UNIVERSITÀ DEGLI STUDI DI PISA
Facoltà di Scienze, Matematiche, Fisiche e Naturali

Corso di Laurea in
Tecnologie Informatiche

Tesi di laurea

Reingegnerizzazione e Routing Multi-Hop nel sistema per la gestione di dati in reti di sensori Mad-WiSe

Relatori:
Prof. Stefano Chessa

Dott. Giuseppe Amato

Controrelatrice:
Prof.ssa Maria Grazia Scutellà

Candidato:
Federico Vincenzo Antonio Porceddu

A.A. 2007/2008

Indice

Capitolo 1

Introduzione

Questo lavoro nasce dall'esigenza dei responsabili del progetto Mad-WiSe [10] (Management of Data in Wireless Sensor networks), che vede collaborare il laboratorio "Wireless Networks and Multimedia Networked Information Systems" dell' istituto ISTI del CNR di Pisa con il Dipartimento di Informatica dell'Università di Pisa, di dare continuità al proprio progetto in seguito al rilascio da parte dell' Università della California di Berkeley della versione 2.x di TinyOs [11], il sistema operativo utilizzato dai nodi delle reti wireless di sensori. Il lavoro di tesi svolto tratta la reingegnerizzazione da TinyOs 1.x a TinyOs 2.x e l'implementazione della funzionalità di Multi-hop Routing su Mad-WiSe, un sistema [1,5,7] di trattamento di interrogazioni complesse per reti di sensori, finalizzato a rendere l'utilizzo di questa tecnologia accessibile a tutti, attraverso l'utilizzo di un linguaggio di query SQL-like.

La tesi si articola in sette capitoli: il primo è dedicato a una breve esposizione delle tecnologie e dei lavori correlati al progetto. Nel secondo capitolo è esposta la struttura di Mad-WiSe, del suo stack protocollare e delle funzionalità offerte da ogni livello. Nel terzo capitolo si affronta la questione della reingegnerizzazione di Mad-WiSe al TinyOs 2.x soffermandosi sulle differenze concettuali e sulle funzionalità introdotte e modificate rispetto al TinyOs 1.x. Il quarto capitolo spiega come è stato implementato il Multi-hop Routing adattando l'algoritmo AODV [3] al livello di rete già esistente. Gli ultimi tre capitoli, il quinto, il sesto e il settimo consistono rispettivamente nelle conclusioni, l'elenco delle figure e tabelle e in una serie di note bibliografiche.

1.1 Wireless Sensor Networks

Una rete di sensori [7] è un insieme di microcontrollori, disposti nelle vicinanze o all'interno del fenomeno da osservare, dotati di componenti in grado di rilevare grandezze fisiche (temperatura, luminosità, accelerazione, campo magnetico, ecc), capaci di comunicare tra di loro tramite tecnologia wireless a raggio limitato. Ogni nodo è autonomo dal punto di vista energetico in quanto provvisto di batterie.

L'utilizzo di componenti a basso consumo energetico in fase di costruzione dei nodi e in fase di sviluppo dell'applicazione, l'introduzione di particolari accorgimenti (utilizzo ponderato delle comunicazioni radio), permette di ottimizzare il consumo energetico dei nodi in modo tale da garantire un periodo di regime lavorativo più ampio. Il nodo è costituito da un microprocessore e da una serie di trasduttori in grado di rilevare diverse grandezze fisiche dall'ambiente circostante: la sua dotazione hardware determina le potenzialità delle reti di sensori in quanto consente di trattare dati elaborati e raffinati da ogni singolo nodo e non dati *grezzi.* All'interno di una rete di sensori, viene assegnato a un nodo il ruolo di *sink*: esso ha funzione di gateway nella rete di sensori. Attraverso l'interfaccia seriale il *sink* è connesso a un normale Personal Computer. Un utente può mandare messaggi alla rete di sensori, che verranno recapitati al *sink* tramite interfaccia seriale. Il *sink* potrà inoltrare via radio il messaggio all'effettivo destinatario, per esempio un comando o un parametro di configurazione. Viceversa ogni nodo può comunicare con il *sink* attraverso la radio e il nodo *sink* può recapitare il messaggio a un eventuale applicazione sul Personal Computer attraverso l'interfaccia seriale, per esempio il risultato di un campionamento di un determinato parametro ambientale. Il numero di nodi che costituisce una rete varia da poche unità fino a migliaia di unità, a seconda della dimensione dell'area che si vuole monitorare e in base al

livello di accuratezza. Mediamente la copertura radio consente la trasmissione in campo aperto per qualche centinaio di metri. Nel caso in cui la distanza tra due nodi vicini sia inferiore alla portata della radio, la comunicazione tra una qualsiasi coppia di nodi della rete è assicurata dall'introduzione di un protocollo di rete Multi-Hop: ogni nodo presente nel percorso tra il mittente e il destinatario assume funzionalità di routing inoltrando verso il destinatario i messaggi oggetto della comunicazione.

Un altro aspetto di cui bisogna tenere conto riguarda il potenziale mal funzionamento di un nodo della rete in seguito a fattori esterni o ambientali come l'esaurimento delle batterie, urti accidentali, condizioni climatiche avverse (piogge, temperature elevate, ecc) o interferenze elettromagnetiche che rendono difficoltose, a volte impossibili, le comunicazioni radio. In questo caso è opportuno che uno o più nodi della rete prendano in carico il lavoro che i rispettivi nodi malfunzionanti non sono più in grado di svolgere consentendo che il sistema resti sempre operativo e funzionante.

Dal punto di vista delle applicazioni, le reti wireless di sensori possono avere largo impiego in ogni campo dove sorge la necessità di monitorare grandezze fisiche quali temperatura, umidità, campi magnetici, luminosità, accelerazione di un corpo, ecc … . Per esempio, nel campo della Protezione Civile o Protezione Ambientale, trovano applicazione nella prevenzione di incendi boschivi, terremoti o inondazioni: a questo proposito è da sottolineare che le versioni precedenti dell'applicazione oggetto di questa tesi sono state testate, insieme ai Vigili del Fuoco, come strumento integrato nei dispositivi di protezione individuale degli operatori al fine di monitorare valori rilevanti in un'esercitazione. Altre applicazioni si possono trovare in ambito industriale o commerciale, in fase di monitoraggio di un impianto o di una linea di produzione del tipo "a catena di montaggio". Anche la domotica fa largo uso di questa tecnologia: si può pensare a una rete di sensori all'interno di

un abitazione domestica, collegata alla caldaia, che controlla la temperatura di ogni stanza, la presenza di persone all'interno delle stanze per gestire l'accensione e lo spegnimento dell'illuminazione artificiale o, semplicemente, il livello di illuminazione. Le reti di sensori possono essere impiegate anche in ambito medico consentendo per esempio il controllo a distanza dei valori vitali di un paziente. Ancora, in ambito militare, si può ipotizzare un uso mirato a sorvegliare particolari aree o a gestire e coordinare il movimento di mezzi sul territorio.

1.2 TinyDB

TinyDB [12] è un middleware per il management di dati in una rete di sensori. Concettualmente si astrae dall'esistenza della rete che viene vista come un database. L'accesso avviene attraverso un linguaggio dichiarativo SQL-like. L'interazione da parte dell'utente della rete avviene mediante un'interfaccia grafica. In alternativa è possibile interfacciare la propria applicazione con la rete attraverso una serie di api Java di natura non molto diversa da JDBC. Il concetto di query si discosta leggermente da quello dei DBMS tradizionali: in TinyDB la query viene iniettata nella rete e rimane attiva per un tempo specificato. In questo periodo i sensori ritrasmettono i loro dati a intervalli di tempo precisi, quindi una query aggiorna i dati in possesso dell'utilizzatore a intervalli costanti anziché una sola volta. Il sistema è sviluppato per sensori motes sotto forma di applicazione TinyOS implementata in nesC [8].

Le caratteristiche di TinyDB

Le caratteristiche salienti di TinyDB sono:

• Gestione della topologia di rete. TinyDB gestisce la topologia della rete sottostante mantenendo delle tabelle di routing. Per ogni query viene costruita una struttura ad albero che permette di inviare all'utilizzatore i dati da tutti i sensori. Viene inoltre gestita la dinamicità della rete in modo trasparente all'utente.

• Query multiple. È possibile eseguire più query contemporanee sulla stessa rete; esse vengono eseguite in maniera del tutto indipendente. E' inoltre possibile coinvolgere diversi sottoinsiemi dei sensori della rete.

• Gestione dei metadati. Il framework mantiene un elenco di metadati che caratterizzano i vari tipi di rilevazione che possono essere ottenuti dai sensori.

Architettura di TinyDB

TinyDB è composto da due applicazioni: il software eseguito sui singoli sensori (sviluppato in nesC) e l'interfaccia client (attualmente sviluppata in Java). Il flusso dell'applicazione (fig. 1) tipicamente ha inizio quando il client invia una query al nodo sink, collegato tramite seriale al Personal Computer. Nel sink viene attuato il parsing della query, inoltre la query viene ottimizzata per l'esecuzione. La query, a questo punto, viene diffusa a tutti i sensori della rete tramite la costruzione di una struttura di routing ad albero che ha come radice il nodo sink. Ogni nodo provvede a rilevare i dati dai propri sensori, propagare le informazioni che arrivano dai nodi successivi ed eventualmente aggregare le informazioni

prima di propagarle. Quando tutte le informazioni sono giunte al nodo radice la query termina e il risultato viene visualizzato nell'interfaccia Java.

L'applicazione di ogni singolo sensore è strutturata nei seguenti componenti:

• **Sensor catalog**: questo componente tiene traccia del set delle misure disponibili e di alcune proprietà (per esempio il sensore padre nella struttura ad albero usata per il routing) che caratterizzano il singolo sensore.

• **Query processor**: questo componente svolge un compito fondamentale durante l'esecuzione di una query: riceve i dati rilevati dall'hardware e dai sensori figli, aggrega e filtra tali dati e li rispedisce al sensore padre a intervalli di tempo specificati nella query stessa.

• **Network topology manager**: questo componente fornisce un'interfaccia in grado di interagire con diversi componenti sottostanti per il routing; inoltre esso offre le primitive per inviare una query e risultati e gli eventi per ricevere query e risultati.

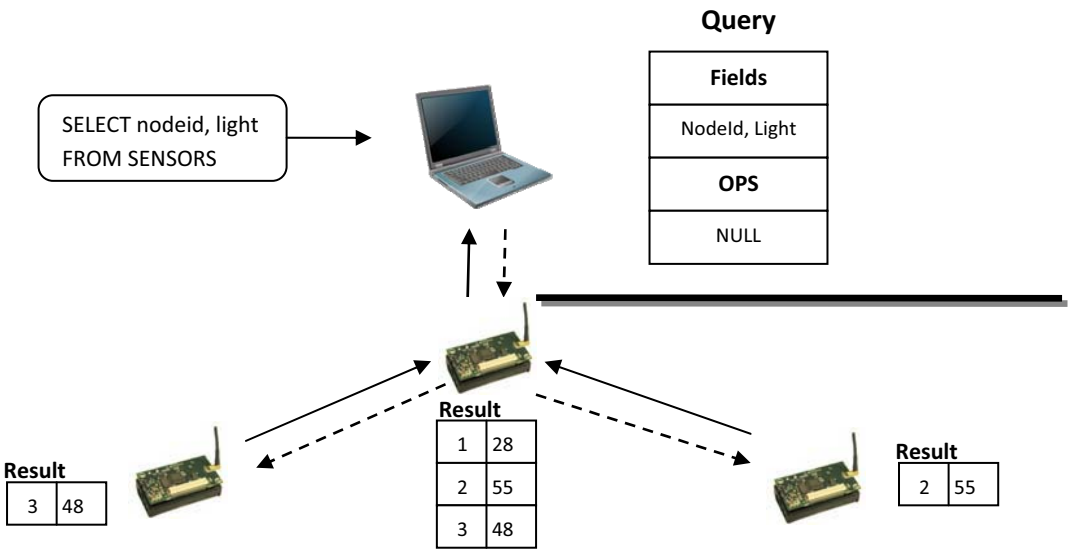

Figura 1. TinyDB Query Processing Model

1.3 Mad-WiSe

MaD-WiSe (Management of Data in Wireless Sensor networks) [10] è un progetto nato dalla collaborazione tra il laboratorio “Wireless Networks and Multimedia Networked Information System” dell’istituto ISTI del CNR di Pisa e il dipartimento di Informatica dell’Università di Pisa. L’obiettivo del progetto è quello di gestire l'interazione con una rete di sensori in modo da consentire anche agli utenti non esperti di installare, configurare ed utilizzare questa tecnologia. Mad-WiSe consente di gestire una rete di sensori attraverso l’utilizzo di un linguaggio di query SQL-Like. L’astrazione di base è quella che consente di vedere la rete di sensori come un database distribuito [1,5,7], dove ogni nodo dispone di una tabella composta da colonne che corrispondono ai trasduttori disponibili sul nodo stesso (ed altre colonne per uso specifico come ad esempio contenuti di natura temporale). Mad-Wise è composto da due applicazioni principali: una installata sui sensor motes per TinyOS sviluppata in nesC [8] e un’ interfaccia grafica sviluppata in Java per Personal Computer. La lettura dei dati rilevati dai trasduttori può essere effettuata introducendo una query che viene interamente eseguita dalla rete stessa. Ciascun nodo coinvolto nella query partecipa attivamente all'esecuzione, ricevendo i record delle tabelle dei vicini che poi verranno processati localmente con gli operatori dell'algebra relazionale dichiarata dal linguaggio. L'utente, mediante l'interfaccia grafica, definisce una query che viene ottimizzata [7] ed elaborata per definire il cosiddetto *query plan*, cioè l'insieme delle operazioni che ciascun nodo deve svolgere durante l'esecuzione. Il *query plan* viene iniettato nella rete tramite il nodo *sink* collegato via seriale che provvede all’inoltro dei messaggi via radio. Ogni nodo esegue la porzione di query ad esso assegnata: da questo punto in poi, la rete è programmata ed esegue autonomamente il lavoro assegnato; ciascun nodo rileva le informazioni sull'ambiente circostante, elabora i dati acquisiti e quelli ricevuti dai vicini e li trasmette fino

al nodo *sink* che si occupa a sua volta di ritrasmetterli all'interfaccia utente perché possano essere mostrati mediante grafici e tabelle.

Tra le varie soluzioni che consentono il management di dati nelle wireless sensor networks (vedi TinyDB al paragrafo 1.2), Mad-WiSe effettua delle distinzioni tra l'acquisizione dei dati, il trasferimento dati, le fasi di trattamento dei dati. L'approccio infatti è quello di un'architettura a livelli. Ogni nodo della WSN possiede questi tre:

- Network layer [6]
- Stream system [2]
- Stream query processing [5,7]

Un'ulteriore funzionalità è offerta dal modulo *energy efficiency*. La gestione dell'energia è una questione critica: per affrontare meglio questo aspetto è stato costruito un meccanismo che consente di inserire dei periodi di radio sleep durante il ciclo di vita dei sensori. In particolare l'algoritmo fornito estende l'autonomia energetica dei nodi mantenendo inalterate le funzioni riguardanti la corretta esecuzione dell'applicazione in ogni momento.

1.4 TinyOS 2.x

TinyOS [11] è un sistema operativo open-source real-time sviluppato per le reti wireless di sensori dall'Università della California, Berkley, in collaborazione con il centro ricerche Intel. TinyOS cerca di porre rimedio ai limiti e ai vincoli stringenti delle WSN; per questo motivo, è stato concepito in modo da introdurre caratteristiche di efficiente gestione energetica, basso carico computazionale, dimensioni ridotte, modularità e supporto alla concorrenza. TinyOS non è un sistema operativo nel senso convenzionale del termine, ma si configura come un framework avanzato per applicazioni embedded. Per questo motivo esso si differenzia in modo consistente dai sistemi operativi general-purpose, integrandosi direttamente con le applicazioni sviluppate. D'altra parte, come tutti i sistemi operativi, TinyOS si occupa di gestire le risorse hardware e software ed esporta allo strato applicativo un'astrazione dello strato hardware, in modo tale da sollevare gli sviluppatori dall'onere di conoscere i dettagli di basso livello delle risorse fisiche.

Una tipica applicazione TinyOS based ha le dimensioni di qualche decina di KB, dei quali soltanto 400 byte riguardano il sistema operativo: è proprio dalla dimensione decisamente ridotte che deriva il nome TinyOS. Tra gli elementi più significativi di TinyOS emergono il modello di programmazione component-based, codificato dal linguaggio NesC [8] (definito da uno dei principali progettisti come "un dialetto del C") ed il modello di esecuzione event-driven quale supporto alla concorrenza. TinyOS non possiede un vero e proprio kernel, ma permette l'accesso diretto all'hardware, per mezzo di una *Hardware Abstraction Architecture*. Esistono, tuttavia, alcuni componenti di sistema che prescindono da tale approccio. TinyOS è stato sviluppato originariamente per i Mica Motes di Berkeley, ma attualmente supporta più di dieci diverse piattaforme e numerosi tipi di sensor board.

Un'ampia comunità di utilizzatori impiega TinyOS per lo sviluppo e il test di algoritmi, applicazioni e protocolli per wireless sensor network. Molti gruppi di ricerca accademici e industriali lo utilizzano sui nodi Crossbow [14] /Berkeley. La natura open-source del sistema operativo ha consentito una diffusione tra i vari gruppi di sviluppo che attualmente concorrono per la definizione di standard e servizi di rete, sulla base sia di esperienze dirette che di esigenze commerciali. TinyOs oggi ha guadagnato una posizione di grande rilievo nel contesto delle WSN, affermandosi sempre più come uno standard de facto tra i sistemi operativi ad esse dedicati.

L'attuale release ufficiale di TinyOS è la 2.1.0, rilasciata il 22 Agosto 2008. La prima release di TinyOS risale al 2002; da allora ha avuto inizio un periodo di grande partecipazione nelle ricerche e negli sviluppi tecnologici delle reti di sensori. Il sistema operativo TinyOS è riuscito ad adattarsi a questo progresso, soprattutto con l'ausilio della comunità internazionale di sviluppatori. Tuttavia, questi anni di esperienza hanno portato i ricercatori di Berkely a reingegnerizzare e riprogettare alcuni aspetti basilari del sistema operativo. Nonostante TinyOS 2.x abbia mantenuto molti dei concetti chiave delle precedenti versioni di TinyOS 1.x (primo fra tutti l'architettura component-based e event-driven codificata da NesC [8]), esso presenta al momento una implementazione profondamente diversa dalle precedenti realease. Ciascuno dei più importanti sottosistemi di TinyOS 2.x è stato associato ad un documento TEP (*TinyOS Enhancement Proposal)* [11] che illustra le linee guida della progettazione e la struttura del sottosistema stesso, facendo riferimento ad implementazioni esemplificative. Ogni TEP è aperto a modifiche e revisioni da parte della comunità di utilizzatori. Le novità introdotte riguardano tre aree fondamentali:

1. Flessibilità e Portabilità rispetto alle piattaforme

2. Robustezza ed affidabilità

3. Concetto di distribuzioni di servizi

La maggiore flessibilità rispetto alle piattaforme hardware, che nel tempo sono diventate sempre più numerose, è stata realizzata con l'introduzione della versione 1.2 del linguaggio NesC [8] e tramite la realizzazione *dell'Hardware Abstraction Architecturee* in tre livelli: *Hardware Presentation Layer (HPL)*, *Hardware Adaptation Layer (HAL)* e *Hardware Interface Layer(HIL)* [11]. TinyOS 2.x, migliora invece la robustezza e l'affidabilità ridefinendo alcune tra le principali astrazioni e politiche delle precedenti versioni tra le quali: il processo di inizializzazione e la fase di boot, la gestione della coda dei task e, soprattutto, la gestione energetica. Infine, il nuovo concetto di distribuzione di servizi, paragonabile ad una vera e propria API, semplifica notevolmente lo sviluppo delle applicazioni.

Architettura del sistema operativo TinyOS

L'architettura di TinyOS 2.x [13] si può rappresentare attraverso una decomposizione in layer verticali ed orizzontali dei componenti del sistema operativo (fig. 2).

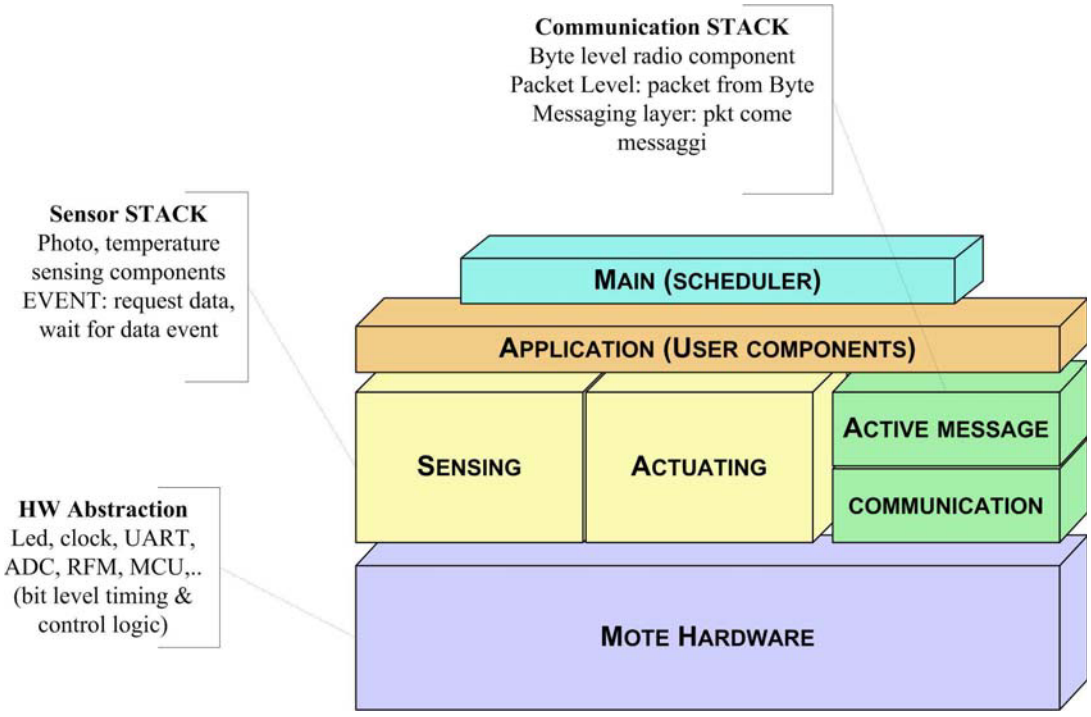

Figura 2. Rappresentazione concettuale dell'architettura di TinyOS 2.x

Le astrazioni hardware, lo stack per la comunicazione, quello per i sensori e gli attuatori e lo strato applicativo sono esempi di layer verticali. I layer che occupano una posizione più bassa all'interno dell'architettura del sistema, sono hardware dipendenti che esportano verso l'esterno interfacce complesse per la gestione dei dispositivi fisici dei nodi sensori. In modo duale, i layer che occupano posizioni più elevate sono indipendenti dai dispositivi fisici ed esportano interfacce più user-friendly. La decomposizione orizzontale riguarda la struttura degli specifici sottosistemi e permette di semplificare la portabilità, consentendo il riutilizzo delle astrazioni in piattaforme diverse. La decomposizione orizzontale è particolarmente evidente nel layer che realizza l'astrazione hardware del nodo

sensore. Il layer *Mote Hardware* rappresenta perfettamente la struttura fisica di un nodo sensore realizzata da una composizione orizzontale di chip standard (MCU, RF Transceiver, FLASH,...). Modellando ogni chip come astrazione indipendente dalla piattaforma si consente la portabilità dei chip in piattaforme diverse. Nel livello più basso dell'architettura di TinyOs 2.x troviamo l'*Hardware Abstraction Architecture*, ovvero i componenti che forniscono l'astrazione dei dispositivi hardware del nodo sensore. La rete coinvolge un complesso stack in cui i componenti a livello più basso, che sono a diretto contatto con il canale radio, realizzano lo stream di dati in uscita suddividendolo in pacchetti, l'attività di codifica di errori e di scheduling del canale di comunicazione. Questi componenti implementano inoltre la gestione dei pacchetti in arrivo e del loro smistamento nei buffer di ingresso. Nel livello più alto dell'architettura si trovano i componenti che si occupano delle attività di gestione dei buffer e realizzano il multiplexing della rete attraverso i diversi componenti applicativi. Parallelamente al *communication stack* si trova il *sensor stack* che fornisce meccanismi per l'interazione con i dipositivi sensori e con gli eventuali attuatori. Immediatamente sopra i *sensor* e *communication stack* si collocano i componenti implementati dall'utente per rispondere alle specifiche esigenze applicative della rete. A livello più alto troviamo, infine, il componente di sistema *Main* che include, a sua volta, un altro componente indispensabile per ogni applicazione, ovvero il componente di *Scheduler*. Il *Main* implementa la sequenza di boot del sistema operativo, realizza l'inizializzazione della piattaforma hardware, dei componenti software ed esegue il main task loop.

Hardware Abstraction Architecture[1]

Nel TinyOS 2.x [13], come nei sistemi operativi moderni, le risorse fisiche di un dispositivo sono mappate in un *Hardware Abstaction Layer (HAL)* software così da consentire la portabilità e semplificare lo sviluppo dello strato applicativo. Tuttavia questo approccio ha un certo peso sulle prestazioni e non primeggia per efficienza energetica. Nel contesto delle WSN, dove l'ottimizzazione dei consumi energetici e delle risorse hardware sono requisiti funzionali indispensabili, occorre trovare un compromesso tra le caratteristiche di modularità, flessibilità e riusabilità da un lato e l'utilizzo efficiente delle risorse a disposizione dall'altro Per raggiungere questo scopo, gli sviluppatori di TinyOS 2.x hanno realizzato una *Hardware Abstraction Architecture (HAA)* articolata in tre distinti strati orizzontali di componenti: *Hardware Presentation Layer (HPL)*, *Hardware Adaptation Layer (HAL)* e *Hardware Interface Layer (HIL)* (fig. 3). La sovrapposizione di questi tre strati definisce in modo graduale, tra il sistema operativo e le applicazioni, un'interfaccia indipendente dalla piattaforma. Ogni strato è caratterizzato da funzionalità offerte ben definite ed è subordinato alle funzionalità esportate dai livelli inferiori; percorrendo l'HAA dal basso verso l'alto diminuisce la dipendenza dei componenti dall'hardware e aumenta potenzialmente la riusabilità delle applicazioni.

[1] Dal TEP 2 - http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep2.html

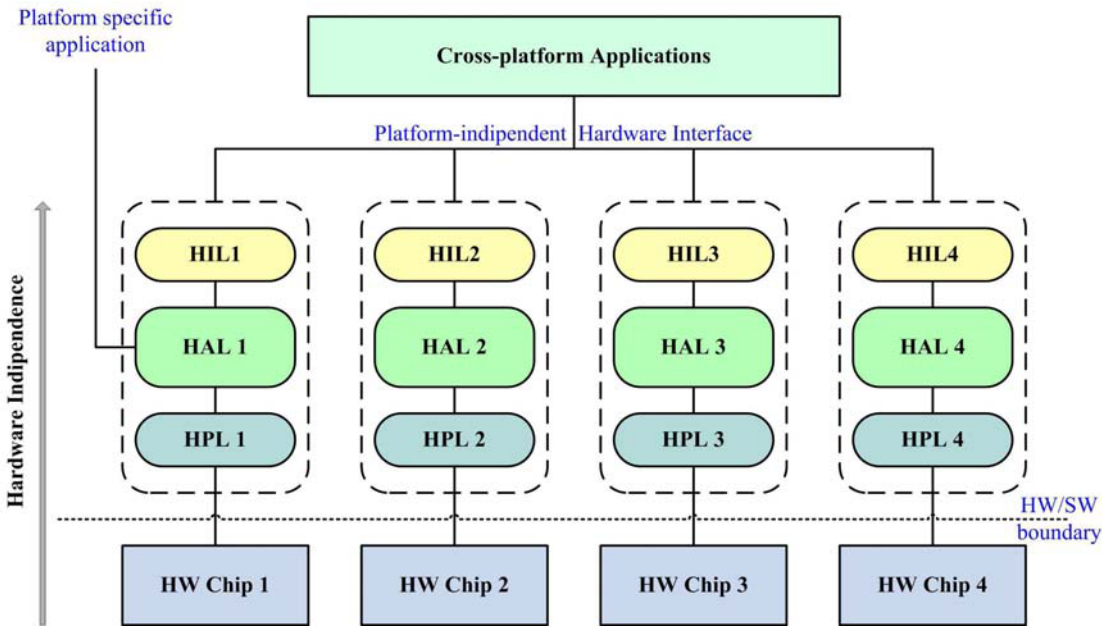

Figura 3. Layers dell'Hardware Abstraction Architecture di TinyOS 2.x

Hardware Presentation Layer

L'Hardware Presentation Layer (HPL) [13] rappresenta lo strato dell'HAA a diretto contatto con i dispositivi hardware dei nodi sensori: questo livello presenta le funzionalità e i servizi dello strato hardware ai componenti di livello superiore, integrando i concetti nativi del sistema operativo. La presenza dell'HPL, consente di nascondere la complessità dell'hardware esportando al resto del sistema un'interfaccia usufruibile in modo più semplice. Ciascun componente dell'HPL è unico e determinato completamente dalle specifiche del modulo hardware di cui rappresenta l'astrazione. Questa caratteristica limita pertanto la libertà d'azione del progettista nell'implementazione dei componenti dell'HPL. Con lo scopo di ottimizzare l'integrazione con il resto dell'architettura, ogni componente HPL rispecchia una struttura generale basata su:

- Comandi per l'inizializzazione, l'avvio e l'arresto: requisiti indispensabili per le politiche di gestione energetica.
- Comandi di get e set per il controllo dei registri.

- Comandi per l'abilitazione e la disabilitazione degli interrupt hardware.
- Comandi per le operazioni di flag-setting e di test.
- Routine di servizio per le interruzioni time-critical, generate dal modulo hardware. La gestione delle interruzioni senza vincoli temporali stretti viene delegata ai componenti di livello superiore i quali possiedono una conoscenza estesa dello stato del sistema.

Si giunge quindi alla conclusione che la struttura dei componenti HPL semplifica sostanzialmente la manipolazione dell'hardware, rendendo trasparente all'utilizzatore l'esistenza di gran parte del codice hardware-dipendent e dispensando gli sviluppatori dalle difficoltà legate all'interazione di basso livello con i dispositivi fisici. Inoltre il layer HPL, prepara l'ambiente allo sviluppo di componenti astratti di livello superiore. A questo proposito è necessario ricordare che molti dei microcontrollori impiegati nelle WSN possiedono due moduli *USART* (*Universal Synchronous-Asynchronous Receiver/Transmitter*) per la comunicazione delle interfacce seriali. Entrambi i moduli hanno le stesse funzionalità ma differiscono per i nomi dei registri e per gli interrupt generati. I componenti del livello HPL consentono di nascondere queste differenze, realizzando astrazioni di livello superiore indipendenti dal tipo di risorsa utilizzata. Allo sviluppatore è consentito cambiare i moduli USART semplicemente rivedendo in modo opportuno il wiring dei componenti HPL, senza che sia necessario modificare l'implementazione di base.

Hardware Adaptation Layer

L'*Hardware Adaptation Layer* [13] rappresenta il nucleo dell'architettura TinyOS 2.x. I componenti di questo livello utilizzano le interfacce di basso livello, esportate dai componenti dell'HPL, per realizzare astrazioni di alto livello che nascondano la complessità tipicamente associata all'utilizzo delle risorse hardware. In contrasto con i componenti dell'HPL, i componenti HAL sono in grado di mantenere informazioni di stato e possono essere impiegati per realizzare arbitraggio e controllo delle risorse. Sebbene questo strato dell'architettura aumenti significativamente il livello di astrazione, resta una forte dipendenza dalle risorse hardware. I componenti dell'HAL sono infatti strutturati in modo specifico sulle classi e sulle piattaforme dei dispositivi; tuttavia, anziché nascondere, così come avveniva nell'HPL, le caratteristiche individuali dei singoli dispositivi fisici dietro a modelli generici, essi esportano delle interfacce peculiari del particolare dominio hardware come EEPROM, il canale per la conversione A/D, Alarm, MCU, etc..

Hardware Interface Layer

I componenti dell'*Hardware Interface Layer (HIL)* [13] costituiscono l'ultimo livello dell'HAA. Tali componenti, partendo dalle astrazioni platform-specific fornite dall'HAL sottostante, esportano allo strato applicativo un'astrazione di tipo hardware-indipendent. L'HIL, pertanto, semplifica sensibilmente lo sviluppo e accresce significativamente la portabilità delle applicazioni, nascondendo agli sviluppatori le differenze hardware degli strati sottostanti. Per poter essere utilizzabile l'HIL deve riflettere i caratteristici servizi hardware delle tipiche applicazioni per WSN. La complessità dei componenti dell'HIL dipende, in generale, dalla complessità dell'hardware sottostante e dalle funzionalità richieste

all'interfaccia platform-indipendent: l'HIL potrebbe semplicemente dover mappare i servizi già offerti dai dispostivi fisici sottostanti oppure dover implementare, attraverso simulazioni software, particolari capacità non possedute dall'hardware. Poiché i componenti dell'HIL seguono l'evoluzione tecnologica delle piattaforme hardware, i progettisti di TinyOS hanno introdotto il versionamento delle loro interfacce. Assegnando un numero ad ogni versione di un'interfaccia HIL si abilita lo sviluppo di applicazioni caratterizzate da interfacce legacy compatibili con specifiche caratteristiche dei dispositivi fisici. Il versionamento consente inoltre di ramificare l'HIL, secondo livelli funzionali crescenti.

Possibilità di selezionare il livello di astrazione

La ragione principale alla base della strutturazione dell'HAA in tre layer distinti (anziché in un'unica hardware interface platform-indipent direttamente a livello dei componenti) è stata quella di voler accrescere la flessibilità del sistema separando le astrazioni specifiche delle piattaforme hardware, dai componenti wrapper attraverso i quali si realizza l'interfaccia platform-indipendent. Questo approccio consente di massimizzare le prestazioni, ammettendo la possibilità di aggirare l'HIL accedendo direttamente alle piene potenzialità del modulo hardware per mezzo dell'HAL. Non è esclusa l'altra importante possibilità di accedere ai servizi di un dispositivo fisico utilizzando due differenti livelli di astrazione da parti distinte dell'applicazione o delle librerie del sistema operativo. Al fine di supportare questa forma di flessibilità verticale, sono stati aggiunti a livello dell'HAL adeguate funzionalità di arbitraggio e controllo, cosi da consentire un accesso sicuro e condiviso alle risorse esportate dall'HPL.

Capitolo 2

Il sistema Mad-WiSe

Il sistema MaD-WiSe è stato brevemente introdotto nel Capitolo 1. Questo capitolo mostra il funzionamento di Mad-WiSe, i componenti principali del sistema e la struttura del sistema, organizzato secondo un preciso stack protocollare.

2.1 Il funzionamento del sistema

MaD-WiSe è un sistema software che consente la gestione di dati nei wireless sensor networks attraverso l'utilizzo di un linguaggio SQL-Like [1,5,7]. Il principio fondamentale di questo sistema è che considera la rete di sensori come un database distribuito dove ogni nodo possiede una tabella composta da colonne che corrispondono ai trasduttori disponibili sul nodo stesso e da colonne per uso specifico come per esempio contenuti di natura temporale. Il rilevamento dei dati dai trasduttori si effettua preparando una query (fig. 4) che viene interamente eseguita dalla rete. Nella query rappresentata in tabella 1 si richiede ogni 500 millisecondi il dato della luminosità dai sensori 1, 2 e 4 nel caso in cui il dato della luminosità del sensore 2 sia maggiore di 20.

```
SELECT * FROM 1.Light, 2.Light, 4.Light
where 2.Light > 20 every 500
```

Tabella 1. Esempio di query per l'interrogazione di sensori con Mad-WiSe

Ogni nodo coinvolto nella query partecipa attivamente all'esecuzione collaborando con gli

altri nodi, processando localmente i dati con gli operatori dell'algebra relazionale. L'utente, utilizzando l'interfaccia grafica, definisce una query che viene ottimizzata [7] ed elaborata per definire il cosiddetto *query plan*, cioè l'insieme delle operazioni che ciascun nodo deve svolgere durante l'esecuzione. Il *query plan* viene iniettato nella rete tramite il nodo *sink* collegato via seriale che provvede all'inoltro dei messaggi via radio. Ogni nodo esegue la porzione di query ad esso assegnata. Da questo momento la rete, in quanto programmata, esegue autonomamente il lavoro assegnato: ciascun nodo rileva le informazioni sull'ambiente circostante, elabora i dati acquisiti e quelli ricevuti dai vicini e li trasmette fino al nodo *sink* che si occupa a sua volta di ritrasmetterli all'interfaccia utente affinché possano essere mostrati mediante grafici e tabelle.

2.2 I componenti di Mad-WiSe

In questo paragrafo verranno esposti i principali componenti del sistema Mad-WiSe e il loro ruolo nel funzionamento dell'applicazione.

Interfaccia utente (GUI) SensorViz

L'interfaccia utente di Mad-Wise si chiama SensorViz. È un'applicazione sviluppata in Java che viene eseguita su un comune PC dotato di porta seriale. SensorViz permette all'utente di definire le query da eseguire sulla rete di sensori e il relativo piano di esecuzione. Consente inoltre di visualizzare i risultati prodotti dalla rete, tramite tabelle riassuntive e grafici ottenuti in tempo reale. Il compito principale è quello di permettere all'utilizzatore di preparare la query in base alla quale la GUI effettua l'ottimizzazione e definisce il planning

per ogni nodo. In questa fase vengono definite le comunicazioni che si instaureranno, quando la *query plan* passerà in esecuzione, tra i vari nodi e gli operatori eseguiti da ciascun nodo con i rispettivi argomenti. Nel momento in cui l'utente deciderà di eseguire la query che ha definito, SensorViz si occuperà di trasmetterla alla rete di sensori via seriale attraverso il nodo *sink*, inviando comandi appositi per programmare ciascun nodo.

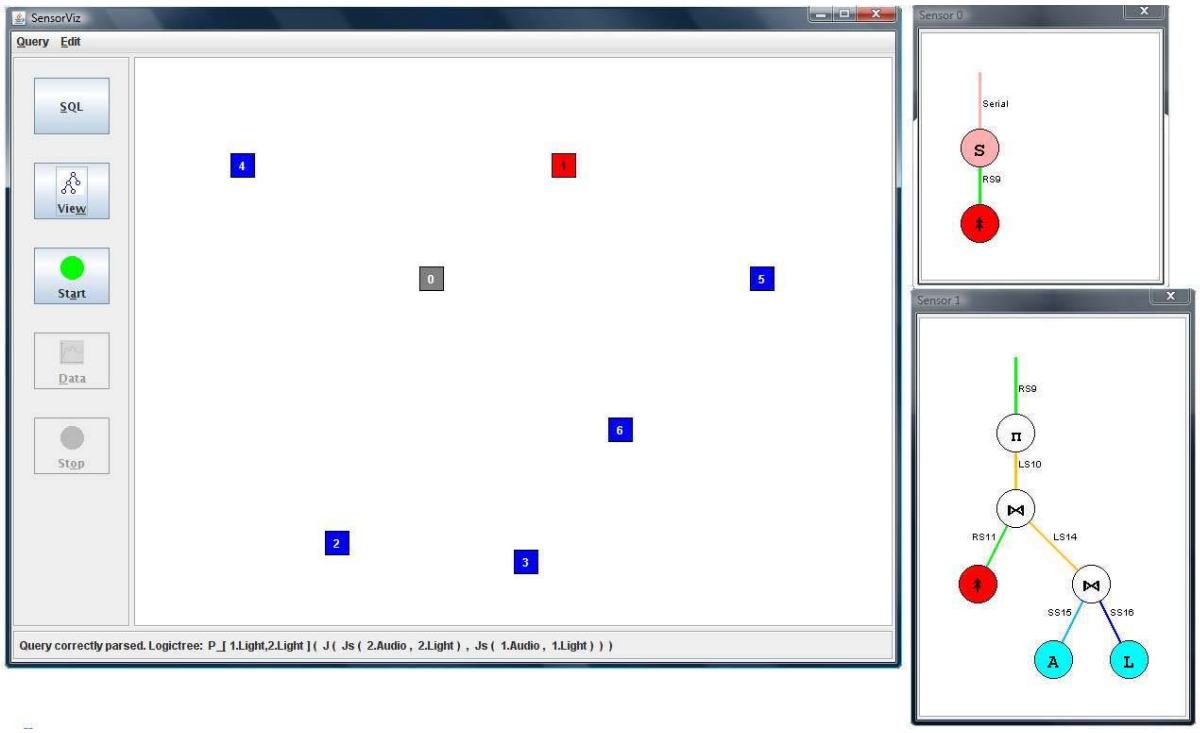

Figura 4. SensorViz nella sua schermata principale visualizza la topologia della rete e il query plan

Durante l'esecuzione della query i nodi scambiano continuamente informazioni tra di loro e verso il nodo sink. Il nodo sink provvede a inoltrare le informazioni a SensorViz tramite seriale. La GUI riceve i dati e consente di visualizzare i risultati sottoforma di tabella e grafici riguardanti i valori dei sensori richiesti con la query.

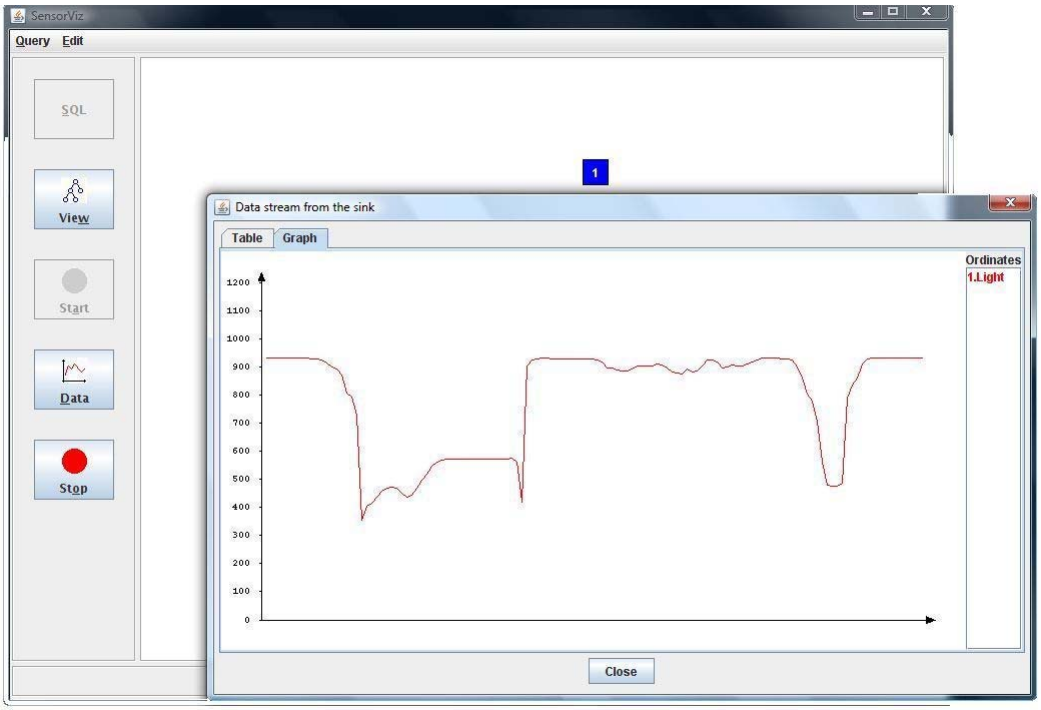

Figura 5. SensorViz consente di visualizzare il risultato della query tramite grafici e tabelle

Il nodo sink

Si tratta di un nodo della rete al quale viene assegnata la caratteristica di *sink*. Esso offre la funzionalità di gateway tra la rete di sensori e il PC. Dal punto di vista fisico il *sink* è un nodo identico agli altri nodi della rete, sul quale non è presente una sensor board infatti non è previsto che il *sink* partecipi al rilevamento dell'ambiente. Inoltre è collocato sopra un supporto, il *MIB520* [14], che consente l'interfacciamento seriale del nodo.

Figura 6. Il supporto MIB520 che consente di interfacciare un nodo della rete alla seriale di un PC

Così connesso, il nodo *sink* riceve i messaggi da SensorViz contenenti il query plan e provvederà ad inoltrare messaggi via radio al nodo destinatario contenenti il compito che esso deve svolgere Viceversa ogni nodo può recapitare dei messaggi via radio al *sink* per esempio il responso di una query.

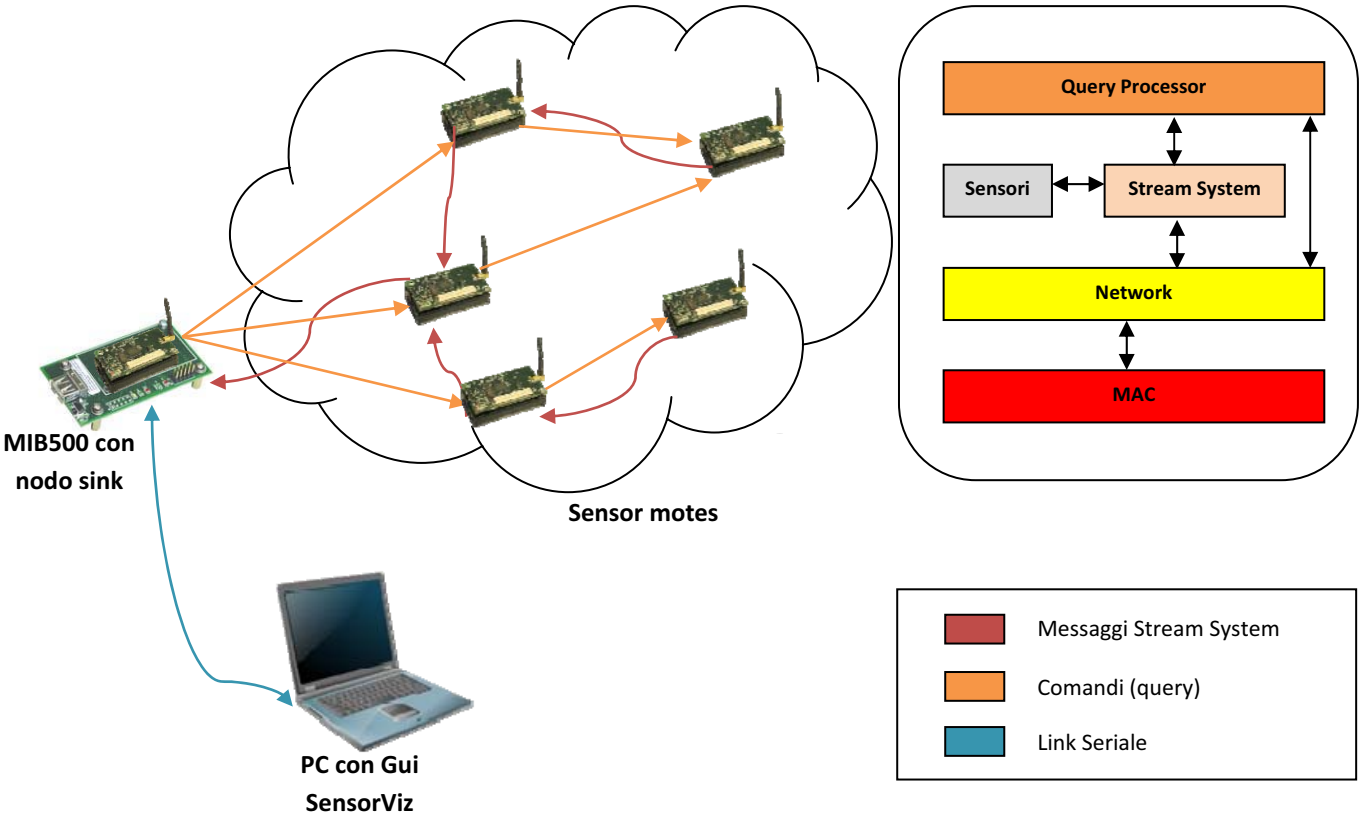

Figura 7. Funzionamento ed elementi del sistema MaD-WiSe

Nodo della rete

Ogni nodo della rete ha in esecuzione lo stack di MaD-WiSe e quindi è in grado di gestire l'hardware del sensore, effettuare l'elaborazione dei dati e gestire le comunicazioni. Lo stack è composto da tre livelli:

- Query Processor [5] : è il livello “applicazione” dove agiscono gli operatori dell'algebra relazionale definita dal linguaggio di interrogazione SQL-like di MaD-WiSe.

- Stream System [2] : è il livello di trasporto che interfaccia lo strato applicativo con quello di rete e con i trasduttori hardware. A questo livello è definito il concetto di stream come supporto per il trasporto di dati. L’insieme di trasduttori, operatori e stream costituisce il piano di esecuzione della query per ciascun nodo.

- Network [6] : è il livello che si occupa della comunicazione tra i nodi. Quando si presenta una richiesta di comunicazione da parte dello Stream System o del Query Processor verso altri nodi, è il livello network che si occupa di instaurare un canale di comunicazione tra i nodi coinvolti, utilizzando il livello MAC messo a disposizione da TinyOS. Il dettaglio implementativo della funzionalità Multi-hop del livello Network, uno degli aspetti principali di questo studio, sarà affrontato approfonditamente nel capitolo 4. Lo strato di rete è strettamente connesso ad un modulo chiamato EnergyEfficiency che si occupa di gestire l’accensione e lo spegnimento della radio quando l’utilizzo non è richiesto, nell’ottica di perseguire il risparmio energetico. È noto, nell’ambito delle reti wireless di sensori, che dal punto di vista del consumo energetico la radio è il componente più dispendioso: per questo motivo è opportuno gestire accuratamente gli istanti in cui l’utilizzo della radio non è richiesta.

2.3 Layer Architecture di Mad-Wise

I livelli dello stack di Mad-Wise adottano il paradigma di interfaccia comandi/eventi classico di nesC e TinyOS. Mostriamo adesso come i tre strati dell'architettura interagiscono tra di loro utilizzando questo paradigma.

Interfaccia Network / Stream System

Considerando Mad-WiSe in esecuzione su una rete di sensori, grazie al livello Network, lo Stream System [6] ha la possibilità di inviare i propri pacchetti agli altri nodi della rete; lo strato mette a disposizione chiamate per creare canali di comunicazione attraverso i quali è possibile inviare e ricevere dati mediante l'interfaccia ConnectionOriented.

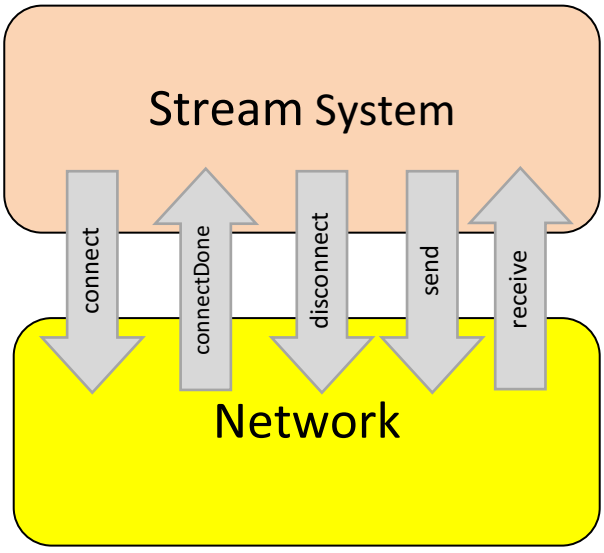

Figura 8. Interfaccia ConnectionOriented

La figura 8 rappresenta l'interfaccia sopraccitata. Gli eventi e i comandi di questa interfaccia sono i seguenti:

- La primitiva *connect* consente di aprire un canale di comunicazione tra il nodo che la invoca e il nodo destinazione; il livello Network si occupa di allocare le strutture dati necessarie a gestire il canale sui nodi coinvolti nella comunicazione. Questo insieme di operazioni richiedono l'impiego di un tempo, durante il quale il nodo può continuare a svolgere le proprie funzioni.

- Una volta stabilita la connessione, il livello Network invia il segnale *connectDone* al livello Stream System, il quale può utilizzare il canale per inviare i dati al nodo partner.

- Il comando *disconnect* serve a chiudere il canale e a liberare le risorse allocate sul nodo locale, su quello remoto e sui nodi intermedi.

- Il comando *send* offre la funzionalità di invio dati lungo il canale; lo Stream System utilizza questo comando mettendo in atto il passaggio di argomenti quali l'identificativo del canale ottenuto mediante la *connect*, il pacchetto di livello Stream System e la sua dimensione.

- Tra i compiti del Network esiste anche quello di ricevere i pacchetti che arrivano via radio dagli altri nodi. Quando questo accade, un evento di tipo *receive* viene segnalato allo Stream System per ciascun pacchetto, fornendo l'id del canale da cui è stato ricevuto il pacchetto, la sua dimensione e un puntatore al contenuto. Il livello Stream System processa ogni pacchetto e intraprende diverse azioni in funzione del suo tipo e del suo contenuto.

Interfaccia Stream System / Query Processor

Il livello Stream System [2] mette a disposizione l'interfaccia SSToApp ed è schematizzata in figura 9. La struttura è simile all'interfaccia offerta dal livello Network verso il livello Stream System con la differenza che a fronte di un solo tipo di chiamata, qui se ne trovano diversi tipi, ognuna relativa a una tipologia di stream differente. Il Query Processor [5] si occupa di rispondere alla query iniettata dal *sink* attraverso il collegamento con altri nodi della rete e con i sensori presenti sul nodo stesso; i collegamenti avvengono attraverso gli stream creati con i comandi messi a disposizione da questa interfaccia.

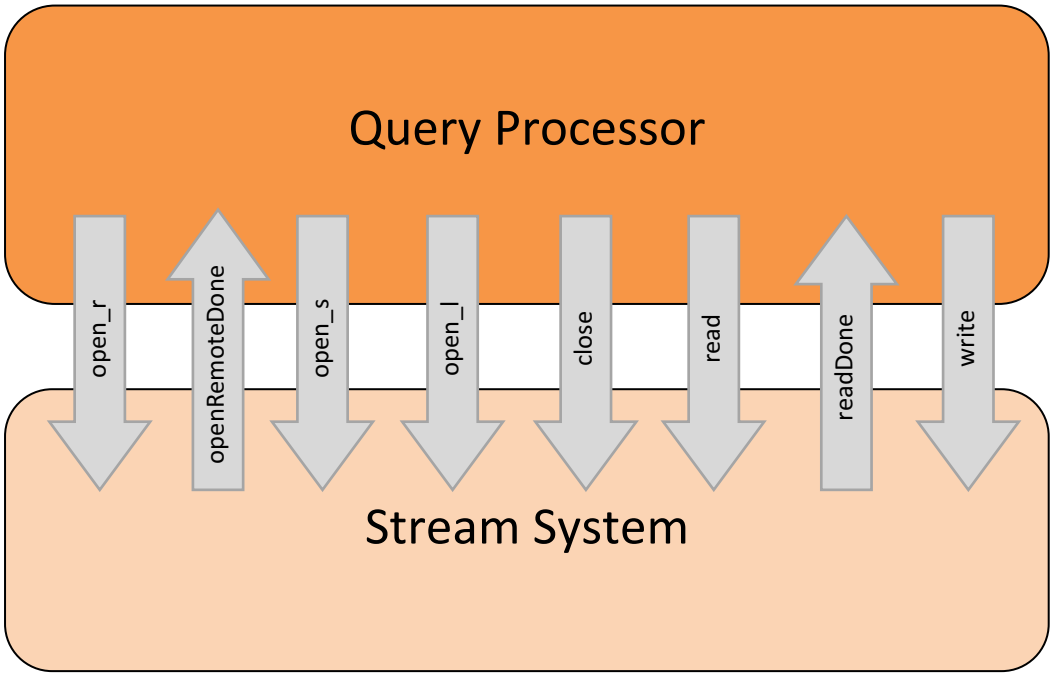

Figura 9. Interfaccia SStoApp

- Il comando *open_r* consente al Query Processor di creare uno stream per trasportare dati dal nodo locale al nodo remoto. Le operazioni che consentono la creazione dello stream sono asincrone e richiedono del tempo, anche perché nella loro

implementazione comprendono la chiamata alla *connect* nel livello Network, utilizzata per creare il canale di comunicazione su cui viaggeranno i dati dello stream.

- L'evento *openRemoteDone* interviene per segnalare al livello Query Processor l'avvenuta creazione di uno stream. L'evento ha luogo in seguito alla rilevazione da parte del livello Stream System dell'evento *connectDone* generato dal livello Network.

Gli altri comandi necessari alla creazione di stream sono:

- open_s: consente la creazione di uno stream sensore attraverso la configurazione del trasduttore, dell'allocazione di memoria e della preparazione di un timer
- open_l: permette la creazione di uno stream locale e prevede l'allocazione della memoria destinata ai buffer dello stream e la registrazione dello stream in una tabella

Questi tre comandi non necessitano un lungo periodo di tempo per svolgere il proprio compito, pertanto non è stato necessario predisporre un evento asincrono che notifichi l'avvenuta creazione dello stream. Alla conclusione si ottiene immediatamente l'esito dell'operazione e il descrittore dello stream appena creato.

- La chiusura di uno stream si compie mediante il comando *close*.
- Il comando *read* consente di leggere i dati da uno stream aperto; la chiamata è unica per tutti i tipi di stream poiché, grazie al descrittore passato come argomento, l'implementazione decide da dove e come reperire i dati.
- La lettura dei dati da uno stream è un evento asincrono e richiede la presenza dell'evento *readDone* che segnali al livello Query Processor che i dati sono effettivamente disponibili e pronti per essere usati. Il campionamento di un dato da un

trasduttore è implementato in modo asincrono dal sistema operativo (il quale genera a sua volta un evento quando il dato è pronto). La lettura di dati da uno stream remoto è subordinata alla ricezione dei medesimi da parte del livello Network e alla loro scrittura da parte del partner remoto nel proprio estremo dello stream.

- Il comando *write* effettua la scrittura in uno stream, anche qui discriminando il tipo di stream, in funzione del descrittore passato al comando.

Capitolo 3

Reingegnerizzazione del sistema Mad-WiSe

In questo capitolo si vedrà come è stata pianificata e realizzata la reingegnerizzazione di Mad-WiSe al TinyOs 2.x. In particolare descriveremo come hanno influito le differenze tra TinyOs 1.x e TinyOs 2.x nelle fasi della reingegnerizzazione.

3.1 Differenze tra TinyOs 1.x e TinyOs 2.x

Nel paragrafo 1.4 si è visto come i ricercatori dell'Università di Berkley siano giunti alla versione 2.x di TinyOs [11] tramite la reingegnerizzazione della versione 1.x. L'architettura component-based e event-driven codificata da NesC [8] è la stessa di TinyOs 1.x, tuttavia, l'approccio usato ha permesso una elevata astrazione dall'hardware utilizzato. Ma le novità introdotte riguardano anche altri aspetti del Sistema Operativo, dovute soprattutto all'esperienza nell'utilizzo nel campo delle reti a sensori, al fine di ottenere un Sistema Operativo più stabile ed efficiente, anche in previsione dell'introduzione di nuove funzionalità. I cambiamenti più consistenti nella logica del Sistema Operativo [11], riguardano il modello di gestione dei *Task*, il *Bootstrap* dei nodi, la gestione della funzionalità di radio, i *Timer* e il significato delle variabili di tipo *result_t* come risultato di ritorno di un evento o di un comando.

In 1.x il modello dei *Task* si basa su una coda di dimensioni definite, condivisa tra i vari *Task*. Questo consente a ciascun componente il *post* di *Task* multipli, anche dello stesso

tipo. In 2.x ciascun *Task* ha uno slot di memoria riservato. Questo significa che, nel caso in cui il *Task* non sia andato ancora in esecuzione, il *post* di *Task* multipli non ha un esito positivo.

In 2.x è cambiata anche la sequenza di *Bootstrap*[2] con l'introduzione delle interfacce *Boot* e *Init*: il comando *Init.init()* consente di inizializzare con certezza componenti e hardware prima del boot vero e proprio del sistema. Il suo utilizzo risulta essere particolarmente utile per assicurarsi che l'inizializzazione sia avvenuta: questa operazione è indispensabile per il corretto funzionamento del sistema. L'evento *Boot.booted()* viene segnalato nel momento in cui il sistema ha completato la fase di boot con successo. Nella versione 1.x l'interfaccia *StdControl* si occupa dell'inizializzazione e l'avvio dei componenti tramite gli eventi *StdControl.init() e StdControl.start()*. La fase di inizializzazione è strettamente legata all'inizializzazione hardware. Se un componente deve essere inizializzato necessita infatti di un avvio esplicito. Se un componente ha invece la caratteristica di dover essere avviato e fermato spesso (per esempio la radio), si dovrà affrontare l'inconveniente che lo *start* e lo *stop* non seguono un modello sincrono: questo potrebbe causare il malfunzionamento del sistema.

La gestione della funzionalità radio[3], in TinyOs 1.x viene resa operativa all'avvio tramite l'interfaccia *GenericComm*: nel caso in cui si debba attivare e disattivare la funzionalità radio sarà necessario effettuare lo *start* e lo *stop* (passando per lo *StdControl*) dell'intera interfaccia linkata a *GenericComm*. In TinyOs 2.x, dopo la fase di boot la radio non è operativa: la funzionalità della radio, gestita tramite l'interfaccia *SplitControl*, ha

[2] http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep107.html

[3] http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep116.html

bisogno di uno *start* esplicito che interessa esclusivamente questa interfaccia. Le chiamate *Splitcontrol.start()* e *Splitcontrol.stop()* non sono altro che un esplicito intervento sul controllo energetico della funzionalità radio.

Anche l'utilizzo dei *Timer*[4] è stato rinnovato: nella versione 1.x esiste una distinzione tra timer assoluti e timer relativi. È l'interfaccia *TimeSet* che consente di impostare il clock interno del nodo con un valore fornito come parametro. Questa funzionalità, utilizzata spesso nel caso in cui si richiede la sincronizzazione dei nodi, non è stata inclusa in TinyOs 2.x presumibilmente perché si è ritenuto concettualmente scorretto impostare il clock interno con un valore differente da quello impostato nella fase di boot. La ridefinizione dei *Timer* in TinyOs 2.x è un'operazione strettamente legata alla dotazione hardware, basata sul concetto di precisione, lunghezza e accuratezza di un ciclo di clock. Attualmente vengono forniti timer assoluti, configurabili parametricamente in base alla precisione con *TMilli* (1024 ticks al secondo), *T32khz* (32768 ticks al secondo) e *TMicro* (1048576 ticks al secondo) di tipo *Periodic* (ripetuti nel tempo) o *OneShot* (una sola esecuzione) basati su una implementazione con timer relativi *startPeriodicAt(t0,dt)* e *startOneShotAt(t0,dt)* dove *t0* rappresenta il tempo attuale (*getNow()*) mentre *dt* indica il tempo in cui verrà segnalato l'evento *fired* a partire dal tempo *t0*.

Un cambiamento significativo, per la logica di un applicazione TinyOs based, è avvenuto con l'introduzione della variabile *error_t* al posto di *result_t*. In TinyOs 1.x un evento o un comando potevano restituire un *result_t* con il valore SUCCESS o FAIL rispettivamente uguali a 1 e 0. Nella versione 2.x è stata introdotta *error_t* che può assumere valori SUCCESS o FAIL, ma con rispettivo significato 0 e 1. *error_t* inoltre può assumere

[4] http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep102.html

altri valori che indicano particolari errori. Questo cambiamento è stato introdotto per prevedere nuovi tipi di errori da definire in modo standard all'interno del framework anche se comporta, nel caso di porting di un applicazione, una revisione completa delle condizioni nei costrutti che prevedono istruzioni di salto condizionato.

I cambiamenti nelle interfacce fornite da TinyOs 2.x sono dovuti all'introduzione di *message_t*[5] che sostituisce *TOS_Msg*, il buffer per i messaggi in TinyOs 1.x. Esso contiene un pacchetto *ActiveMessage* e una serie di metadati, come *timestamps e* bit per l'ack. Inoltre ha una dimensione fissata e implementa il meccanismo *zero-copy*: quando un componente riceve un messaggio anziché effettuarne la copia, restituisce al livello sottostante un puntatore ad un nuovo buffer da utilizzare per la ricezione del messaggio successivo. La configurazione di *TOS_Msg* dipende dalla piattaforma radio utilizzata; ogni piattaforma può definire un proprio formato di messaggio. Questo tipo di approccio presenta due tipologie di problema: in primo luogo, l'esposizione dei campi del messaggio al *link layer* consente ai componenti di accedere direttamente alla struttura del pacchetto, introducendo dipendenze tra i componenti di un livello più alto e la struttura del messaggio. In secondo luogo questo modello non supporta facilmente *link layer* multipli. Le implementazioni dei chip radio, danno per assunto che siano presenti i campi per loro configurati. Nel caso in cui si usino interfacce radio differenti sarà necessario assegnare a ognuna la giusta quantità di memoria; inoltre si dovrà consentire ai livelli superiori di accedere a ulteriori campi dell'header del messaggio. Anche l'adeguamento del payload risulta essere problematico in quanto molti componenti si riferiscono a questo campo: è quindi necessario stabilire un offset dall'inizio della struttura. Per tutti questi motivi gli sviluppatori di TinyOS hanno pensato di riconsiderare il buffer per i messaggi. La struttura di un *message_t* è la seguente:

[5] http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep111.html

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Tabella 2. Struttura di un *message_t* in TinyOS 2.x

Tutti i campi del messaggio sono del tipo *nx_uintVALUE_t* dove *VALUE* può essere 8, 16 o 32 e la struttura è del tipo *nx_struct*. Con l'introduzione degli *nx types* o (network byte-order type) si rende uniforme la codifica da little-endian a big-endian nel caso in cui l'hardware in dotazione utilizzi codifiche differenti. Per quanto riguarda la struttura del messaggio, ogni campo ha una dimensione definita dipendente dalla piattaforma. Nel caso in cui si debba veicolare un messaggio tra due *link layers* differenti, è necessaria un'operazione di copia (a seconda della dimensione del campio). A differenza di TinyOs 1.x, dove il *TOS_Msg* è un pacchetto con caratteristiche esplicite di *ActiveMessaging*, *message_t* è un buffer molto più generico. Ai campi *header, footer* e *metadata* non vi si può accedere in modo diretto dal codice sorgente; inoltre devono essere rigorosamente definiti in un file, il cui nome è *platform_message.h*, contenuto nella relativa directory di TinyOs dedicata ai chip radio.

TinyOs 2.x consente di interagire con i pacchetti attraverso apposite interfacce. Il pacchetto viene considerato come una struttura dati di tipo astratto (ADT), con tutti i vantaggi che porta questo tipo di approccio. In primo luogo non sussiste la dipendenza dal nome di un campo o dalla sua posizione. Per esempio in TinyOs 1.x un componente può accedere direttamente al campo *TosMsg.addr* che rappresenta il destinatario di un messaggio. È possibile accedere al suo contenuto in lettura o modificarlo con un semplice assegnamento. In TinyOS 2.x, viene fornita un interfaccia *AMPacket* con dei comandi che consentono l'accesso ai campi dell'ActiveMessage. Un componente può effettuare la chiamata *AMPacket.destination(msg)*, dove *msg* è del tipo *message_t*, e ottenere così il destinatario del

messaggio. Per impostare il destinatario di un messaggio si deve chiamare il comando *AMPacket. setDestination(msg,addr)* dove *addr* è di tipo *am_addr_t* e indica il *TOS_NODE_ID* del nodo destinatario del messaggio ovvero il numero identificativo del nodo stabilito in fase di installazione. Buona parte dei comandi che consentono l'accesso al payload di *message_t* vengono forniti dall'interfaccia *Packet*. Il comando *Packet.getPayload(msg,call Packet.payloadLength(msg))* per esempio consente di accedere al payload di *msg* la cui dimensione viene passata come parametro tramite una ulteriore chiamata al comando *Packet.payloadLength(msg)*.

La comunicazione seriale[6] per TinyOs 2.x è stata modellata in modo molto simile alla comunicazione radio. È presente un *SerialActiveMessage* e il buffer del messaggio è sempre del tipo *message_t* . Anche le interfacce per accedere ai campi del pacchetto sono le stesse, in modo da semplificare l'utilizzo da parte del programmatore. Nei livelli sottostanti sono presenti le diverse astrazioni che consentono a un particolare Hardware di possedere, se necessario, diverse interfacce seriali. Questo differisce notevolmente dalla versione per TinyOs 1.x dove il formato dei pacchetti seriali (UART) è specifico per ogni piattaforma: in questo modo si aggiunge complessità al protocollo al fine di gestire correttamente le caratteristiche delle differenti piattaforme.

[6] http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep113.html

3.2 Reingegnerizzazione del sistema

In seguito all'analisi delle differenze tra le due versioni di TinyOs, si è proseguito con la reingegnerizzazione del sistema. L'obiettivo che ci siamo posti è stato quello di ottenere una versione di Mad-WiSe per TinyOs 2.x funzionalmente equivalente. L'architettura peculiare di Mad-WiSe [1] ci ha consentito di isolare i tre livelli Network, StreamSystem e QueryProcessor ed effettuare le modifiche su di essi singolarmente.

Per ogni livello si è proceduto in linea di massima come segue:

- Il codice sorgente di Mad-WiSe 1.4.8 è stato modificato nelle sezioni che implementavano funzionalità non estendibili (per scelta progettuale) nella nuova versione di Mad-WiSe.

- Le chiamate dei comandi e la segnalazione di eventi, i cui parametri sono cambiati sono stati modificati (per esempio *event TOS_MsgPtr RadioReceiveMsg.receive (TOS_MsgPtr pMsg)* è diventata *event message_t* RadioReceiveMsg.receive (message_t* pMsg, void* payload, uint8_t len)*. Si è seguita la stessa procedura per il relativo wiring con le nuove interfacce.

- L'accesso al payload dei *message_t* è stato implementato con le relative chiamate ai comandi delle interfacce *Packet* e *AMPacket*, come spiegato nel paragrafo precedente.

- Si è adattata la sequenza di Boot con le nuove interfacce fornite da TinyOS 2.x

- I componenti di TinyOs 1.x quali *GenericComm* e *TimerC* sono stati sostituiti con *AMSenderC*, *AMReceiverC* e *TimerMilliC*.

- Si è intervenuto nella definizione dei messaggi di ogni livello con l'introduzione degli *nx types* come *nx_uint8_t* al posto degli *uint8_t* .

- Si è intervenuto nella logica delle condizioni dei costrutti con salto condizionato per adeguarsi ai nuovi significati di FAIL e SUCCESS

- È stata effettuata la sostituzione di stringhe per le costanti il cui nome è cambiato dalla versione 1.x alla versione 2.x (per esempio TOS_LOCAL_ADDRESS ovvero l'identificativo del nodo è stato cambiato in TOS_NODE_ID)

Il livello Network [6] è stato il primo livello interessato dalla reingegnerizzazione. Sono stati isolati i moduli *EnergyEfficency* e *NetToSSM*, il file di configurazione *NetToSSC*, il file header *NetToSS* e i files che offrono le interfacce la cui implementazione si trova in questo livello: *Connectionless* e *ConnetionOriented*. I messaggi di questo livello sono definiti nel file header *MsgNet*. In seguito alle modifiche predefinite, enunciate all' inizio di questo capitolo, si è intervenuti nella logica del livello Network. Il cambiamento principale riguarda la gestione dei task e l'eliminazione della modalità promiscua nell'invio dei messaggi seriali. Nel Network sono presenti tre task:

- *InQueueTask*: gestisce i messaggi che arrivano via radio o via seriale.

- *OutQueueTask*: gestisce i messaggi in uscita via radio

- *UARTQueueTask:* gestisce i messaggi in uscita via seriale.

A ogni task è associata una coda di parametri. Se per esempio si deve inviare un messaggio via radio, il messaggio viene prima inserito in una coda circolare del tipo FIFO (OutQueue) e

successivamente viene effettuato il *post* del task. Per ovviare al possibile fallimento del *post*, nel caso in cui il task sia stato già postato e non è ancora entrato in esecuzione, l'implementazione del task è stata modificata come mostrato in tabella 3:

```
task void OutQueueTask() {

   message_t* e;
   uint8_t len;
   bool test;
   test=FALSE;

   e = dequeue(&OutQueue);
   len = call Packet.payloadLength(e));

   call RadioSendMsg.send(call AMPacket.destination(e),  e, len);

       atomic {
              test=isEmpty(&OutQueue);
              }
   if(test==FALSE) post OutQueueTask();

  }
```

Tabella 3. Il task *OutQueueTask* del livello Network

Con le modifiche evidenziate in tabella 3, alla prima *post* il task viene eseguito e viene tolto dalla coda il *message_t* appena inserito. Se nel frattempo vengono effettuati altri *post* dello stesso task, ed è presente almeno un task che non è ancora andato in esecuzione, essi falliscono. Il controllo sul numero di elementi presenti nella coda *OutQueue* effettuato in *atomic* ovvero come sequenza indivisibile, consentono di far processare al task anche gli altri elementi presenti in coda, il cui *post* è fallito. Questa modifica è stata apportata per tutti i task presenti a livello Network. Si è reso necessario un intervento nella ricezione e inoltro dei messaggi seriali. In Mad-WiSe la programmazione dei nodi avviene tramite l'invio dall'interfaccia SensorViz di appositi messaggi seriali definiti nel *OperatorMsg.h* verso il *sink*. Nella versione precedente questo avveniva in modalità promiscua e solo l'effettivo destinatario del pacchetto processava il contenuto del messaggio. Attualmente è stato aggiunto un campo al messaggio definito dal *OperatorMsg.h* in modo tale che se il *sink* non è l'effettivo destinatario del pacchetto, possa inoltrarlo a quel preciso destinatario via radio.

```
typedef nx_struct OperatorMsg {
      nx_uint8_t queryId;           // Query id
      nx_uint8_t msgType;           // Message type
      nx_uint8_t moteDest;          // Destination Mote Id
      msgBody_t body;               // Message body
} OperatorMsg_t;
```

Tabella 4. Il nuovo campo del *OperatorMsg.h*

Il modulo di gestione del consumo energetico *EnergyEfficency* consente di adattare l'utilizzo della funzionalità radio all'effettivo funzionamento del sistema. Se la query programmata sui nodi richiede comunicazioni radio sporadiche, questo modulo gestisce l'accensione della radio esclusivamente nei periodi di tempo in cui sono previste comunicazioni. L'intervento principale che è stato effettuato su questo modulo riguarda l'introduzione di un evento *setCurTime* all'interfaccia *EEtoNet*. Il nodo *sink* viene acceso per ultimo tra tutti i nodi della rete. Quando viene segnalato l'evento *Boot.booted* il *sink* invia un messaggio in broadcast sulla rete del tipo SETCLOCK, la cui gestione avviene all'interno del *InqueueTask*: nel relativo case dei messaggi di tipo SETCLOCK avviene la chiamata a *EEtoNet .setCurTime(time)* che consente di sincronizzare le accensioni e gli spegnimenti sui nodi. Inoltre l'algoritmo è stato adattato tenendo conto del cosiddetto *TimeDrift* calcolato come la differenza tra il tempo locale del nodo e il tempo che è stato impostato attraverso la *setCurTime.* Le interfacce esposte dal livello Network sono state sottoposte a test definendo un'applicazione contenente le chiamate essenziali di questo livello. In questo modo è stato possibile confrontare la logica di funzionamento dei livelli delle due versioni, e appurarne l'effettiva corrispondenza.

Il secondo livello sottoposto a reingegnerizzazione è stato lo StreamSystem [2]. Questo livello si occupa di costruire degli stream di dati sulle connessioni create nel Network, il livello sottostante. Come tale comprende anche le chiamate che consentono di rilevare i valori forniti dai trasduttori nella fase di sensing. Nel caso specifico, oltre alle modifiche definite a inizio capitolo, si è intervenuti sulla gestione dei task: nella versione precedente, ogni chiamata che consentiva la lettura del valore rilevato, era contenuta in un *task*. Nella versione per TinyOs 2.x è stato definito un task *SSsamplingTask* ed è stata creata una coda di parametri che contiene elementi del tipo *SStaskMSG*. La struttura dati è così definita:

```
typedef struct SStaskMSG{
   uint16_t data;          //il valore rilevato
   uint16_t ts;            //il timestamp
   uint16_t sensor_type;   // l'identificativo del sensore
  } SStaskMSG;
```

Tabella 5. Il nuovo campo del *OperatorMsg.h*

In questo modo ogni richiesta di rilevamento di un valore implica la creazione di un elemento *SStaskMSG* e il conseguente *post* del task *SSsamplingTask* . Inoltre sono stati utilizzati gli stessi accorgimenti descritti in tabella 3 (utilizzo di *atomic* ed eventuale *post* del *Task*) per gestire i *post* multipli del task. Un altro intervento consistente riguarda l'utilizzo specifico di sensorboards *Mts300*[14] con nodi del tipo MicaZ e Iris[14]. La versione di TinyOs utilizzata per lo sviluppo di questa applicazione è la 2.0.2 che non comprende il supporto completo e stabile per gli Iris. Come conseguenza si è reso necessario un wiring adeguato nel file di configurazione dello StreamSystem *SStoAppC* con l'interfaccia apposita *SensorMts300C* . Non è stato possibile utilizzare il componente generico *DemoSensorC* che offre per ciascuna

sensorboard un meccanismo di accesso ai trasduttori in modo trasparente. Anche in questo caso il livello è stato testato con un'applicazione apposita per verificare l'effettiva corrispondenza delle due versioni dal punto di vista funzionale.

Il livello che ha subito meno modifiche è il *QueryProcessor* [5,7]. Sul modulo *QueryProcessorM* e *QueryManagerM*, così come sui relativi files di configurazione *QueryProcessorC* e *QueryManagerC*, è stato necessario intervenire esclusivamente con le modifiche descritte a inizio capitolo. L'unico task presente nel modulo *QueryManagerM*, il *CommandCodeTask*, non richiede *post* multipli: il suo compito è quello di gestire i messaggi di programmazione che il nodo ha ricevuto e, in base ad essi, provvedere alla costruzione degli stream a seconda del tipo di operatore ricevuto, e successivamente allo start e stop dell'esecuzione della query.

L'interfaccia Java SensorViz descritta nel capitolo 2, ha subìto interventi di modifica nella gestione dei pacchetti ricevuti dal nodo *Sink* in seguito al cambiamento della struttura dei messaggi seriali. L'attuale struttura è la seguente:

DestAddr	LinkSourceAddr	MsgLen	GroupID	HandlerID	Payload
2 bytes	2 bytes	1 byte	1 byte	1 byte	Fino a 28 bytes

Figura 10. La struttura dei messaggi seriali in TinyOs 2.x

Le interfacce per la comunicazione seriale da un'applicazione Java sono rimaste invariate nel passaggio a TinyOs 2.x, quindi si è intervenuti solo nel *parsing* del messaggio calcolando gli offset in modo opportuno.

Alla fase di reingegnerizzazione è seguita quella di test sia sul simulatore che sui nodi che ha portato alla conferma che Mad-WiSe 2.0 è funzionalmente equivalente all'ultima versione di Mad-WiSe per TinyOs 1.x.

In particolare sono stati effettuati dei test su rete mista, nodi MicaZ e nodi Iris [14]: anche in questo caso il sistema ha dato prova di solidità e correttezza per un ampia gamma di query di prova.

Il sistema Mad-WiSe si può considerare alla versione 2.0 per TinyOs 2.x, compatibile con nodi MicaZ e Iris sui quali possono essere installate sensorboards *Mts300* [14].

Capitolo 4

Il livello di rete

In questo capitolo descriveremo l'implementazione della funzionalità Multi-Hop in Mad-WiSe. La presenza di questa funzionalità in un'applicazione per reti di sensori aumenta in modo consistente le possibilità di utilizzo della stessa, in modo particolare in ambienti che si caratterizzano per estensione o per proprietà strutturali, come gli edifici, dove la propagazione del messaggi via radio può risultare difficoltosa. Si è reso pertanto necessario introdurre un protocollo di routing [3] nella versione di Mad-WiSe per TinyOs 2.x. In seguito si vedrà come l'implementazione ha sfruttato in modo efficiente la modularità di TinyOs [13]. Questo ha reso l'integrazione nel sistema non problematica e consente che questo livello venga riutilizzato in applicazioni di altro tipo che necessitano della funzionalità Multi-Hop.

4.1 Il routing on demand

Gli algoritmi di instradamento per una rete di sensori devono essere in grado di indicare il percorso corretto che un pacchetto deve seguire. Tuttavia, in contrasto con la necessità delle reti wireless di minimizzare l'utilizzo delle risorse di comunicazione, l'ammontare del traffico necessario a un algoritmo di routing risulta essere molto elevato. Nel caso in esame, si è reso necessario trovare una soluzione che consentisse l'utilizzo di Mad-WiSe in modalità Multi-Hop, cercando di minimizzare l'utilizzo di risorse di memoria per consentirne l'utilizzo anche sui nodi MicaZ: questi hanno una dotazione di RAM di 4 kB [14]. a differenza degli Iris che ne possiedono 8 kB [14]. In particolare si è optato per una strategia di routing di tipo reattivo [3] in cui peculiarmente il calcolo della rotta avviene su richiesta:

se, nell'atto di trasmettere un pacchetto, non esiste una rotta valida, viene invocata una particolare procedura per determinare il corretto instradamento. In questo modo si intensifica l'utilizzo della radio solo durante la richiesta e la scoperta di una cammino di routing, a scapito di un aumento dei tempi di consegna. Dal punto di vista concettuale l'instradamento avviene con la costruzione di una rotta valida, inviando in broadcast ai nodi della rete, un messaggio di *Route Request (RREQ)* [3]. I nodi così vengono a conoscenza della richiesta, da parte di un nodo, di una rotta che ha come destinatario un altro nodo della rete. Quando il nodo destinatario riceve la *RREQ* inoltra un messaggio di *Route Reply (RREP)* [3] lungo la rotta inversa generando in questo modo la rotta valida. Inoltre la logica del routing è stata adattata per rendere funzionalmente equivalente Mad-WiSe, definendo particolari vincoli che verranno illustrati nei paragrafi successivi.

4.2 Le strutture dati.

Le strutture dati utilizzate per le tabelle e per i pacchetti, insieme alle costanti di configurazione del protocollo, sono definite nel file header *MHOPtoNET.h* La tabella di rotta è stata definita come segue:

```
typedef struct {
  am_addr_t  dest;
  am_addr_t  next;
  uint8_t    hop;
  uint8_t    lock;
  uint8_t    RREQpending;
  uint8_t    validity;
  uint32_t   time;
} RouteEntry;
```

Tabella 6. Una entry della tabella di rotta.

La struttura dati *RouteEntry* rappresenta una rotta per il destinatario *dest* dal nodo in cui è allocata la tabella di rotta. Il nodo a cui è necessario inviare il messaggio per raggiungere *dest* è *next.* Il campo *hop* rappresenta la distanza tra il nodo mittente e il destinatario. Il campo *lock* è stato introdotto per adattare il protocollo alla logica di Mad-WiSe. Durante l'esecuzione di una query vengono stabilite delle connessioni tra nodi sulle quali vengono creati degli stream per la trasmissione dei valori rilevati. Queste connessioni restano stabili per tutta la durata dell'esecuzione della query. Ai fini della logica di Mad-WiSe [1], le entries della tabella di rotta coinvolte in connessioni al livello superiore, una volta stabilite non possono essere rimosse per tutta la durata della query. Di conseguenza al relativo campo *lock* viene assegnato il valore 1 in modo che, nel caso in cui ci fosse la necessità di eliminare una entry per fare spazio ad una nuova, le rotte contrassegnate dal campo lock non verrebbero eliminate. Quando la query ha termine, vengono inviati dei messaggi di *Disconnect* per le connessioni precedentemente stabilite. Il campo lock delle entries della tabella di routing, coinvolte nella Connect, viene impostato a 0. Il campo *RREQpending* è un flag che viene impiegato esclusivamente nel nodo mittente e consente di sapere se una *RREQ* per il destinatario *dest* ha ricevuto una *RREP.* Nel caso in cui una *RREQ* non ha ricevuto una *RREP* in un tempo predefinito, è previsto il re invio. Quando un nodo riceve una *RREP* da un certo *dest* il campo *validity* viene settato a 1. Con questo flag è possibile distinguere le rotte valide ovvero quelle costruite con la ricezione di una *RREP*. Il campo *time* rappresenta l'istante temporale in cui è stata inviata la *RREQ* e viene preso in considerazione non solo per effettuare il re invio di una *RREQ* nel caso in cui non sia arrivata una *RREP,* ma anche come criterio di eliminazione di una rotta dalla tabella. Quando si presenta la necessità di inserire una nuova rotta, la tabella di rotta è piena e tutte le rotte risultano essere valide, viene eliminata quella con flag *validity* settato a 0. In alternativa quella con *time* minore, flag *lock*

settato a 0 e non coinvolta in una *RREQ* (flag *RREQpending* non settato a 0), ovvero quella relativamente più datata.

L'altra struttura dati che è stata introdotta in questo modulo riguarda il buffer di messaggi. La *RREQ* viene inviata per far fronte alla necessità di recapitare un messaggio a un destinatario la cui rotta non è nota. Per consentire l'invio di ulteriori messaggi, anche durante la fase di scoperta della rotta, il messaggio che ha richiesto l'invio della *RREQ* viene bufferizzato per poi essere inviato una volta ricevuta la *RREP*. Il buffer di messaggi è definito dalla seguente struttura dati:

```
typedef struct  {
          uint8_t       seq_buf;
          am_addr_t     dest;
          uint8_t       size;
          uint8_t       validity;
          message_t*    pmsg;
          nx_uint8_t    data[PAYLOAD_DATA_LENGTH];
    } MsgBuff;
```

Tabella 7. Una entry del buffer di messaggi

Ad ogni messaggio bufferizzato viene assegnato un numero identificativo *seq_buf* che consente di stabilire l'esatto ordine di arrivo del messaggio nel buffer. Questo consente di gestire il caso in cui sia stata inviata una *RREQ* per quella destinazione e non si sia ancora ottenuta una *RREP*. In questo modo viene preservato l'ordine di arrivo dei messaggi nel buffer per mantenerlo nell'invio alla ricezione della *RREP*. Questa caratteristica è fondamentale in quanto i messaggi di programmazione dei nodi devono essere recapitati in un preciso ordine. Il campo *dest* rappresenta il destinatario del messaggio bufferizzato. Il campo

size rappresenta la dimensione del payload del messaggio bufferizzato. Il campo *validity* è un flag che viene settato nel momento in cui il messaggio viene tolto dal buffer per essere inviato. Nel caso in cui il buffer di messaggi sia pieno viene eliminata la entry con il campo *validity* uguale a 0. Il campo *pmsg* rappresenta il puntatore alla locazione di memoria del buffer del messaggio inviato dal livello superiore, il Network. L'introduzione di questo campo è fondamentale quando si deve segnalare l'evento *SendDone* al livello Network. Il campo *data* è l'effettivo buffer del messaggio. La struttura dati seguente consente il corretto funzionamento dell'invio di messaggi broadcast nella rete.

```
typedef struct {
              uint8_t    seq;
              nx_am_addr_t  src;
} BroadcastEntry;
```

Tabella 8. Una entry della tabella di Broadcast

Ogni nodo, ha una tabella le cui entry sono rappresentate in tabella 8. Il campo *seq* rappresenta un numero sequenziale che contraddistingue ogni messaggio di broadcast. Ogni nodo è provvisto di un proprio *seq* che incrementa all'invio di ogni messaggio broadcast differente. La coppia *seq-src* consente al nodo di verificare se ha ricevuto o meno quel particolare messaggio di broadcast con numero sequenziale *seq* dal mittente *src*. Nel caso in cui il messaggio non sia ancora stato ricevuto, viene aggiunta una entry nella tabella del broadcast e il messaggio viene processato e inviato ulteriormente in broadcast. Qualora il messaggio invece risulti essere stato ricevuto, viene scartato. Questo meccanismo consente di evitare l'effetto "rimbalzo continuativo" dello stesso messaggio all'interno della rete. Le successive strutture dati riguardano la definizione degli header per i pacchetti di tipo *RREQ e RREP*. I messaggi di *RREQ* hanno il seguente formato:

```
typedef nx_struct {
              nx_am_addr_t  dest;
              nx_am_addr_t  src;
              nx_uint8_t    seq;
} mhop rreq hdr;
```

Tabella 9. L'header del messaggio RREQ

Il campo *dest* rappresenta il nodo per il quale si sta cercando la rotta. Il campo *src* è il mittente della richiesta. La RREQ viene inviata in broadcast sulla rete, pertanto si utilizza lo stesso meccanismo dei messaggi in broadcast del livello applicazione. In particolare la tabella di broadcast e il valore *seq* sono condivisi tra le *RREQ* e i messaggi broadcast. In questo modo si limita l'inoltro di una *RREQ* a un solo invio per nodo. Il messaggio del tipo *RREP* ha il seguente header:

```
typedef nx_struct {
      nx_am_addr_t  dest;
      nx_am_addr_t  src;
      nx_uint8_t  hop;
} mhop_rrep_hdr;
```

Tabella 10. L'header del messaggio RREP

I campi *dest* e *src* rappresentano rispettivamente il destinatario e il nodo sorgente della RREQ. Il campo *hop* viene incrementato per ogni nodo che inoltra il messaggio verso la sorgente della *RREQ*. In questo modo il nodo *src* avrà nella sua tabella di rotta il numero esatto di hop che lo separa dal destinatario del messaggio. Questo valore non viene utilizzato attivamente nella logica dell'algoritmo di routing, per esempio per stabilire se esistono rotte con un numero di hop inferiore rispetto ad un'altra rotta, ma è stato inserito in vista di riutilizzo del modulo con un algoritmo la cui logica implementa questa funzionalità. L'ultimo tipo di header del tipo *MHopMsg* viene utilizzato nell'inoltro dei messaggi ricevuti dal livello Network al nodo destinatario attraverso i vari hop, una volta stabilita la rotta. È così configurato:

```
typedef nx_struct {
      nx_am_addr_t  dest;
      nx_am_addr_t  src;
      nx_uint8_t    app;
      nx_uint8_t    broad;
      nx_uint8_t    data[];
} mhop_msg_hdr;
```

Tabella 11. L'header del messaggio MHopMsg

Il campo *dest* e *src* sono rispettivamente il destinatario e il mittente del messaggio. Il campo *app* è stato introdotto per rendere parametrico il modulo. Gli *Active Messages*[78] hanno un campo identificativo del tipo *am_id_t* della dimensione di 1 byte che consente di inviare messaggi con *am_id_t* differente senza conflitti. Il campo *broad* viene utilizzato con i messaggi broadcast e contiene il valore di *seq* incrementato. Il campo data contiene il payload del messaggio da inoltrare al destinatario.

7

8 http://www.tinyos.net/tinyos-2.x/doc/html/tep116.html

4.3 Architettura del livello Network

Prima di esporre i dettagli implementativi della generazione delle *RREQ* e delle *RREP* vediamo nel dettaglio la nuova architettura del livello Network.

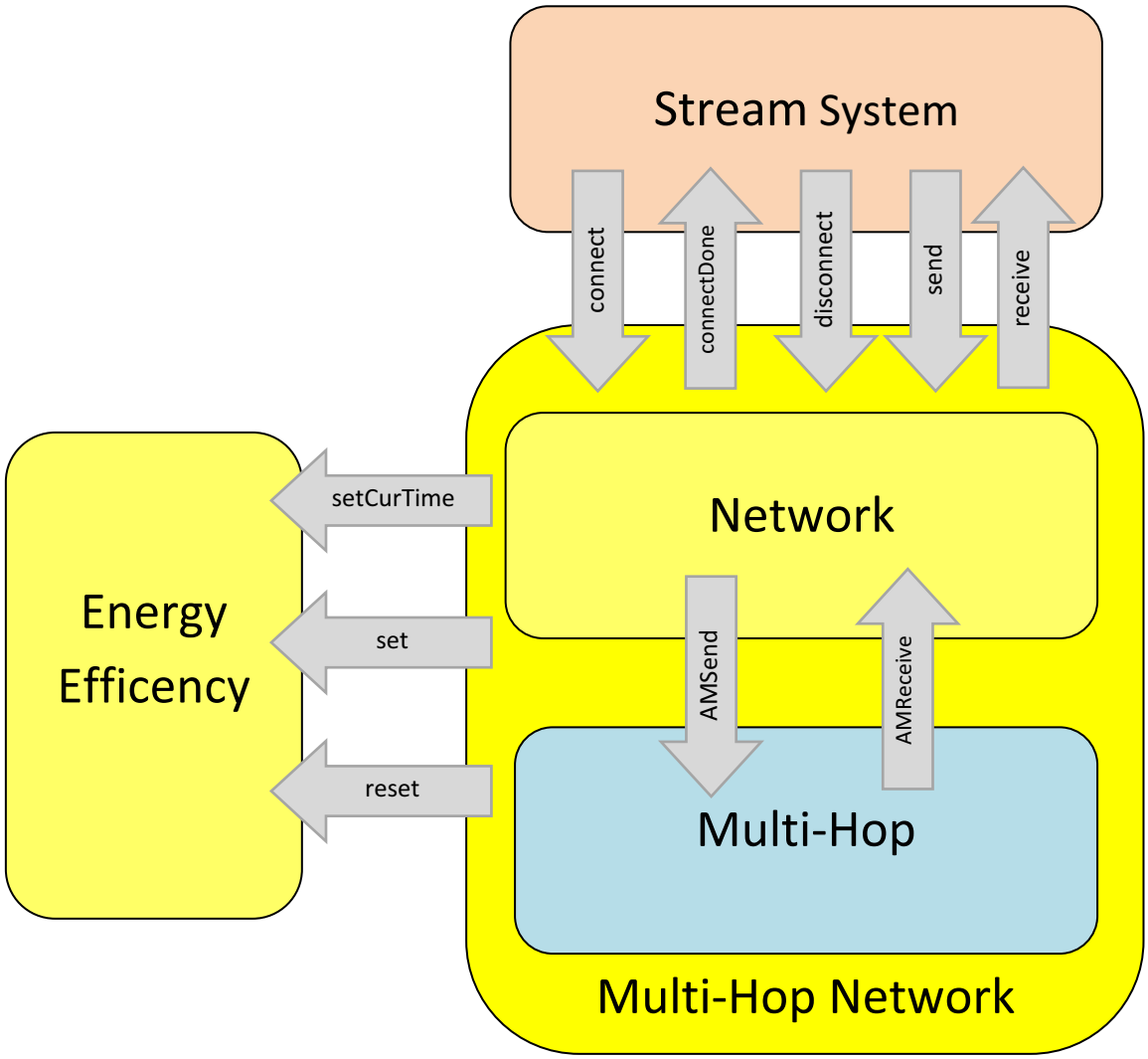

Figura 11. L'architettura del nuovo Multi-Hop Network

La tabella 11 rappresenta l'integrazione del livello Multi-Hop con il Network della versione precedente e il resto del sistema. L'inserimento del livello Multi-Hop tra il Network e il livello MAC fornito da TinyOs è stato possibile grazie alla modularità di questo Sistema

Operativo. Nella versione precedente, il modulo *NetToSSM* utilizzava le interfacce *AMSend* e *Receive* del livello MAC per inviare messaggi via radio. Nel file di configurazione *NetToSSC* il *wiring* corrispondente era il seguente:

```
NetToSSM.RadioSendMsg    -> Radio.AMSend[AM_RADIOMSG];
NetToSSM.RadioReceiveMsg -> Radio.Receive[AM_RADIOMSG];
```

Tabella 12. Il wiring delle interfacce radio prima dell'integrazione con il componente Multi-Hop

La Tabella 12 mostra il wiring per le interfacce radio in assenza del componente Multi-Hop: definendo *Radio* come un alias per l'*ActiveMessageC*, si ha *RadioSendMsg* mappato sull'interfaccia *AMSend* e *RadioReceiveMsg* mappato sull'interfaccia *Receive*. Il tipo di *ActiveMessage* è *AM_RADIOMSG*. Nella versione che include il componente Multi-Hop, l'idea di base è stata quella di definire un modulo, *MHOPtoNET* che implementasse le interfacce *AMSend* e *Receive* per il livello Network, si occupasse di stabilire le rotte inviando messaggi di *RREQ* e *RREP* e di inoltrare il messaggio al destinatario attraverso i vari hop. *MHOPtoNET* è quindi l'unico modulo che utilizza il livello MAC di TinyOS. In seguito alla definizione di questo modulo, nel file di configurazione del Network *NETtoSSC* si è dovuto modificare il *wiring* mostrato in tabella 13:

```
NetToSSM.RadioSendMsg    -> MHOPtoNET.AMSend[AM_RADIOMSG];
NetToSSM.RadioReceiveMsg -> MHOPtoNET.Receive[AM_RADIOMSG];
```

Tabella 13. Il wiring delle interfacce radio dopo l'integrazione con il livello Multi-Hop

Utilizzando le strutture dati descritte nel paragrafo 4.2, si è proseguito con la definizione del modulo *MHOPtoNET* ricorrendo all'implementazione multipla delle interfacce *AMSend* e *Receive*, con la definizione di tre tipi differenti di *ActiveMessage*. *AM_MHOP_RREQ* è il

tipo dei messaggi per la *RREQ*. Viene utilizzato per creare l'istanza *MHSendRREQ* del componente *AMSenderC* e l'istanza *MHReceiveRREQ* per il componente *AMReceiverC*. *AM_MHOP_RREP* è il tipo dei messaggi per la *RREQ*. É necessario per creare l'istanza *MHSendRREP* del componente *AMSenderC* e l'istanza *MHReceiveRREP* del componente *AMReceiverC*. *AM_MHOP_MSG* è il tipo di messaggi impiegato per l'inoltro dei messaggi dal mittente al destinatario, attraverso i vari hop: serve a creare l'istanza *MHSend* del componente *AMSenderC* e l'istanza *MHReceive* del componente *AMReceiverC*. Definire la struttura del modulo in questo modo ci consente di avere eventi distinti per le *Send* e le *Receive* per ogni tipo di messaggio, evitando eventuali conflitti e consentendo il trattamento separato di ogni messaggio inviato e ricevuto. Per l'amministrazione dei messaggi in uscita è stato impiegato un task *SendQueueTask*: ogni messaggio in uscita viene inserito in una coda e viene effettuato il *post* del task. In questo modo si può gestire la richiesta della risorsa radio da parte delle *Send*.

```
task void SendQueueTask() {

       uint8_t i;
       message_t* e;
       uint8_t len;
       bool test;
       test=FALSE;

       e = dequeueMessage(&SendQueue);

       len= call Packet.payloadLength(e);
           if(call AMPacket.type(e)==AM_MHOP_RREQ) //RREQ send
              call SendRREQ.send(AM_BROADCAST_ADDR, e, len);

           if(call AMPacket.type(e)==AM_MHOP_RREP){//RREP send
              call SendRREP.send(call AMPacket.destination(e), e, len);

           if(call AMPacket.type(e)==AM_MHOP_MSG){//MH_MSG send
              call SubSend.send(call AMPacket.destination(e),  e, len);

         atomic {
                  test=isEmptyQueue(&SendQueue);
                }
}
if(test==FALSE) post SendQueueTask();
```

Tabella 14. Il task SendQueueTask

Il modulo di gestione dell'efficienza energetica, rappresentato nella figura 11, è stato integrato considerando che il suo funzionamento deve tenere conto anche dei nodi intermedi in un cammino Multi-Hop. In particolare, quando viene effettuata una *Connect* si comunica al modulo EnergyEfficency che è stata richiesta una connessione con un particolare *streamreate*. Il modulo programma le accensioni e gli spegnimenti della radio in base a questo *streamrate.* Nel momento in cui il livello Network invia un messaggio di *Connect* sulla rete, i valori del modulo EnergyEfficency vengono impostati allo stesso modo anche per i nodi intermedi.

4.4 Generazione e trattamento di un messaggio Route Request

Come accennato nei paragrafi precedenti la *RREQ* viene inviata ogni volta che un nodo necessita di una rotta verso un nodo destinazione [3]. In particolare dal livello Network viene effettuata una *Send* di un messaggio con una chiamata a *RadioSendMsg* il cui wiring non fa più riferimento all'interfaccia *AMSend* del livello MAC ma all'interfaccia *AMSend* nel modulo *MHOPtoNET*. Nell'implementazione del *AMSend* del modulo Multi-Hop si procede come segue:

- Si verifica nella tabella di rotta se esiste una rotta, ovvero con flag *validity* impostato a 1 (vedi tabella 6), per quel destinatario. Nel caso in cui tale rotta non venga trovata, il messaggio proveniente dal livello superiore, viene bufferizzato nella struttura dati la cui entry è stata descritta in tabella 7. Viene quindi creato un messaggio il cui header è del tipo descritto in tabella 9. Il campo *seq* viene incrementato, al campo *dest* è assegnato come valore il destinatario del messaggio giunto dal livello superiore, e al campo *src* corrisponde il TOS_NODE_ID del nodo. I valori di *seq* e *src* vengono

aggiunti alla tabella di broadcast. Nella tabella di rotta si inserisce una entry così composta:

Dest	Next	Hop	Lock	RREQpending	Validity	Time
dest	INVALID NODE ID	INVALID HOP	0	0	0	*time*

Figura 12. La entry della tabella di rotta al momento dell'invio della Route Request

INVALID_NODE_ID e INVALID_HOP sono delle costanti definite nel file *MHOtoNET.h* utilizzate per assegnare dei valori di default in questi campi, in attesa della ricezione di una *RREP*. Il campo *RREQpending* è impostato a 0. Da questo momento in poi il nodo è in attesa di una *RREP* per quella particolare entry. Il messaggio creato in precedenza va nella coda *OutQueue* e viene eseguito il *post* del task *SendQueueTask*.

- Il messaggio di *RREQ* è un messaggio broadcast. Come tale, ogni nodo che lo riceve, deve inoltrarlo a sua volta in broadcast. Quando un nodo riceve il messaggio di *RREQ*, il livello MAC segnala l'evento *ReceiveRREQ.receive* implementato nel modulo *MHOPtoNET*. Il primo controllo riguarda i campi *src* e *seq*. Se risultanono essere già presenti nella tabella di broadcast, il messaggio è stato già ricevuto una volta e quindi viene ignorato. In caso contrario il flusso dell'evento prosegue: i valori *src* e *seq* vengono aggiunti alla tabella di broadcast e se il destinatario della *RREQ* non è il nodo che ha ricevuto il messaggio, viene inserita una entry nella tabella di rotta con questi valori:

Dest	Next	Hop	Lock	RREQpending	Validity	Time
dest	*RREQ src*	INVALID HOP	0	1	0	*time*

Figura 13. La entry della tabella di rotta quando un nodo riceve una Route Request

Nel campo *Next* viene inserito il TOS_NODE_ID del nodo dal quale è stato ricevuto il messaggio *RREQ* . Quando viene ricevuta la *RREP*, essa sarà inoltrata al *RREQ src*: così facendo si costruisce effettivamente la rotta inoltrando i messaggi sul percorso inverso. Al campo *RREQpending* è attribuito il valore 1 perché non è questo il nodo dal quale è iniziata la *RREQ*. Un nuovo messaggio di *RREQ* con gli stessi parametri del messaggio ricevuto ha origine, ma il mittente del messaggio del livello MAC viene impostato a TOS_NODE_ID, in modo tale che chi riceve questo messaggio può individuare il mittente, sempre ai fini dell'inserimento del valore *Next* nella tabella di rotta. Il messaggio viene inserito nella *OutQueue* e viene effettuato il *post* del *SendQueueTask.*

- Nel caso in cui una *RREQ* non riceva entro un periodo di tempo la RREP, essa deve essere inviata nuovamente. La costante DEFAULT_TTL_RREQ definita nel *MHOPtoNET.h* indica per quanto tempo una *RREQ* può rimanere in attesa di una *RREP* senza essere inviata nuovamente. Questa procedura è gestita da un timer periodico *TimerPendingRREQ* il cui relativo evento *TimerPendingRREQ.fired* viene segnalato ogni RREQ_RESEND_DEFAULT_PERIOD (costante definita nel file *MHOPtoNET.h*). Nell'evento *fired* del timer viene chiamata la funzione *ResendPendingRREQ* che scorre la tabella di routing in cerca di entries le cui *RouteReply* non sono state ancora ricevute (campo *RREQpeding* settato a 0) . Quando la funzione *ResendPendingRREQ* trova una entry con queste caratteristiche, la modifica nel campo *time* aggiornandolo con il valore del tempo attuale e viene inviata

una nuova *RREQ* con un nuovo *seq*. Di conseguenza le entries della tabella di rotta dei nodi che hanno ricevuto la precedente *RREQ* non verranno più prese in considerazione in quanto il loro campo *time* risulta minore rispetto a quello della nuova *RREQ* per quel destinatario. La funzione *ResendPendingRREQ* viene invocata solo se ci sono elementi nella tabella di rotta.

4.5 Generazione e trattamento di un messaggio Route Reply

Nel caso in cui il nodo che riceve la *RREQ* sia il destinatario della richiesta, o abbia nella tabella di rotta un instradamento valido per quel destinatario si deve inviare una *Route Reply* [3]. La procedura è la seguente:

- Viene preparato un messaggio di *RREP* il cui header è descritto in tabella 10. I campi *dest* e *src* sono gli stessi della *RREQ* mentre il campo *hop* nel caso in cui il nodo abbia una rotta valida per quel destinatario, assume il valore di *hop*, in caso contrario gli viene assegnato il valore 0. Il messaggio, che ha come destinatario a livello MAC il mittente a livello MAC della *RREQ*, viene messo nella coda di messaggi *OutQueue* e viene effettuato il post del task *SendQueueTask*

- Il nodo che riceve la *RREP*, se non è il mittente della *RREQ* deve modificare la propria tabella di rotta in questo modo

Dest	Next	Hop	Lock	RREQpending	Validity	Time
dest	*RREP src*	New Hop	0	1	1	*time*

Figura 14. La entry della tabella di rotta viene modificata alla ricezione di una RREP

Il campo *hop* viene aggiornato con il valore ricevuto con la *RREP* incrementato di una unità. Il campo *validity* viene impostato a 1 in quanto la entry della tabella è entrata a far parte di un cammino valido. Il campo *Next* contiene il valore *RREQ src* assegnato in precedenza alla ricezione della *RREQ*. Questo valore viene assegnato a una variabile temporanea; successivamente viene aggiornato il campo *Next* con il mittente della *RREP* a livello MAC ovvero *RREP src*. Il messaggio di *Route Replay* viene inviato sempre con il task apposito, impostando come destinatario a livello MAC il valore salvato in precedenza nella variabile temporanea.

- Nel caso in cui il nodo che riceve la *RREP* sia il mittente della *RREQ*, significa che la creazione del cammino di routing è giunta al termine. La tabella di routing viene aggiornata come in figura 14 con l'ulteriore aggiornamento del campo *RREQpending*: è stata infatti appena ricevuta la *RREP* per quel destinatario. Il messaggio bufferizzato in precedenza, prima dell'invio della *RREQ* deve essere tolto dal buffer e inviato utilizzando la rotta appena creata.

4.6 Gestione di messaggi sulla rete

Quando nella tabella di rotta è presente un istradamento valido, i messaggi provenienti dal livello Network, possono essere inoltrati lungo la rete verso il destinatario [3] attraverso la procedura seguente:

- Il messaggio viene gestito all'interno del comando *Amsend.send* . Nella tabella di rotta si verifica che il destinatario del messaggio ha una rotta valida e si acquisisce il valore dell'hop successivo (*Next).* Viene preparato un messaggio il cui header

mhop_msg_hdr è descritto in tabella 11, con tipo *AM_MHOP_MSG*, destinatario *nexthop* e il cui payload viene copiato nel campo *data* dell'header. Nel caso in cui il messaggio ricevuto dal livello Network sia un messaggio di *Connect*, viene settato il flag *Lock* a 1, come illustrato nel paragrafo 4.2. Viceversa, se si tratta di un messaggio di Disconnect, il flag viene settato a 0. Il messaggio viene inserito nella coda *OutQueue* e viene effettuato il *post* del task relativo.

- Nel momento in cui la *Send* del messaggio di tipo *AM_MHOP_MSG* termina, il livello MAC effettua la *signal* dell'evento *SendDone* per la Send in questione. Questo evento viene segnalato anche al livello superiore per confermare al livello Network che la *Send* è stata effettuata. Questo avviene esclusivamente per la *Send* effettuata dal mittente effettivo del messaggio. Gli invii effettuati dagli altri nodi (forward) non sono stati generati dal proprio livello Network ma da messaggi ricevuti a livello Multi-Hop.

- Il messaggio, ricevuto dal nodo successivo (*Next*), viene gestito nella *Receive* relativa ai messaggi di tipo *AM_MHOP_MSG*. Se il destinatario del messaggio del livello Network è il nodo in questione, viene effettuata la *signal* della *Receive* al livello superiore, mandando un messaggio il cui payload è il campo *data* del messaggio di tipo *AM_MHOP_MSG* appena ricevuto. Viceversa, se il nodo in questione non è l'effettivo destinatario, il messaggio viene inoltrato al nodo successivo, acquisendo il valore di *next* per *dest* dalla tabella di rotta. In questo caso se si tratta di un messaggio di *Connect* o di *Disconnect* viene modificato il valore di *Lock* nella tabella di routing per la entry relativa a quel destinatario.

4.5 Gestione di messaggi broadcast sulla rete

L'invio di messaggi broadcast sulla rete da parte del livello Network necessita di un trattamento particolare:

- Il messaggio è gestito all'interno del comando *Amsend.send*. Una volta verificato che il messaggio in questione è un messaggio broadcast, si prepara un messaggio il cui header *mhop_msg_hdr* è descritto in tabella 11, con tipo *AM_MHOP_MSG*, destinatario *AM_BROADCAST* e il cui payload viene copiato nel campo *data* dell'header. Si procede all'incremento del valore *seq* per il nodo in questione e al campo *broad* viene assegnato il valore di *seq*. La coppia *seq – src* si inserisce nella tabella di broadcast. Come si può notare questo tipo di messaggi non fanno uso della tabella di routing. Il messaggio viene inserito nella coda *OutQueue* e viene effettuato il *post* del task relativo.

- Nel momento in cui la *Send* del messaggio di tipo *AM_MHOP_MSG* termina, il livello MAC effettua la *signal* dell'evento *SendDone* per la Send in questione. Questo evento viene segnalato anche al livello superiore per confermare al livello Network che la *Send* è stata effettuata. Questo avviene solo per la *Send* effettuata dal mittente effettivo del messaggio broadcast. Gli invii effettuati dagli altri nodi (forward) non sono stati generati dal proprio livello Network ma da messaggi ricevuti a livello Multi-Hop.

- Il nodo riceve il messaggio e lo gestisce nella *Receive* relativa ai messaggi di tipo *AM_MHOP_MSG*. Per prima cosa controlla nella tabella di broadcast se il messaggio sia stato già ricevuto, altrimenti inserisce i valori di *seq* e *src* nella tabella e prosegue con la gestione. Il messaggio che viene segnalato al livello Network ha come payload il campo *data* del messaggio appena ricevuto. In seguito si procede con la

propagazione del messaggio verso gli altri nodi, preparandone uno nuovo con gli stessi parametri di quello ricevuto, inserendolo nella coda dei messaggi in uscita ed effettuando il *post* del task relativo.

Capitolo 5

Conclusioni

A conclusione di questo studio facciamo alcune considerazioni sui due importanti obiettivi che sono stati raggiunti per il sistema Mad-WiSe: da un lato renderlo compatibile per la nuova versione di TinyOs, la 2.x, dall'altro realizzare un strato di rete Multi-Hop reale ed efficiente. Il punto di partenza, l'analisi delle differenze tra le due versioni del Sistema Operativo per reti di sensori, ha consentito di definire un modello per la reingegnerizzazione di applicazioni per TinyOs 1.x a TinyOs 2.x.: esso infatti non era stato definito in modo esauriente dagli stessi sviluppatori di TinyOs, se si fa eccezione per qualche cenno contenuto nei *TinyOS Extension Proposal* (TEP) [11], ancora in revisione da parte della Community. Si è proseguito col raffinare la conoscenza dei nuovi nodi Iris della Crossbow Technlogy Inc [14], la cui compatibilità è stata appurata individuando le funzionalità di TinyOs non ancora messe a punto per questo tipo di hardware. Un altro strumento utilizzato per portare a compimento questo studio è stato il simulatore TOSSIM[7]. Il suo utilizzo nel processo di testing delle comunicazioni radio e la correttezza della logica dell'applicazione è stato positivo, perché ha permesso di riprodurre il flusso dell'applicazione senza doverla installare sui nodi fisici. Per quanto riguarda l'utilizzo delle connessioni seriali sul simulatore si è riscontrato che queste ultime non sono supportate in modo ufficiale nella release di TinyOs 2.x utilizzata. Per questo motivo è stato necessario ricorrere all'impiego di librerie sperimentali ancora in fase *Beta,* "TOSSIM Live" [9], messe a disposizione da un team appartenente all'ambito degli sviluppatori di TinyOs. TOSSIM Live, pur non funzionando in

[7] www.cs.berkeley.edu/~pal/research/tossim.html

modo sempre soddisfacente, ha consentito di portare a termine il progetto. Gli strumenti messi a disposizione dal sito web di TinyOs, giorno dopo giorno consentono di far progredire questo sistema, a dimostrazione della grande importanza del lavoro svolto dagli sviluppatori all'interno della community. La reingegnerizzazione è stata realizzata sfruttando l'architettura a livelli di Mad-WiSe: ognuno di essi è stato reso compatibile con TinyOs 2.x e testato singolarmente con successo, sia sui nodi che con il simulatore. Completata la reingegnerizzazione si potuto procedere con l'integrazione dei vari livelli e in seguito al test dell'applicazione sul simulatore e sui nodi Iris e MicaZ, configurando il sistema anche su rete mista, raggiungendo così il primo obiettivo.

Il lavoro svolto successivamente è ripreso a partire dall'applicazione compatibile per TinyOs 2.x funzionante su strato di rete single-hop. L'obiettivo era la realizzazione di uno strato di rete Multi-Hop reale ed efficiente. Inizialmente si è provveduto a definire le caratteristiche del nuovo strato di rete, tenendo conto della logica di funzionamento di Mad-WiSe. Una volta ottenuta l'implementazione, il livello Multi-Hop è stato testato sul simulatore e sui nodi, insieme al livello Network già esistente. Appurato il corretto funzionamento, sono stati integrati i livelli restanti. Questa funzionalità, che permette di estendere un gruppo di sensori ben oltre il raggio di copertura di ognuno di essi, è stata testata insieme al resto del sistema con il simulatore TOSSIM attraverso una vasta gamma di query di prova. Attualmente è possibile costruire una rete con un grafo connesso qualsiasi. Gli stessi test sono stati effettuati anche sui nodi reali, dando prova di stabilità e confermando la correttezza delle soluzioni trovate.

Capitolo 6

Indice delle figure e tabelle

Capitolo 7

Bibliografia

[1] Giuseppe Amato, Paolo Baronti, and Stefano Chessa, “MaD-WiSe: Programming and Accessing Data in a Wireless Sensor Networks”, IEEE Eurocon, Belgrado, Serbia & Montenegro, November 2005, pp.1846-1849.

[2] Giuseppe Amato, Paolo Baronti, Stefano Chessa, and Valentina Masi, “The Stream System: a Data Collection and Communication Abstraction for Sensor Networks”, 2006 IEEE International Conference on Systems, Man, and Cybernetics, Taipei, Taiwan, 8-11 October 2006, pp.1449 - 1454.

[3] C. Perkins and E. M. Royer, Ad-hoc on-demand distance vector (AODV) routing. IEEE WMCSA 99, pp. 90–100, 1999

[4] Paolo Baronti, Prashant Pillai, Vince Chook, Stefano Chessa, Alberto Gotta, and Y. Fun Hu, “Wireless Sensor Networks: a Survey on the State of the Art and the 802.15.4 and ZigBee Standards”, Computer Communications, 30 (7): 1655-1695 (2007).

[5] Giuseppe Amato, Paolo Baronti, and Stefano Chessa “MaD-WiSe: a Distributed Query Processor for Wireless Sensor Networks” Technical Report ISTI-2006-TR-39, Istituto di Scienza e Tecnologie dell'Informazione del CNR, Pisa, Italy, November 2006, pp.39

[6] Giuseppe Amato, Paolo Baronti, and Stefano Chessa “Connection-Oriented Communication Protocol in Wireless Sensor Networks” Technical Report ISTI-2005-TR-10, Istituto di Scienza e Tecnologie dell'Informazione del CNR, Pisa, Italy, March 2005, pp.15

[7] Giuseppe Amato, Paolo Baronti, and Stefano Chessa, “Query Optimization for Wireless Sensor Network Databases in the MadWise system”, XV Convegno Nazionale su Sistemi Evoluti per Basi di Dati, SEBD 2007, Fasano, Brindisi, 17-20 June 2007, pp.242-249

[8] David Gay, Philip Levis, David Culler, Eric Brewer. "nesC 1.1 Language Reference Manual". May 2003

[9] Chad Stephen Metcalf , “TOSSIM Live: Towards a testbed in a thread”, Faculty and the Board of Trustees of the Colorado School of Mines, November 2007

[10] MaD-WiSe developer team, Mad-wise web page, http://mad-wise.isti.cnr.it.

[11] TinyOS Extension Proposals TEP, http://www.tinyos.net.

[12] TinyDB, http://telegraph.cs.berkeley.edu/tinydb

[13] TinyOS Documentation, http://docs.tinyos.net.

[14] Crossbow Technology Inc., http://www.xbow.com.

Appendice

In questa appendice saranno illustrate le modalità di installazione di Mad-WiSe 2.0 con le varie opzioni di compilazione.

Compilazione e installazione di Mad-WiSe sui nodi Iris o MicaZ:

Portarsi nella directory `Mad-Wise/` ed eseguire per i nodi Iris:

```
$> make iris
```

La compilazione di *default* comprende il modulo Multi-Hop. È prevista la possibilità di installare Mad-Wise con l'esclusione della funzionalità Multi-Hop. Per fare questo è necessario eseguire:

```
$> MULTIHOP=NO make iris
```

Per motivi di occupazione di RAM è necessario compilare i nodi MicaZ escludendo il modulo MultiHop eseguendo:

```
$> MULTIHOP=NO make micaz
```

Per installare Mad-Wise sui nodi è necessario specificare il tipo di interfaccia seriale. Nel caso in cui si utilizzi la base *MIB520* l'installazione dell'applicazione sugli Iris necessita dell'esecuzione del comando:

```
$> make iris reinstall,X mib520,comZ
```

Per i MicaZ bisogna eseguire:

```
$> make micaz reinstall,X mib520,/dev/ttySZ
```

In entrambi i casi **X** rappresenta il TOS_NODE_ID del nodo e **Z** il numero della porta seriale. Il nodo sink deve avere TOS_NODE_ID uguale a 0.

Una volta completata l'installazione dei nodi si può procedere all'avvio dell'interfaccia grafica SensorViz. Per fare questo si deve accedere alla directory `Scripts/` ed eseguire

```
$> ./rungui.bsh serial
```

Compilazione di Mad-WiSe per il simulatore TOSSIM:

La compilazione di Mad-WiSe per il simulatore TOSSIM prevede l'utilizzo del SerialForwarder, ancora in fase *Beta*, per consentire all'applicazione SensorViz di comunicare tramite seriale con il simulatore. Per ovvi motivi i valori rilevati dai trasduttori sono fittizi in quanto trattasi di una simulazione. Per compilare l'applicazione con questa opzione eseguire:

```
$> make iris sim-sf
```

Oppure per i MicaZ:

```
$> make micaz sim-sf
```

Se si vuole compilare Mad-WiSe escludendo il modulo Multi-Hop, bisogna eseguire:

```
$> MULTIHOP=NO make iris sim-sf
```

Oppure per i MicaZ:

```
$> MULTIHOP=NO make micaz sim-sf
```

Per avviare Mad-WiSe sul simulatore è necessario accedere alla directory `Scripts/` ed eseguire:

```
$> ./tossim.bsh
```

Questo script avvia Mad-WiSe in esecuzione su TOSSIM, l'applicazione java SerialForwarder e l'interfaccia grafica SensorViz. La topologia della rete è definita nel file *topologia.txt* nella directory principale dell'applicazione.

Tabella componenti e interfacce

Principali Interfacce di TinyOS citate

ActiveMessageC	È il nome del componente generico che consente l'utilizzo della funzionalità radio. Si interfaccia all'implementazione specifica della funzionalità radio per ogni piattaforma.
AMPacket	Consente l'accesso alle informazioni relative all'ActiveMessage in un *message_t.*
AMReceiverC	Fornisce le interfacce *Receive*, *Packet*, e *AMPacket*
AMSend	Fornisce le interfacce per l'invio di Active Message
AMSenderC	Fornisce le interfacce AMSend, Packet, AMPacket, e PacketAcknowledgements.
Boot	Fornisce l'evento *booted()* che viene segnalato al completamento della sequenza di boot.
GenericComm	In TinyOs 1.x è il componente che consente l'utilizzo della funzionalità radio
Init	Fornisce un comando *init*() che consente di gestire la fase di inizializzazione.
Packet	Fornisce le interfacce per accedere al payload di un *message_t*
Receive	Fornisce un evento che viene segnalato alla ricezione di un messaggio
SerialActiveMessageC	È l'equivalente dell'ActiveMessageC per l'interfaccia seriale.
SplitControl	È l'interfaccia che consente l'accensione e lo spegnimento della radio
StdControl	In TinyOs 1.x si occupa dell'inizializzazione e dell'avvio dei componenti.
TimeSet	In TinyOs 1.x offre le interfacce per impostare il tempo logico del nodo.

Interfacce, componenti, file di configurazione e file header di Mad-WiSe

Connectionless	Interfaccia che definisce eventi e comandi per le *send* senza connessione
ConnectionOriented	Interfaccia che definisce eventi e comandi per le *send* orientate alla connessione
EEtoNET	Interfaccia del modulo Energy Efficency
EnergyEfficiency.h	File header del modulo Energy Efficency
EnergyEfficiencyM	Modulo Energy Efficency, implementa l'interfaccia EEtoNET
MHOPtoNET.h	File header del modulo Multi-Hop to Network
MHOPtoNETC	File di configurazione del modulo Multi-Hop to Network

MHOPtoNETM	Modulo Multi-Hop to Network
NetToSSC	File di configurazione del modulo Network to Stream System
NetToSSM	Modulo Network to Stream System, implementa i comandi definiti nelle interfacce ConnectionOriented e Connectionless
OperatorMsg.h	File Header che definisce il messaggio seriale di programmazione delle query sui nodi
QueryExecutorC	File di configurazione del modulo QueryExecutor
QueryExecutorM	Modulo Query Executor, contiene il Main dell'applicazione
QueryManagerC	File di configurazione del modulo QueryManager
QueryManagerM	Modulo QueryManager, implementa il task *CommandCodeTask* che prepara la tabella degli operatori in base al messaggio di programmazione ricevuto.
QueryProcessorC	File di configurazione del modulo QueryProcessor
QueryProcessorM	Modulo QueryManager, implementa il task *readDoneTask* che scrive sugli stream in base agli operatori presenti in tabella.
SStoApp	Interfaccia del modulo Stream System to Application, definisce eventi e comandi che permettono la gestione degli stream
SSToApp.h	File header del modulo Stream System to Application
SSToAppC	File di configurazione del modulo Stream System to Application
SSToAppM	Modulo Stream System to Application, consente la gestione degli stream implementando l'interfaccia SStoAPP