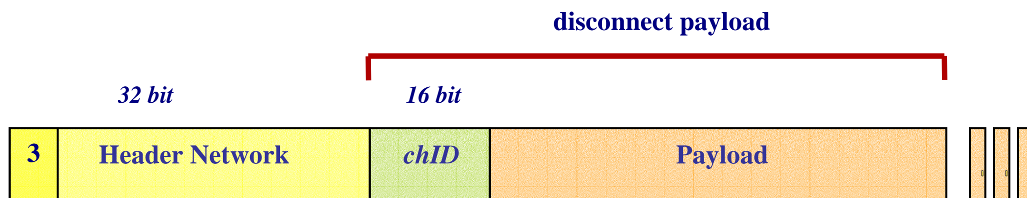

Figura 17: messaggio di tipo DISCONNECT

Gli ultimi comandi che sono inviati nella rete per avviare la query sono sempre messaggi di tipo ASYSEND_WITH_ACK con un payload che contiene il comando *StartQuery* per lo strato applicazione. Alla ricezione di tali pacchetti un nodo può iniziare a spedire periodicamente i dati richiesti: questi viaggiano sulle connessioni stabilite precedentemente, con pacchetti di tipo SEND, composti dall'*identificatore* del canale e dal payload, contenente appunto i valori rilevati. Il pacchetto SEND è mostrato in figura 18.

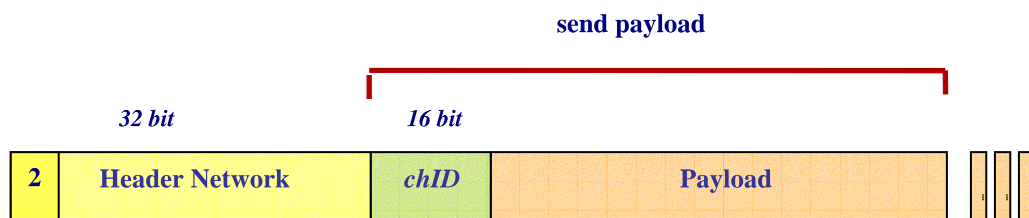

Figura 18: messaggio di tipo SEND

4 Implementazione

4.1 Overview

Il livello di rete è organizzato in due moduli: *EnergyEfficiencyM* e *NetToSSM*. Il primo si occupa del protocollo di risparmio energetico e fornisce i comandi invocabili dal modulo gestore del livello di rete, *NetToSSM*. L'interfaccia implementata tra i due moduli è *EEtoNET* e viene usata per gestire la modalità power-saving sui nodi; inoltre entrambi fanno uso di strutture dati per mantenere le informazioni riguardo ai canali passanti per ogni nodo. I dettagli sono descritti nel paragrafo 4.3.

La configurazione del network esporta verso i livelli superiori due interfacce, *Connectionless* e *ConnectionOriented*. La prima viene usata dal Query Executor, al livello applicazione, l'altra dallo Stream System, al livello di trasporto. All'interno di *NetToSSM* si utilizzano tabelle, per memorizzare i dati dei canali, e code gestite da task per l'invio e la ricezione dei messaggi.

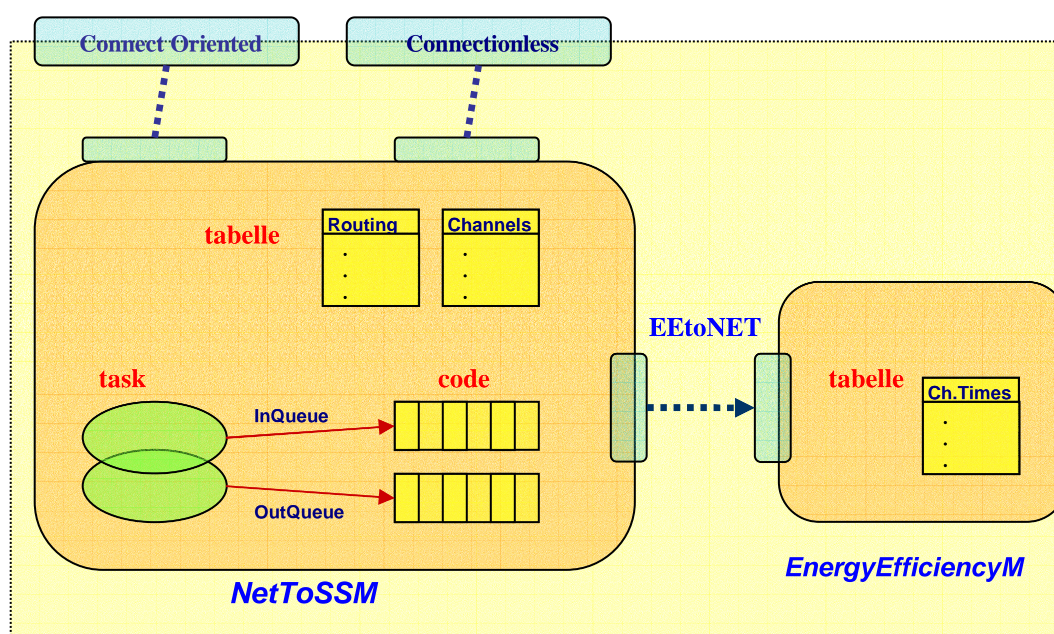

Figura 19: configurazione del livello di rete (NetToSSC)

4.2 Struttura del network

Il livello di rete garantisce l'invio e la ricezione di pacchetti attraverso funzionalità che offre ai livelli superiori, *Stream System* e *Query Processor*: i servizi realizzati verso questi due livelli sono una parte delle interfacce che il network implementa.

Le primitive disponibili sono quattro: *connect*, *disconnect*, *send*, *asysend* e ognuna realizza una specifica fase del protocollo di funzionamento. I primi tre comandi costruiscono (e si basano su) una comunicazione orientata alla connessione e fanno parte di un'interfaccia chiamata *ConnectionOriented*; il comando *asysend* invece viene utilizzato per spedire messaggi non legati ad una connessione (tipo UDP) e implementa l'altra interfaccia *Connectionless*. Il livello Stream System usa l'interfaccia Connection Oriented per inviare messaggi lungo canali di comunicazione costruiti tra i nodi, il Query Executor usa Connectionless per immettere i comandi nella rete e avviare l'esecuzione delle query.

L'operazione di *connect* viene invocata quando deve essere aperto un canale tra due nodi. I dati da passare alla primitiva sono: un intero a 16 bit che identifica l'indirizzo del nodo destinatario, un intero a 32 bit che indica il valore temporale al quale il canale dovrà garantire l'invio di dati, due interi a 8 bit che esprimono rispettivamente la lunghezza messaggio di livello Stream System e del payload del pacchetto di livello network; infine il puntatore a tale payload. Uno schema dei parametri è mostrato in tabella 4.

Nome parametro	Descrizione
dest	Nodo destinatario: è necessario indicare verso quale nodo dovrà essere costruito il canale.
streamrate	Valore temporale in millisecondi: indica ogni quanto tempo il canale avrà bisogno della radio accesa.
pktSS_size	Questo parametro non è utilizzato.
pMsg	Puntatore al payload contenente il messaggio di livello Stream System.
size	Lunghezza del payload del pacchetto

Tabella 4: parametri per la primitiva *connect*

In figura 20 è mostrato un esempio di comunicazione che avviene per una connect.

Il nodo A, che nell'esecuzione della query dovrà inviare i dati da processare sul nodo B, riceve la comunicazione di apertura di stream remoto verso quel nodo: a quel punto controlla di poter allocare una nuova entry in una tabella locale di canali e, se l'operazione riesce, aggiorna tale connessione con le informazioni sul prossimo hop per raggiungere il nodo B. Nel caso non sia possibile allocare una entry, l'operazione fallisce. L'ultimo intervento sulle strutture dati riguarda la tabella per l'energy efficiency: se tutto è andato bene si invoca un comando che provvede ad aggiornare quest'altra tabella, riservata ai tempi di accensione e spegnimento necessari per il corretto invio dei dati sui canali. Adesso il lavoro sul nodo A è terminato ed è necessario fare altrettanto sul nodo destinatario della comunicazione: un pacchetto di tipo connect viene costruito dal nodo A e inviato al nodo B, specificando le informazioni per una corretta allocazione nella tabella dei canali corrispondente.

Alla ricezione di un pacchetto di connect, un nodo alloca una entry nella tabella dei canali e una nella tabella per la gestione dell'energy efficiency: entrambe vengono aggiornate con le informazioni reperibili nel pacchetto. A questo punto, se il nodo che ha ricevuto il pacchetto non è il destinatario della connect provvede all'inoltro, modificando opportunamente l'indirizzo di livello MAC secondo la tabella di routing. Altrimenti il messaggio è arrivato alla meta e viene segnalato l'evento receive al livello superiore: il messaggio sarà gestito dallo Stream System.

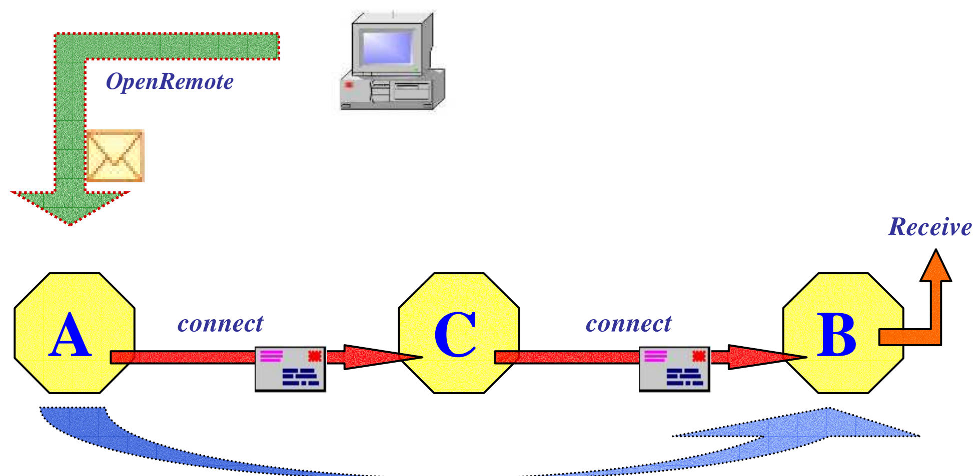

Figura 20: costruzione di un canale tra due nodi (CONNECT)

Una volta che un canale è stato costruito, la spedizione dei messaggi tra due nodi avviene con l'invocazione del comando *send*, che serve per spedire un pacchetto lungo un canale specificato. Gli argomenti da passare all'invocazione della primitiva sono: un intero a 16 bit che identifica il canale sul quale si vuol comunicare, un intero a 8 bit che esprime la lunghezza del payload del messaggio e il puntatore al messaggio stesso. I parametri sono mostrati in tabella 5.

Nome parametro	Descrizione
channel_id	Identificatore del canale sul quale si intende inviare dati con la send.
pMsg	Puntatore al payload del messaggio.
size	Lunghezza del payload del pacchetto.

Tabella 5: parametri per la primitiva *send*

In figura 21 è mostrato un esempio di comunicazione su un canale.

Il nodo A che vuole inviare un pacchetto al nodo B utilizza la send sul canale specificato al momento dell'invocazione: in questo modo il pacchetto segue la rotta indicata per quel canale e viene inviato verso il prossimo hop seguendo l'associazione <destinatario-prossimo hop> per raggiungerlo. Quando il messaggio giunge sul nodo B l'unica operazione da fare è assicurarsi che il canale indicato dal pacchetto esista e sia attivo; dopodichè si segnala allo Stream System l'evento receive e il payload del pacchetto, che contiene i dati rilevati per la query, viene gestito opportunamente per comporre il risultato o una parte di esso.

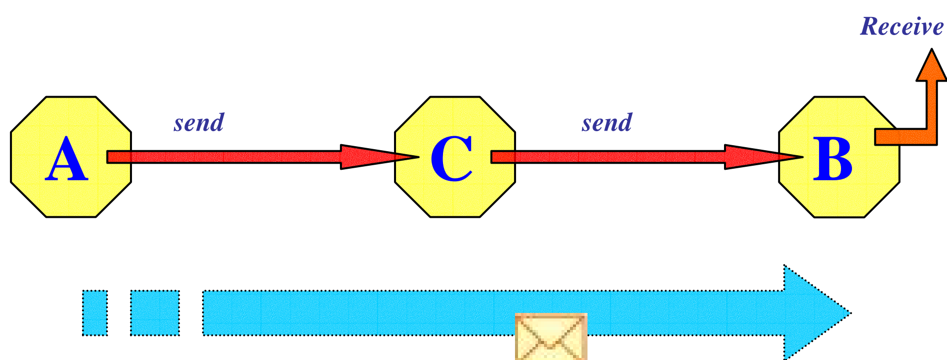

Figura 21: invio di dati lungo un canale specificato (SEND)

La comunicazione tra i nodi va avanti finchè una esecuzione di query non viene interrotta dall'utente, generando un comando dal server verso uno dei nodi per la chiusura del canale. La primitiva per realizzare tale operazione è la *disconnect* ed è sufficiente invocarla passando il valore dell'identificatore di canale da cancellare, oltre naturalmente al valore della lunghezza del payload del messaggio e il puntatore a tale payload. I parametri per la primitiva sono descritti in tabella 6.

Nome parametro	Descrizione
channel_id	Identificatore del canale che si vuole chiudere.
pMsg	Puntatore al payload del messaggio.
size	Lunghezza del payload del pacchetto.

Tabella 6: parametri per la primitiva *disconnect*

Con l'invio di un pacchetto di disconnect, il canale indicato viene recuperato e cancellato dalla tabella dei canali; l'entry diviene nuovamente disponibile per nuove connessioni. Nel caso l'operazione abbia successo si provvede a rimuovere anche l'informazione sui tempi di accensione e spegnimento per quel canale nella tabella dell'energy efficiency.

Alla ricezione lungo tutto il cammino, il messaggio provoca la cancellazione della relativa struttura dati allocata e l'eventuale inoltro, fino al nodo destinatario, sul quale avviene la segnalazione dell'evento di ricezione al livello superiore, come possiamo vedere in figura 22.

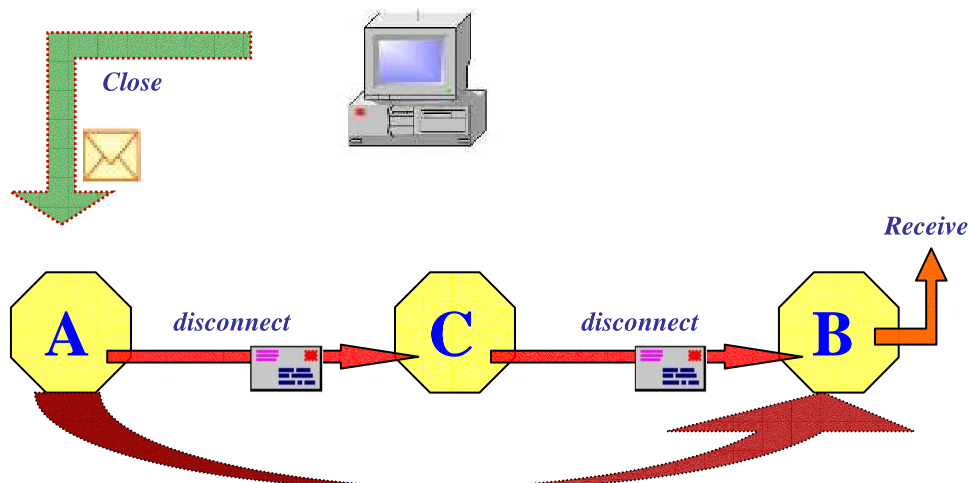

Figura 22: cancellazione di un canale tra due nodi (DISCONNECT)

L'altro comando disponibile per la comunicazione tra i nodi è l'*asysend*. Questa primitiva viene invocata direttamente dal livello del *Query Processor* e serve sia per inviare un pacchetto proveniente dal server verso i nodi sia per inoltrare i risultati di una query dal sink verso il server. L'invocazione della primitiva richiede tre argomenti: il nodo destinatario, il puntatore al messaggio e il valore della misura del suo payload, come descritto in tabella 7.

Nome parametro	Descrizione
dest	Identificatore del nodo con il quale si deve comunicare.
pMsg	Puntatore al payload del messaggio.
size	Lunghezza del payload del pacchetto.

Tabella 7: parametri per la primitiva *asysend*

In figura 23 vediamo che l'invio di un pacchetto con questo comando avviene diversamente rispetto alla send: viene seguito infatti un routing diverso da quello specificato dai canali e i pacchetti seguono la strada che è indicata dalle tabelle di rotta (inviate dal server ai singoli nodi nella fase di inizializzazione). La comunicazione di cui si tratta avviene in modo asincrono rispetto all'interazione nodi–server sui canali relativa alla query e serve principalmente per l'invio dei comandi per la programmazione dei nodi e l'avvio della query.

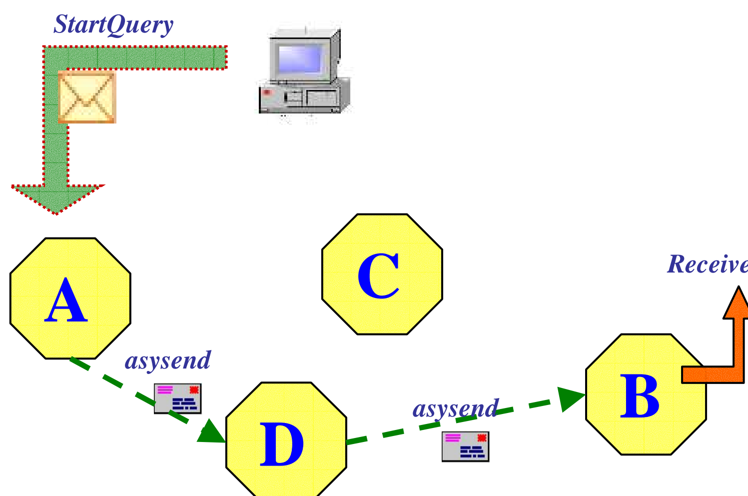

Figura 23: invio asincrono di pacchetti (ASYSEND)

4.3 Strutture dati

Su ogni nodo esistono due strutture dati principali per avere a disposizione informazioni sulla topologia della rete e quindi sulle comunicazioni coi nodi vicini: la tabella dei canali (*channel_table*) e la tabella di routing (*routing_table*).

La prima è un array di *connection* (connessioni) e serve per memorizzare su ogni nodo le connessioni presenti, così da inoltrare correttamente i pacchetti lungo i canali.

Ogni connessione è una struct di 5 byte formata da un identificatore univoco di canale, un byte che indica l'indirizzo di livello network del nodo destinatario, un byte che indica il prossimo hop per quel canale, infine un byte di cui 1 bit segnala se la connessione è attiva e gli altri 7 bit sono una chiave esterna verso un'altra tabella. Questi ultimi 7 bit sono un ulteriore identificatore che il network mantiene per la consistenza con la tabella relativa all'energy efficiency. Le strutture presenti in questa tabella sono riferite da questo campo con un identificatore di 7 bit: le operazioni di inserimento e cancellazione avvengono in conseguenza all'inserimento/cancellazione nella channel_table. In tabella 8 viene mostrata la struttura dati e i campi nel dettaglio.

Nome campo	Lunghezza	Descrizione
channel_id	16 bit	Identificatore univoco di canale nella rete: è costruito a partire dall'indirizzo del nodo sorgente e da un contatore
dest	8 bit	Indirizzo del nodo su cui il canale termina. E' possibile riferire fino a 256 possibili nodi
succ	8 bit	Prossimo hop del canale: identifica il nodo sul quale sono indirizzati i pacchetti che viaggiano sul canale verso la destinazione
validity	1 bit	Bit di controllo di validità del canale. Se è 0 significa che il canale non è attivo e non deve essere utilizzato. In questo stato può essere sovrascritto da un nuovo canale. Se è 1 significa che il canale è attivo.
ee_position	7 bit	Chiave esterna: identificatore univoco della posizione del canale nella tabella connessioni del modulo di energy efficiency. Questi 7 bit permettono di riferire fino a 128 canali allocati dal modulo *EnergyEfficiencyM*. Questo valore viene restituito dalla funzione di allocazione *set* dell'interfaccia EEtoNET.

Tabella 8: struct *connection*

L’altra struttura è la tabella dei vicini (*routing_table*) ed è un array di struct chiamate *neighbor*. Un neighbor è composto da 2 bytes: il primo è suddiviso in 1 bit che indica la validità dell’entry e i successivi 7 bit che sono l’identificatore del vicino; il byte che segue è suddiviso alla stessa maniera e il primo bit indica se il link relativo è bidirezionale, mentre i restanti 7 bit sono l’identificatore del prossimo hop, ovvero del nodo a cui inviare nel caso il destinatario di un pacchetto sia quello relativo alla entry. I campi della struttura sono descritti specificamente in tabella 9.

Nome campo	Lunghezza	Descrizione
validity	1 bit	Bit di validità del link con il nodo vicino. Se 0 indica un link non valido o non più esistente; se 1 significa che il collegamento esiste ed è attivo.
id	7 bit	Identificatore del nodo vicino: è possibile riferire fino a 128 vicini per ogni nodo.
bidir_link	1 bit	Bit che indica se il link con il vicino è bidirezionale. Se 0 significa che il nodo può ricevere dal vicino *id* ma non necessariamente quello riceve dal nodo. Se 1 significa che il nodo e il vicino *id* hanno un link bidirezionale e si “sentono” reciprocamente.
next_hop	7 bit	Identificatore del nodo verso il quale devono essere indirizzati i pacchetti con destinazione *id*.

Tabella 9: struct *neighbor*

La tabella ha due funzioni a runtime: la prima è conservare le informazioni relative ai link con gli altri sensori per la fase di *neighbor discovery* e per la fase di costruzione dell’albero logico della rete; la seconda è funzionare da tabella di routing, una volta che il server ha inviato le informazioni riguardo alle rotte verso tutti gli altri nodi. Durante l’inizializzazione della rete la tabella conserva i dati dai soli vicini da cui riceve pacchetti. Pertanto la sua funzione è quella di aiutare alla costruzione del pacchetto Neighbor_report da inviare verso il server per contribuire alla composizione della tabella di routing. Il campo next_hop non ha utilità in questa fase. Successivamente, dopo aver ricevuto la routing_table, il nodo aggiorna completamente la tabella dei vicini inserendo i prossimi hop da compiere per raggiungere qualsiasi altro nodo.

Oltre a queste due tabelle esiste un altro tipo di struttura dati utilizzata nel network: è la coda (*queue*), definita come un array di messaggi di livello MAC (*TOS_Msg*), un campo *lunghezza* di un byte e un campo *head* anch'esso di un byte. I parametri della *queue* sono descritti in tabella 10.

Nome campo	Lunghezza	Descrizione
elem []	8 bit	Puntatore all'array: permette di riferire l'elemento desiderato della coda
head	8 bit	Indice della posizione dell'elemento di testa nella coda
length	8 bit	Lunghezza della coda

Tabella 10: struct *Queue*

Il network fa uso di code circolari: una per l'invio, *OutQueue*, una per la ricezione, *InQueue,* e una per la composizione dei messaggi, *FreeQueue*. Tutte e tre vengono modificate attraverso operazioni di inserzione (*enqueue*) e estrazione (*dequeue*) per inserire o togliere elementi dalla coda.

Ogni volta che si deve inviare un messaggio, si estrae un puntatore dalla coda libera (*FreeQueue*) e si costruisce il pacchetto; dopodichè, se non si incontrano situazioni di fallimento, si inserisce il puntatore al messaggio in questione nella coda dei messaggi in uscita (*OutQueue*). Se durante le operazioni sopra descritte avviene un fallimento e il messaggio non deve più essere inviato, si provvede a reinserire nella coda *FreeQueue* il puntatore al messaggio estratto precedentemente.

Attualmente non sono previste operazioni di estrazione messaggi dalla coda in uscita (*OutQueue*). Tali operazioni vengono fatte sulla *InQueue* nel momento in cui si deve analizzare il contenuto di un messaggio in arrivo. L'operazione di inserzione nella InQueue viene effettuata nell'handler dell'evento receive nel caso il messaggio sia da passare ai livelli superiori. L'operazione di estrazione risulta critica in quanto può restituire un puntatore ad un elemento nullo nel caso sia effettuata in una situazione di coda vuota e può portare a situazioni di fallimento: per questo è stata introdotta la

variabile esterna *pending* che contribuisce a gestire l'utilizzo e le modifiche sulla coda, segnalando quando è in corso un'operazione. Questo flag viene modificato soltanto dalla procedura che si occupa di modificare le code, alla ricezione o alla spedizione di un messaggio: le procedure in questione sono i *tasks*.

4.4 Tasks

Le code che gestiscono i messaggi in ingresso e in uscita da un nodo devono essere a loro volta amministrate correttamente per regolarne l'accesso e l'utilizzo: in particolare è essenziale che eventi asincroni non interrompano o modifichino un'operazione in corso su un elemento di una coda. Si conviene inoltre che la segnalazione di eventi avvenga in modo indivisibile rispetto ad altri eventi, per evitare il rischio che alcune strutture dati siano modificate.

La coda dei messaggi in arrivo, InQueue, è gestita dall'*InQueueTask* mentre quella dei messaggi in uscita, OutQueue, dall'*OutQueueTask*. Le istanze di queste particolari funzioni non possono essere eseguite in contemporanea realizzando di fatto una mutua esclusione fra task, indispensabile per la corretta elaborazione di alcune operazioni critiche sui messaggi.

L'InQueueTask viene chiamato ogni volta che si deve segnalare l'evento di ricezione ai livelli superiori. L'algoritmo è semplice: prende l'elemento nella posizione di testa della coda dei messaggi ricevuti e segnala l'evento su un'interfaccia, passando come argomento il puntatore al messaggio. Essendo questo algoritmo eseguito all'interno di un task, si ha la certezza che non sarà interrotto da altri task e il messaggio controllato non sarà modificato mentre l'operazione è in corso. L'InQueueTask viene invocato anche per segnalare l'evento *connectDone*: questa scelta è dovuta al fatto che attualmente non esiste un protocollo che prevede l'acknowledgement per i messaggi di connect; di conseguenza l'evento connectDone da segnalare deve essere simulato al momento dell'invio di un

messaggio di connect. In corrispondenza della spedizione di un pacchetto di questo tipo emesso dal nodo stesso si genera perciò anche una segnalazione di connectDone allo Stream System.

L'OutQueueTask invece si occupa della spedizione dei messaggi: l'unico compito a cui adempie è l'estrazione di un messaggio dalla coda e la successiva chiamata del comando *send* dell'interfaccia *SendMsg*. Questa primitiva viene fornita dal modulo GenericComm e consente la trasmissione di un pacchetto sulla radio. Prima di realizzare l'invio vero e proprio il task pone la variabile *pending* a *true* per segnalare l'accesso bloccato alla struttura dati. Questo flag funziona da semaforo per la coda e verrà rimessa a *false* dall'handler dell'evento *sendDone*, generato verso il network una volta terminata la trasmissione oppure in caso di fallimento della chiamata, al fine di lasciare nuovamente libero l'accesso alla coda.

4.5 Energy Efficiency

Il meccanismo di energy efficiency attua una gestione della radio mirata al risparmio di energia. L'idea principale è quella di memorizzare gli istanti di tempo in cui i canali sul nodo dovranno essere attivi, così da cercare eventuali finestre temporali in cui non occorre che la radio sia accesa per tali canali: in questo modo si intende sfruttare questi momenti per risparmiare la batteria dei sensori. Nel network è prevista una tabella di canali con i dati per il routing; nel modulo di energy efficiency si conservano le informazioni sui tempi in cui i vari canali sfruttano la radio per inviare o ricevere messaggi. La tabella *channel_table* viene riempita con una nuova entry ogni volta che una nuova connessione viene creata nel network.

Le entry di questa tabella sono chiamate *channel* e sono struct di 40 bytes contenenti 3 campi con valori temporali assoluti di 8 bytes ciascuno, corrispondenti rispettivamente a tempo di *accensione*, *spegnimento* e *terminazione* del canale; inoltre si hanno due valori

di 4 bytes ciascuno rappresentanti la *durata* e il tempo di *interarrivo* dei dati su quel canale e infine un bit di validità. I valori *nextON* e *nextOFF* variano a tempo di esecuzione e sono aggiornati in base al tempo corrente sui sensori. La variabile *interarrive* deriva dalla query richiesta mentre *duration* è determinata in base a esperimenti e prove e misurabile attualmente in circa mezzo secondo. In realtà quest'ultima dovrebbe essere modificata in base alla lunghezza di un messaggio (più un messaggio è grande, maggiore è il tempo di trasmissione). Infine la variabile *expiry*, che pur essendo attualmente inutilizzata, va considerata nel caso il canale debba invalidarsi automaticamente a un certo istante di tempo. I campi della struct *channel* sono schematizzati in tabella 11.

Nome campo	Lunghezza	Descrizione
active	8 bit	Bit di validità del canale. Se 0 il canale non è da considerare. In caso contrario i tempi relativi a accensione e spegnimento saranno considerati validi.
duration	32 bit	Valore temporale in millisecondi che indica quanto la radio starà accesa per permettere al canale di inviare/ricevere dati.
interarrive	32 bit	Valore temporale in millisecondi che indica ogni quanto tempo il canale avrà bisogno della radio accesa per inviare/ricevere dati.
nextON	64 bit	Valore temporale in millisecondi che indica il prossimo istante assoluto in cui la radio dovrà riaccendersi per far si che il canale invii o riceva dati.
nextOFF	64 bit	Valore temporale in millisecondi che indica il prossimo istante assoluto in cui la radio potrà spegnersi in quanto il canale non deve inviare o ricevere dati.
expiry	64 bit	Campo attualmente inutilizzato. Indica l'istante di tempo in cui il canale dovrà cessare di esistere.

Tabella 11: struct *channel*

Questa struttura è sufficiente a raccogliere i dati necessari per il nostro obiettivo. Il bit di validità infatti indica se il canale è da tenere in considerazione oppure no; il valore della durata e dell'interarrivo, riferiti ai rilevamenti periodici che un sensore deve effettuare, ci permettono di costruire uno schema, variabile nel tempo, che ci mostra eventuali periodi durante i quali la radio del sensore può rimanere spenta senza perdere alcuna comunicazione.

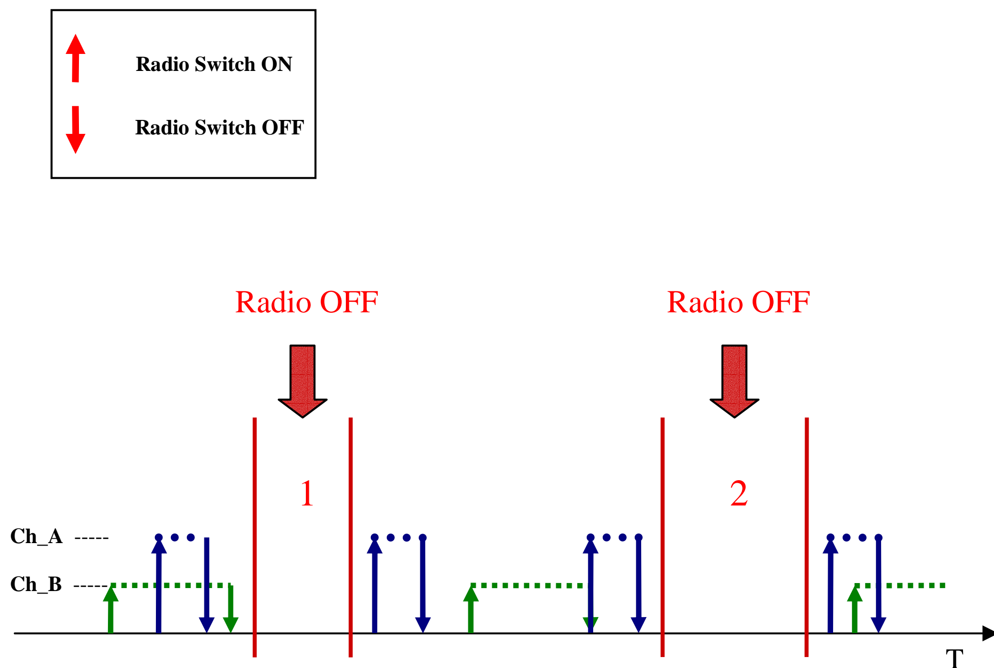

Figura 24: esempio di schema di analisi per energy efficiency

In figura 24 è descritto un esempio in cui i canali presenti sul nodo sono due: ch_A e ch_B. Ciascuno ha i propri tempi necessari di radio ON e radio OFF, ovvero i tempi nei quali la radio dovrà stare accesa o spegnersi per usufruire di quel canale. Le frecce che vanno verso l'alto indicano l'accensione, quelle verso il basso lo spegnimento. Le linee tratteggiate significano che lo stato della radio non cambia in quel periodo di tempo.

4.5.1 Interfaccia EEtoNET

Come possiamo vedere in figura 24 i tempi in cui la radio deve stare accesa per i canali si possono intersecare e lasciare altri intervalli vuoti: l'algoritmo cerca quegli istanti di tempo in cui non vi sia la necessità di radio accesa e li usa per spegnere il controller. In figura 24 queste eventuali pause sono limitate da due linee verticali che evidenziano rispettivamente i momenti in cui la radio non serve e in cui sarà invece necessario riaccenderla. Come possiamo notare, gli istanti in cui si modifica lo stato della ricetrasmittente non coincidono precisamente con gli istanti in cui i canali non ne hanno più bisogno: questa soglia di sicurezza, fissata attualmente nell'ordine di un secondo, anticipa le accensioni programmate e posticipa gli spegnimenti al fine di garantire che eventuali ritardi sulla rete non facciano perdere alcun messaggio. Il parametro di correzione è espresso in millisecondi ed è modificabile con il comando *setRadioDelay* dell'interfaccia *EEtoNET*: se per esempio è impostato a 1000 ms e un nodo deve accendere la radio all'istante $t1$=15000 e spegnerla all'istante $t2$=20000, le reali accensione e spegnimento della radio avverranno rispettivamente all'istante $t1$=14000 e $t2$=21000.

Un secondo accorgimento prevede che se il periodo che intercorre tra uno spegnimento e una riaccensione è troppo breve, e precisamente minore di una tolleranza fissata, la radio viene lasciata accesa in quanto il risparmio ricavato sarebbe praticamente nullo. Anche questa tolleranza si può impostare dall'esterno del modulo, con la chiamata al comando *setTolerance*: se per esempio viene impostato un valore uguale a k (di default è 1 secondo) la radio non viene spenta in tutti quei casi in cui la riaccensione sarebbe programmata entro k millisecondi.

Questi due comandi fanno parte dell'interfaccia *EEtoNET*, implementata dall'energy efficiency verso il network. Essa mette a disposizione anche le primitive *startEE* e *stopEE* con le quali si fa partire e si sospende l'algoritmo di power saving.

Al momento dell'invocazione di startEE si considerano i canali presenti sul nodo e si ricercano possibili intervalli di spegnimento. Tutto l'algoritmo viene coordinato attraverso timer che vengono attivati in corrispondenza dell'accensione della radio e allo scadere dei quali si analizza la situazione. Al momento della chiamata del comando stopEE invece la radio si accende e così rimane fino ad una nuova invocazione di startEE. Ogni volta che un canale viene inserito dal network nella tabella dei canali viene creata una nuova entry in una tabella tramite la chiamata al comando *set*: l'aggiornamento dei tempi del canale avviene in base al tempo corrente e le altre informazioni sono prese direttamente dalla chiamata alla connect del network, in quanto presenti nella query stessa. In tabella 12 sono descritti i parametri per questa primitiva.

Nome campo	Lunghezza	Descrizione
duration	32 bit	Durata del canale: indica quanto deve stare accesa la radio per permettere al canale in questione di inviare e ricevere dati
interarrive	32 bit	Interarrivo: questo valore indica ogni quanto la radio dovrà riaccendersi per far funzionare il canale
expiry	64 bit	Valore temporale di terminazione del canale: attualmente non previsto dall'applicazione

Tabella 12: parametri per la primitiva *set*

La primitiva *setSynch* ha il solito compito ma viene invocata all'inizializzazione per creare un canale di sincronizzazione con il server: anche se ci troviamo in regime di energy efficiency la radio non si spegne mai per questo periodo. Tale connessione è fissa per ogni nodo e non può essere rimossa; tuttavia è possibile riprogrammarla con una nuova chiamata al comando setSynch.

Tutti gli altri canali invece possono essere invalidati con il comando *reset* che ne azzera il bit di validità rendendoli sovrascrivibili appena questo sarà necessario: infatti se un canale ha il bit di validità uguale a zero è come se non esistesse e la relativa entry della

tabella può essere modificata coi valori di un nuovo canale. Infine il comando *resetAll* annulla in un'unica esecuzione tutti i canali eccetto quello di sincronizzazione.

L'interfaccia EEtoNET non prevede eventi: il modulo EnergyEfficiencyM la fornisce all'esterno implementando i comandi e il modulo NetToSSM la utilizza chiamando le primitive disponibili.

4.5.2 Algoritmo OffTimer

L'algoritmo che seleziona il tempo esatto per modificare lo stato della radio è invocato allo scadere di un timer: all'avvio del modulo di energy efficiency questo timer scatta e l'algoritmo imposta il primo istante da analizzare.

Il compito spetta alla funzione *OffTimer*: questa innanzitutto provvede ad aggiornare i valori presenti nella tabella in base al tempo corrente e poi considera l'istante di spegnimento più grande tra i dati relativi ai canali attivi sul nodo, prendendo in considerazione soltanto quelli minori rispetto ad un certo valore, che chiamiamo *nextON*, corrispondente al minimo dei tempi di prossima accensione per i canali attivi. Questi tempi sono calcolati addizionando i tempi di interarrivo ai tempi assoluti *proxOn*, relativi agli istanti passati di accensione.

In questo modo viene selezionato un valore *offTime* "candidato" e l'indice del canale relativo a tale istante di spegnimento. A questo punto esistono diversi casi risultanti.

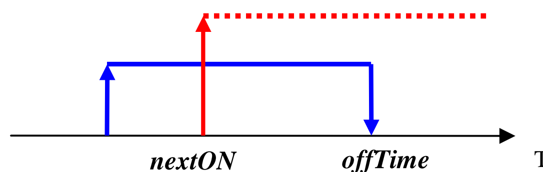

Figura 25: algoritmo OffTimer (caso 1)

Nel primo caso, mostrato in figura 25, abbiamo *offTime* > *nextOn* e la radio non deve essere spenta, a causa di un altro canale. Le variabili vengono aggiornate e il timer per la programmazione dello spegnimento viene posticipato.

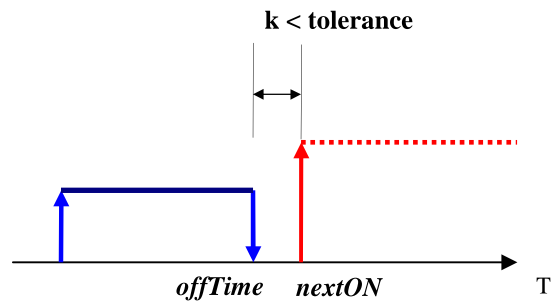

Figura 26: algoritmo OffTimer (caso 2)

Nel secondo caso, mostrato in figura 26, troviamo *nextOn* > *offTime* ma la differenza tra i due valori è minima, e in particolare è minore della tolleranza prevista. Il comportamento che si segue è analogo a quello del caso 1.

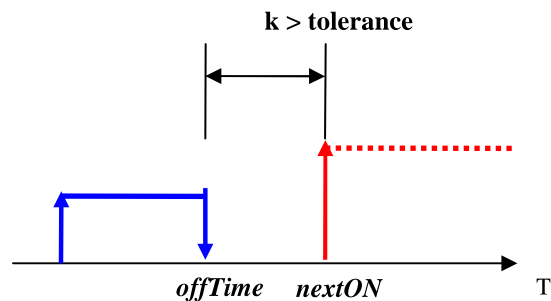

Figura 27: algoritmo OffTimer (caso 3)

Infine l'ultima situazione, mostrata in figura 27: *nextOn* > *offTime* e la differenza tra i valori è abbastanza grande. In questo caso si procede allo spegnimento della radio, si aggiornano le variabili e si fa partire un nuovo timer che dovrà scadere all'istante *nextOn*

trovato. L’evento sollevato alla scadenza di questo timer provocherà la riaccensione della radio e un nuovo aggiornamento dei tempi.

Questi tre casi coprono tutte le eventuali situazioni derivate dall’analisi dell’algoritmo: in tutti i casi OffTimer restituisce come risultato un valore temporale e avvia un apposito timer, sia che la radio debba essere spenta, sia che lo spegnimento debba essere posticipato, sia che la radio vada riaccesa dopo un periodo di stop.

5 Conclusioni

La tesi ha sviluppato e portato a completamento due compiti importanti per un livello di rete: il primo è la gestione multi-hop della rete stessa, che permette di estendere un gruppo di sensori ben oltre il raggio di copertura di ognuno di essi; il secondo, più specifico per le reti di sensori, è la gestione del risparmio energetico sui nodi.

Il lavoro svolto è iniziato da uno strato di rete single-hop ed ha avuto come obiettivo la realizzazione di uno strato di rete reale ed efficiente multi-hop. Le funzionalità sono state testate con il simulatore Tossim [1] per un ampia gamma di query di prova: grazie alle primitive implementate adesso è possibile costruire una rete con un grafo connesso qualsiasi. Inoltre i test effettuati su sensori Mica2 e MicaZ [2] hanno dato esito positivo e confermato la correttezza delle soluzioni trovate.

Per quanto riguarda il risparmio della batteria è stato studiato un algoritmo per calcolare i tempi adatti agli spegnimenti della radio e le problematiche derivanti. I metodi utilizzati hanno permesso di sfruttare al meglio gli elementi già esistenti del TinyOS per cercare di limitare al massimo l'uso di energia sui sensori, risparmiando batteria sui dispositivi di rilevazione, sul processore e naturalmente sulla radio. I test svolti sul simulatore hanno dati esiti positivi; analogamente le prove effettuate sui sensori su query semplici hanno messo in evidenza un risparmio considerevole di energia, per i periodi in cui la radio rimane spenta. Questo risparmio è dovuto principalmente ai comandi impiegati per ridurre al minimo i consumi durante lo stato di *sleep* della radio e all'algoritmo, di cui si è parlato in precedenza, che analizza i *rate* degli stream e si occupa degli spegnimenti periodici dei dispositivi. Si prevede quindi un consistente allungamento del tempo di vita medio del sensore nei casi di utilizzo tipici del MaD-WiSe, ovvero con rilevazioni

periodiche di dati con lunghi periodi di inattività: in queste circostanze infatti la radio si trova per lo più spenta e il risparmio energetico viene sfruttato consistentemente.

Indice tabelle

Header di livello MAC....13
Header di livello Network 14
Tipi di messaggio di livello Network....15
Tabella parametri per la primitiva *connect*....23
Tabella parametri per la primitiva *send*....25
Tabella parametri per la primitiva *disconnect* 26
Tabella parametri per la primitiva *asysend* 27
Struct *connection* 28
Struct *neighbor*....29
Struct *queue* 30
Struct *channel*....33
Tabella parametri per la primitiva *set* 36

Indice figure

Struttura dell'applicazione Mad-WiSe 4
Stack del Mad-WiSe 5
Schema di un canale nella rete 10
Interfaccia Connection Oriented (Network - Stream System) 11
Interfaccia Connectionless (Network - Query Executor) 12
Header di livello Network 15
Messaggio di tipo SETCLOCK 16
Messaggio di tipo ND_START 16
Messaggio di tipo HELLO 17
Messaggio di tipo BUILDTREE 17
Messaggio di tipo NEIGHBOR_REPORT 18
Messaggio di tipo ROUTING_TABLE 18
Messaggio di tipo EE_START 18
Messaggio di tipo EE_STOP 19
Messaggio di tipo ASYSEND 20
Messaggio di tipo CONNECT 20
Messaggio di tipo DISCONNECT 21
Messaggio di tipo SEND 21
Configurazione del livello di rete (NetToSSC) 22
Costruzione di un canale tra due nodi (CONNECT) 24
Invio di dati lungo un canale specifico (SEND) 25
Cancellazione di un canale tra due nodi (DISCONNECT) 26
Invio asincrono di pacchetti (ASYSEND) 27
Esempio di schema di analisi per l'Energy Efficiency 34
Algoritmo OffTimer (caso 1) 37
Algoritmo OffTimer (caso 2) 38
Algoritmo OffTimer (caso 3) 38

Bibliografia

[1] Philip Levis and Nelson Lee, "TOSSIM: A Simulator for TinyOS Networks", September 2003

[2] Santosh Kumar, Anish Arora, OSU, Young-ri Choi, Mohamed Gouda, University of Texas at Austin, "Power Management Implementation", January 2005

[3] G. Amato, P. Baronti, S. Chessa, "MaD-WiSe: Programming and Accessing Data in a Wireless Sensor Network", November 2005

[4] S. Chessa, P. Maestrini, "Dependable and Secure Data Storage and Retrieval in Mobile, Wireless Networks"

[5] G. Amato, P. Baronti, A. Caruso, S. Chessa, "Application-Driven, Energy-Efficient Communication in Wireless Sensor Networks"