UNIVERSITÀ DEGLI STUDI DI MODENA

E REGGIO EMILIA

Facoltà di Ingegneria - Sede di Modena

Corso di Laurea Specialistica in Ingegneria Informatica

Applicazione del database relazionale MySQL alle Wireless Sensor Networks (WSNs): monitoring e memorizzazione parametri ambientali

Relatore

Prof. Sonia Bergamaschi

Candidato

Daniele Caiti

Anno Accademico 2008 - 2009

Alla mia famiglia

Indice

Capitolo 1

Introduzione

Negli ultimi anni, i passi in avanti nella miniaturizzazione, nella progettazione di circuiti a basso consumo e l'ottimo livello di efficienza raggiunto dai dispositivi di comunicazione a onde radio, hanno reso possibile una nuova prospettiva tecnologica: le reti di sensori wireless o WSN (Wireless Sensor Network).

Le reti di sensori in senso generale sono state concepite per interagire con l'ambiente che le circonda. Fino ad oggi la possibilità di cooperare con altri sistemi era limitata in molti casi dalle connessioni via cavo con cui sono realizzate, ora i progressi nelle comunicazioni senza fili hanno eliminato questo vincolo facilitandone la collocazione all'interno dell'ambiente da monitorare. Si vengono a formare in questo modo delle reti in cui ciascun nodo assolve uno specifico compito e collabora con gli altri nodi della rete per raggiungere determinati obbiettivi. Le reti di sensori combinano le capacità di raccogliere informazioni dall'esterno, effettuare elaborazioni e comunicare attraverso un ricetrasmettitore, per realizzare una nuova forma di rete, che può essere installata all'interno di un ambiente fisico, sfruttando le dimensioni ridotte dei dispositivi che la compongono e il loro basso costo.

Una rete radio di sensori è quindi un sistema complesso che nasce dalla cooperazione tra diversi oggetti elementari, detti *nodo sensore*. Si tratta, nella sostanza, di veri e propri *sistemi embedded*, ovvero dispositivi elettronici in grado di svolgere in modo autonomo un certo insieme di operazioni più o meno complesse, di interagire con l'ambiente circostante e di cooperare tra loro per mezzo di opportune interfacce di comunicazione.

Ed è proprio l'interconnessione di questi oggetti in una WSN che amplia incredibilmente le potenzialità del sistema e apre la strada alla realizzazione di servizi avanzati e innovativi a partire dalle funzionalità elementari fornite dai nodi sensori. I possibili campi di impiego delle WSN sono numerosi ed eterogenei e vi

sono molti contesti in cui le WSN trovano impiego. Ad esempio nella tutela ambientale, le WSN possono essere utilizzate per rivelare tempestivamente incendi boschivi, oppure per monitorare l'inquinamento di falde acquifere e laghi o, ancora, per controllare aree a rischio, come discariche, impianti industriali e chimici, centrali nucleari. Un altro possibile impiego è nel campo della domotica, dove si possono sfruttare le WSN per realizzare un'interazione tra ambiente e persona, facendo in modo che l'ambiente reagisca e si adattati alla presenza e alle esigenze dell'utente in modo automatico. Scenari futuri, inoltre, prevedono l'utilizzo delle WSN in ambito medico, attraverso l'impianto nell'organismo di minuscoli nodi sensore in grado di monitorare i parametri vitali della persona e somministrare la terapia farmacologica in modo autonomo, quando necessario. Le potenzialità di questo nuovo paradigma di comunicazione, unitamente alle problematiche ad esso connesse, sono da alcuni anni oggetto di studio nella comunità scientifica internazionale.

Molti lavori di ricerca si sono inoltre focalizzati sulle modalità di *interrogazione* e *aggregazione dei dati* delle WSN. Risulta fondamentale disporre di strumenti in grado di permettere all'utente di richiedere in modo semplice le informazioni di cui necessita dalla rete di sensori. In particolare, sarebbe auspicabile disporre di un sistema in grado di mascherare i livelli più bassi della rete e di funzionare con WSN eterogenee.

In questo lavoro di tesi si è cercato di progettare un sistema di WSN distribuito, governato da struttura centralizzata, in grado di interfacciarsi efficacemente con l'utente, acquisendone le richieste e fornendo i risultati attesi. Il sistema ha l'obiettivo di accettare diverse tecnologie WSN, mascherando all'utente la reale implementazione e le differenze architetturali. A livello di implementazione si è sviluppato un modello complessivo del sistema di interrogazione e gestione delle WSN e si è implementata la parte distribuita del sistema, ovvero la parte direttamente connessa alla particolare rete WSN.

1.1 Struttura della tesi

La tesi si articola nei seguenti capitoli:

Capitolo 2: Wireless Sensor Network

Nel capitolo sarà fornita una descrizione sulle differenti tipologie di WSN, le tecnologie, l'hardware, i protocolli e il software utilizzati per la realizzazione.

Capitolo 3: Texas Instruments EZ430-RF2500

Nel capitolo sarà fornita una descrizione del nodo sensore per la realizzazione della WSN.

Capitolo 4: Sistema di interazione con WSN basato su query

Nel capitolo sarà descritto il modello del sistema complessivo per l'interazione con le WSN. Sarà descritto la parte centralizzate per il recupero delle richieste degli utenti e la visualizzazione dei risultati.

Capitolo 5: Architettura e software Host Wrapper

Nel capitolo sarà descritta la parte distribuita del sistema. Essa si occupa dell'interconnessione diretta alla WSN e alla sua gestione nonché della comunicazione col sistema centrale. Sarà proposta un'effettiva realizzazione del componente e ne sarà descritto in dettaglio il funzionamento

Capitolo 6: Risultati sperimentali

Nel capitolo sono riportati alcuni esperimenti volti a verificare il funzionamento del sistema sviluppato e della WSN ad esso connessa.

Capitolo 7: Conclusioni

Appendice A: Il codice utilizzato

Sono forniti i listati del codice realizzato, corredati da alcuni flow-chart che ne descrivono il funzionamento.

Capitolo 2

Wireless Sensor Network (WSN)

Fino a pochi anni fa, qualsiasi tipo di connessione di rete era costruita con cavi. I vantaggi offerti dalle reti cablate sono ben noti: *nessuna limitazione energetica*, poiché laddove è possibile portare una connessione è possibile prevedere anche una o più linee di alimentazione, *un buon livello di sicurezza*, dato che per prelevare informazioni dalla rete è necessario un accesso fisico al canale e *prestazioni elevate*. Nel contempo però le reti cablate soffrono di gravi limitazioni, tra cui l'evidente difficoltà di realizzazione in ambienti inospitali e gli elevati costi che un cablaggio strutturato comporta. Inoltre una infrastruttura di collegamento cablata risulta rigida, in quanto diviene difficile aggiungere nuovi nodi alla rete o modificare a piacimento la posizione di sensori preesistenti. Le reti wireless, invece, permettono ai terminali di usufruire dei servizi senza richiedere una connessione diretta via cavo, ma sfruttando un collegamento a onde elettromagnetiche.

In questi ultimi anni le reti wireless hanno aperto scenari decisamente innovativi, nei quali gli utenti possono, integrandosi ai classici servizi di telefonia, sfruttare reti wireless dispiegate nel territorio per essere connessi ad Internet. Le reti di questo tipo si sono rapidamente diffuse in quanto offrono una serie di innegabili vantaggi: la *mobilità*, che permette ai terminali di potersi muovere, la *flessibilità* ed i *bassi costi* di realizzazione. Tuttavia anche le reti wireless devono affrontare alcune problematiche. Una di queste è indubbiamente la capacità del mezzo trasmissivo, l'etere, che è unico e condiviso da tutti i nodi connessi. L'esistenza di un unico canale limita necessariamente il numero massimo di utenti che possono usufruire del servizio contemporaneamente. Allo stesso modo, la presenza di più utenti determina una riduzione della velocità di trasmissione, in quanto la capacità del canale trasmissivo deve essere suddivisa tra tutti coloro che ne stanno facendo uso. Non da meno è il problema della sicurezza: in assenza di specifici controlli, risulta facile per un attaccante intercettare le informazioni che viaggiano nell'etere o riuscire ad accedere a servizi anche senza autorizzazione. Occorre inoltre considerare che la qualità della comunicazione può venir influenzata anche da fattori esterni, come interferenze elettromagnetiche e ostacoli in movimento. Infine il

consumo energetico degli apparati di trasmissione radio è tipicamente più elevato di quelli per la comunicazione via cavo.

2.1 Tipologie di reti wireless

Come mostrato in Figura 2.1, una classificazione delle reti wireless può essere fatta in base all'area coperta dal segnale trasmesso dai dispositivi. [1]

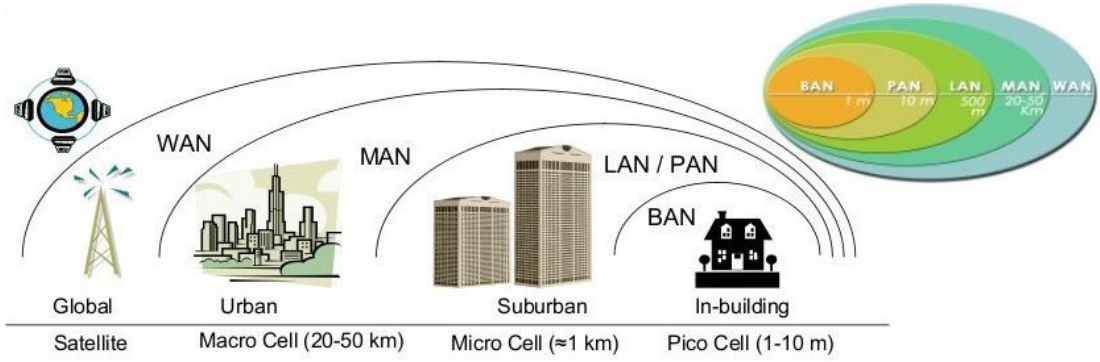


Figura 2.1: Classificazione delle reti wireless

Le reti wireless più piccole vengono chiamate Wireless Body Area Network (WBAN). Una Body Area Network serve per connettere dispositivi "indossabili" sul corpo umano come ad esempio possono essere i componenti di un computer (auricolari); idealmente una rete di questo tipo supporta l'auto-configurazione, facendo così in modo che l'inserimento e la rimozione di un dispositivo dalla BAN risultino essere trasparenti all'utente. Aumentando il raggio d'azione di una rete si passa alle le Wireless Personal Area Network (WPAN). Il raggio di comunicazione delle PAN si aggira ai 10 metri e l'obiettivo di queste reti è quello di consentire a dispositivi vicini di condividere dinamicamente informazioni. Pertanto, mentre una BAN si dedica all'interconnessione dei dispositivi indossabili da una persona, una PAN è operativa nella immediate vicinanze degli utenti. In una Personal Area Network è possibile perciò connettere dispositivi portatili tra loro oppure con stazioni fisse, ad esempio per accedere ad Internet, sincronizzare o scambiare dati e contenuti multimediali su computer portatili, desktop e palmari, etc. Un raggio d'azione maggiore hanno invece le Wireless Local Area Network (WLAN), che consentono la comunicazione tra dispositivi distanti tra loro anche alcune centinaia di metri e risultando così adeguate per la realizzazione soluzioni di interconnessione in ambienti chiusi o tra edifici adiacenti. La famiglia degli standard più diffusi per la realizzazione di questo genere di interconnessioni è IEEE 802.11x. e può essere considerata l'alternativa wireless alle tradizionali LAN basate su IEEE 802.3/Ethernet. [2]

Le Wireless Metropolitan Area Network (WMAN) hanno invece un raggio d'azione di circa 10 Km, fornendo così un accesso a banda larga ad aree residenziali anche piuttosto estese. La tecnologia più accreditata per la realizzazione di reti Wireless MAN è WiMAX, basata sullo standard IEEE 802.16. [3]

Quello sopra proposto non è però l'unico modo possibile per classificare le reti wireless. Ad esempio è possibile confrontare le tecnologie in base al consumo e alla velocità di trasmissione (data-rate). Come è possibile vedere in Figura 2.2 il rapporto tra consumo e velocità di trasmissione risulta essere proporzionale: in generale, alti data-rate implicano consumi elevati e viceversa.

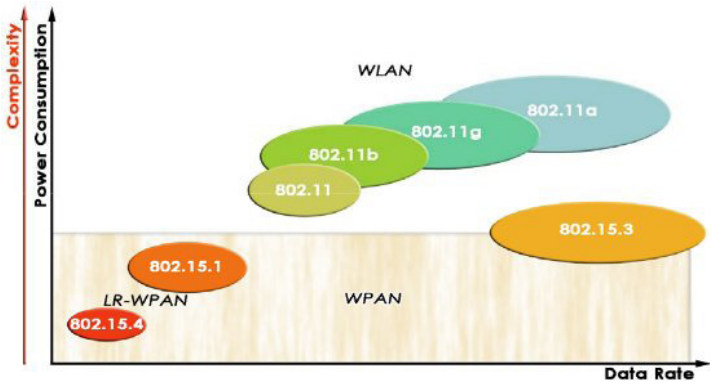


Figura 2.2: Classificazione delle reti wireless in base al consumo e alla velocità

All'interno della categoria delle Wireless PAN, è possibile considerare anche la categoria delle Wireless Sensor Network (WSN). Questo tipo di reti hanno il compito di interconnettere apparati di misura o attuatori al fine di svolgere compiti specifici, come rilevare temperature, umidità, luminosità, oscillazioni, etc. Ogni nodo appartenente alla rete ha la capacità di eseguire calcoli, di catturare dati e di comunicare e cooperare con gli altri nodi della rete.

2.1.1 Le Wireless Sensor Network

Le Wireless Sensor Network sono, letteralmente, reti di sensori senza fili: in realtà il sensore è solo una parte dell'intero sistema che costituisce la WSN, mentre c'è tutto un insieme di hardware e software dedicato affinchè le misure rilevate dal sensore siano scambiate tramite comunicazione radio e messe a disposizione dell'utente. L'idea di avere qualcosa che possa monitorare dei parametri ambientali e che sia capace di integrarsi nell'ambiente in cui si trova nasce nei laboratori della NASA verso la fine del XX secolo: l'ambiziosa intuizione iniziale prevedeva addirittura di realizzare la *Smartdust* [4], la "polvere intelligente", ossia un insieme di piccolissimi elementi in grado di configurarsi e comunicare fra loro costruendo una rete di comunicazione. Nella realtà non si è poi dato seguito a questo progetto, ma è nato un nuovo filone di ricerca, quale quello delle reti di sensori wireless. Nel recentissimo passato grazie allo sviluppo delle nanotecnologie si è tornato a parlare di reti smartdust. Le unità smartdust si potrebbero basare su un sistema di nanoelettronica sub-voltage o deep-sub voltage e includere micro generatori di corrente costruiti con supercondensatori allo stato solido (supercondensatori nanoionici). Anche lo sviluppo di dispositivi radio di dimensioni nanometriche potrà probabilmente essere utilizzato come primo impulso verso lo sviluppo della tecnologia smartdust allo stato pratico [5].

Ad oggi, con dimensioni di qualche ordine di grandezza superiori rispetto alle ancora fantascientifiche smartdust, esistono sul mercato numerosi tipi di hardware per realizzare Wsn. Su ogni dispositivo è alloggiata una parte di controllo, una di comunicazione e una di sensoristica che permettono la costruzione di reti di sensori, in grado di essere dislocate su aree abbastanza vaste e capaci di comunicare tra loro tramite protocolli di comunicazione sviluppati ad hoc.

In questi ultimi tempi le reti di sensori wireless di tipo distribuito, ovvero il processo dei segnali avviene distribuito tra i sensori stessi, stanno sempre più incrementando la loro popolarità per il grandissimo numero di applicazioni possibili nelle discipline scientifiche più diverse.

Tali applicazioni possono essere raggruppate come prima analisi in tre grandi categorie:

• *Monitoraggio ambientale (Enviromental and habitat monitoring).* In campo ambientale le applicazioni possono essere molteplici: è possibile monitorare movimenti di animali oppure studiare particolari habitat, come il mare, il terreno o l'aria impiegando, ad esempio, sistemi di monitoraggio di agenti inquinanti (Figura 2.3). Appartengono a questo settore anche lo studio di eventi naturali catastrofici quali incendi, trombe d'aria, terremoti ed eruzioni vulcaniche

• *Monitoraggio di strutture (Structural Health Monitoring).* Le reti di sensori posizionate sulle strutture rilevano lo stato di salute di edifici, ponti, case sottoposte a sollecitazioni esterne; in alternativa, potrebbero essere utilizzate anche per misurare difetti strutturali di componenti

• *Controllo del traffico veicolare (Traffic Control).* Un sistema di sensori finalizzato al monitoraggio del traffico controlla il passaggio di automobili, analizzare la velocità ed l'intensità del traffico ed individuando eventuali blocchi o situazioni anomale

• *Sorveglianza di edifici (Infrastructure Control).* Questo tipo di reti può essere utilizzato come ausilio per la sorveglianza di centri commerciali o di luoghi a rischio, come stazioni ferroviarie o aeroporti

• *Sorveglianza militare (Militar Control).* Le reti di sensori sono state utilizzate in principio per questo scopo. Le loro applicazioni spaziando dal monitoraggio sullo stato o la posizione delle forze sul campo di battaglia alla sorveglianza di luoghi strategici. Possono inoltre essere dispiegate in luoghi ostili al fine di raccogliere informazioni sugli spostamenti del nemico

• *Monitoraggio di apparecchiature industriali (Industrial Sensing).* I sensori wireless possono essere applicati a macchinari industriali per analizzarne il comportamento dei componenti sottoposti a stress meccanico, migliorarne le performance o prevenirne rotture e guasti

• *Monitoraggio di parametri biologici (Biometric Sensing).* Un sistema di sensori wireless potrebbe essere applicato a pazienti con malattie o in fase di recupero post-operatorio, monitorandone eventuali parametri

fisiologici. Si pensi ad esempio ad un elettrocardiogramma continuo, monitorato in tempo reale in modalità wireless, oppure a degenze domiciliari, con comunicazione e controllo a distanza da parte dei medici

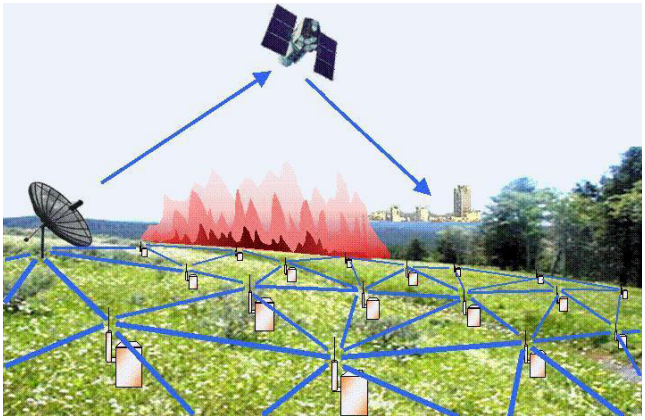


Figura 2.3: Esempio WSN per il monitoring ambientale

Le capacità e le qualità delle trasmissioni tra i sensori di una rete wireless sono però fortemente vincolate alle condizioni degli ambienti in cui sono costruite. Infatti tali ambienti nella maggior parte dei casi non possono essere considerati statici in quanto in essi sono presenti ad esempio ostacoli (muri, mobili, persone...) e campi magnetici variabili (cellulari, apparecchiature hi-fi, dispositivi bluetooth...) che interferiscono con i segnali trasmessi dai singoli nodi. In particolare i fattori che condizionano in modo significativo la qualità delle trasmissioni sono:

• distanza tra i nodi

• ostacoli presenti

• potenza di trasmissione

• disturbi elettromagnetici

• alimentazione fornita

• diversità architetturali dell'hardware

• variazioni climatiche

Il transceiver dei sensori in questione possono fornire alcuni parametri quali ad esempio l'LQI (Link Quality Indicator) e l'RSSI (Received Signal Strength Indicator) [6] che acquistano una notevole importanza qualora vengano utilizzati come indici di grandezze in algoritmi complessi per WSN. L'RSSI ad esempio varia in funzione della distanza e quindi può diventare un ottimo alleato nei campi del tracking e della localizzazione, dove sono proprio le distanze, fisse o variabili, fra i nodi a farla da padrone.

Tali parametri essendo direttamente correlati alla qualità della trasmissione, sono soggetti, per i motivi sopracitati, a variazioni e condizionamenti esterni indesiderati che in molte occasioni li rendono inaffidabili. E' evidente quindi la necessità di studiare un modo per calibrare questi indici istante per istante, fra un nodo e l'altro di una WSN, cercando di escludere le variazioni a cui sono soggetti per interazioni esterne ed avvicinarsi sempre più ad un valore esatto privo di disturbi.

2.2 I dispositivi hardware utilizzati nelle WSN: i nodi sensore

Per comprendere appieno le potenzialità e i limiti di una WSN è necessario descrivere almeno genericamente l'architettura del nodo sensore. Una rappresentazione di principio è data in Figura 2.4, dove si evidenziano cinque blocchi funzionali: trasduttori, attuatori, unità di elaborazione e controllo, memoria ed unità di comunicazione.

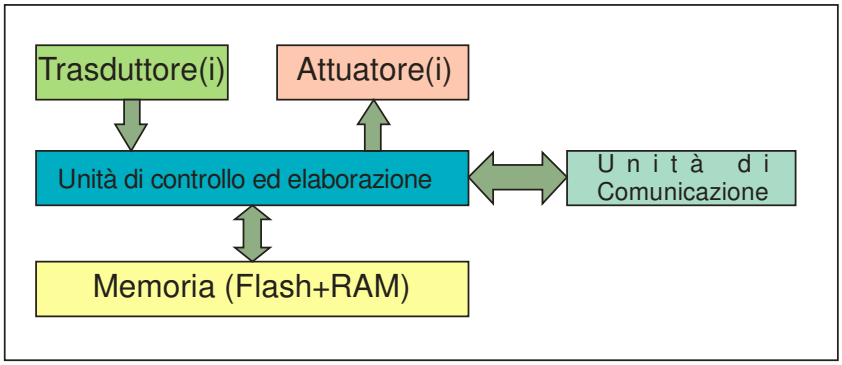


Figura 2.4: Schema nodo sensore

Trasduttori ed attuatori. Il trasduttore è un dispositivo in grado di misurare delle grandezze fisiche o ambientali di diversa natura trasformandole in un segnale elettrico opportuno. Poiché questo tipo di operazione, in inglese, viene detta "sensing," i trasduttori vengono spesso chiamati *sensori*. Questa terminologia può originare confusione, in quanto il termine sensore viene comunemente utilizzato sia per l'unità di sensing vera e propria, sia per il sistema embedded (nodo sensore) nel suo complesso. I sensori che si possono trovare in commercio sono numerosissimi e includono trasduttori di temperatura, umidità, intensità luminosa e pressione, come anche sensori per il rilevamento di fumo, sostanze tossiche (disperse nell'aria o nel suolo) o radioattività, telecamere, sensori di prossimità e così via. Un sensore, quindi, è in grado di rivelare e/o misurare grandezze ambientali, senza tuttavia poterle modificare. Un ruolo duale è svolto dagli attuatori, dispositivi in grado di agire sull'ambiente circostante in diversi modi. Esempi di attuatori sono bracci meccanici, valvole, irrigatori, ma anche sirene d'allarme, altoparlanti, videoproiettori, caloriferi, ecc.

Un nodo sensore può comprendere un numero variabile di trasduttori e attuatori, anche di diversa natura, che possono essere combinati e utilizzati nei modi più disparati. L'equipaggiamento del nodo sensore, quindi, ne determina le funzionalità, ma anche il costo, il consumo energetico e l'ingombro.

Unità di elaborazione e controllo. L'unità di elaborazione contiene un microprocessore (CPU), che fornisce l'intelligenza necessaria al dispositivo per operare in modo autonomo. Il ruolo della CPU può essere svolto da diverse tipologie di circuiti logici. Comunemente la scelta ricade su un microcontrollore a basso consumo, ma è possibile impiegare anche un FPGA (Field Programmable Gate Array), un DSP (Digital Signal Processing) o un ASIC (Application Specific Integrated Circuit), anche a supporto del microprocessore per diminuirne il carico di calcolo.

Oltre al microprocessore, l'unità di elaborazione contiene un convertitore anologico-digitale (ADC), utilizzato per tradurre i segnali elettrici generati dai trasduttori in forma digitale in modo da consentirne l'elaborazione software. Analogamente, è spesso presente un'unità duale, ovvero un convertitore digitale-analogico (DAC), che trasforma segnali numerici generati dal microprocessore in segnali elettrici utilizzati per pilotare gli attuatori.

Memoria. Il microprocessore è spesso integrato con alcuni blocchi di memoria ROM (Read Only Memory) e RAM (Random Access Memory), utilizzati per ospitare il codice eseguibile del sistema operativo e dell'applicazione, nonché i dati acquisiti dai sensori ed elaborati dall'applicazione. In realtà la ROM, spesso è programmabile e fisicamente realizzare con memoria di tipo Flash, cosicché il sistema operativo e il software in esecuzione sul nodo sia modificabile. La gestione e l'utilizzo delle memorie è una fonte di consumo energetico, pertanto i blocchi di memoria integrati hanno capacità ridotte, limitate a poche decine di kbyte. Alcune piattaforme, tuttavia, possono essere dotate di memorie Flash aggiuntive, connesse al microprocessore per mezzo di interfacce seriali. Queste memorie, solitamente di capacità superiore rispetto alle memorie integrate con il microprocessore, possono essere utilizzate per contenere parametri per la configurazione del nodo o altri dati. Chiaramente, l'utilizzo delle memorie aggiuntive aumenta la potenzialità del nodo, ma influisce negativamente sui consumi energetici.

Unità di comunicazione. La comunicazione tra nodi sensori si realizza tipicamente per mezzo di segnali radio anche se, per alcune applicazioni, sono possibili soluzioni alternative che impiegano comunicazioni ottiche o ultrasuoni, come nel caso delle reti di sensori subacquee. Solitamente tra tutti i componenti del nodo, il chip radio è il dispositivo che consuma la maggior parte dell'energia. Per ridurre il costo e il consumo energetico dei nodi, si utilizzano tipicamente modulazioni ben consolidate e di bassa complessità, a discapito della capacità trasmissiva che è spesso limitata a qualche decina di kbit/s. Per limitare ulteriormente il costo finale del dispositivo, la modulazione radio avviene normalmente nelle bande comprese tra gli 868-870 MHz o nelle bande ISM (Industrial Scientific Medical) attorno ai 900 MHz e ai 2,4 GHz, per le quali non è richiesta licenza governativa.

Software. Oltre all'hardware, la piattaforma deve ospitare il software necessario a gestire la comunicazione tra i diversi nodi e a realizzare i servizi più avanzati. Sfortunatamente, le peculiarità delle reti di sensori e le caratteristiche dei nodi rendono difficilmente riutilizzabile il software disponibile in commercio e richiedono lo sviluppo di soluzioni progettate appositamente per la piattaforma utilizzata e per la specifica applicazione da realizzare. Questa argomentazione vale anche per il sistema operativo, a cui è richiesto di soddisfare i seguenti requisiti:

• ridottissima occupazione di memoria

• basso consumo di energia durante l'esecuzione dei processi

• consumo quasi nullo durante lo stato di inattività (idle)

• gestione della concorrenza (accesso simultaneo di più thread alla stessa risorsa)

• supporto efficiente (in termini di consumo energetico) ai protocolli di rete

• facile accesso alle funzionalità di basso livello della piattaforma per mezzo di interfacce astratte

Generalmente, lo sviluppo del sistema operativo e delle applicazioni avviene per mezzo di linguaggi di programmazione derivati dal C, che vengono quindi compilati per lo specifico microprocessore del nodo sensore utilizzato. Tuttavia, è quasi sempre possibile ricorrere alla programmazione di alcuni moduli nel linguaggio Assembly del microprocessore, se si desidera spingere al massimo le prestazione del codice. Come per il sistema operativo, il software che definisce i servizi più avanzati della WSN deve occupare pochissimo spazio di memoria e deve risultare quanto più possibile efficiente in termini di consumi energetici. Questo comporta la necessità di progettare l'applicazione in modo da minimizzare quanto più possibile l'utilizzo delle diverse interfacce (radio, trasduttori, attuatori) e del processore, cosa che si traduce spesso in un tradeoff tra efficienza energetica del sistema e qualità del servizio offerto.

2.2.1 Le piattaforme hardware disponibili per i nodi sensore

Attualmente possono essere reperiti sul mercato svariati tipi di nodi sensore. I più comuni sono:

- Mica Family (Mica Mica2 MicaZ) - Crossbow Technology Inc [7].
- Telos Family (Telos, Telosa, Telosb, TmoteSky) - Moteiv Corporation [8].
- BTnode – ETH Zurigo[9].
- EyesIFX Family (EyesIFX EyesIFXv1 EyesIFXv2) - Ember Corporation [10].
- SquidBee – DiGI [11].

- Firefly - Carnegie Mellon University[12].
- EZ430-RF2500 – Texas Instruments [13].

(Le figure riportate nelle successive pagine sono dimensione reale)

Mica Family. Nella prima versione *MICA*, è stato il primo nodo sensore (qui chiamato *mote*) per WSN ad essere sviluppato. Nato nell'università di Berkeley nell'anno 2002, viene ora commercializzato dalla società Crossbow Technology Inc, dalla sua architettura si sono sviluppate la maggiori famiglie di sensori al momento presenti. Le successive evoluzioni della famiglia MICA, sono i sensori MICA2 di dimensioni 6 cm x 3 cm x 2 cm (Figura 2.5), MICA2DOT e MICA-Z. Tutte e tre le versioni montano un processore ATMega128L, non hanno sensori integrati nella piattaforma hardware di base e sono impiegabili per il rilevamento di temperatura, umidità, accelerazioni, pressione o campi magnetici. Ciò che differenzia le varie versione dei mote è il componente radio: nei MICA è utilizzato il chip TR1000, che offre un data-rate massimo di 40 Kbits/s, nei MICA2 è installata la radio CC1000, con un data rate massimo di 38.4 Kbits/s, mentre nei MICA-Z è presente il più recente integrato CC2420, che può raggiungere velocità trasmissive di 250 Kbits/s.

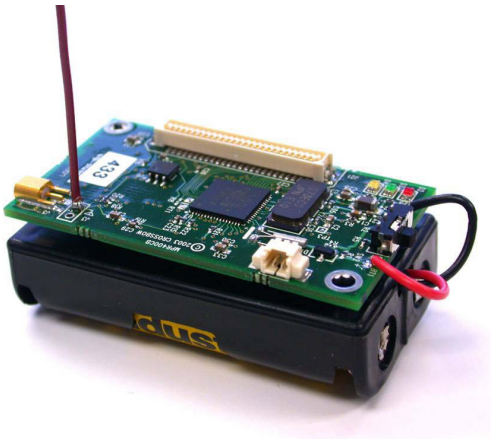


Figura 2.5: Nodo sensore MICA2

Telos Family. Questi motes, tra loro architetturalmente molto simili, sono stati sviluppati sempre presso l'Università della California, Berkeley e prodotti dalla Moteiv Corporation. Sono la naturale evoluzione della prodotti della famiglia Mica, da cui derivano. La principale differenza è l'adozione di un nuovo microcontrollore. Si tratta, infatti, di un MSP430 della Texas Instruments (MSP430F1611), che presenta un'architettura a 16 bit di parallelismo. È dotato di 10kB di memoria RAM, 48kB di memoria flash, di un convertitore ADC a12-bit e di un controllore DMA (Direct Memory Access). Il ricevitore è lo stesso utilizzato dai MicaZ, vale a dire il CC2420 della Chipcon. I nodi della famiglia Telos sono pertanto conformi alle specifiche dello standard IEEE 802.15.4 e compatibili con lo stack radio Zigbee (si rimanda alla Sezione 2.4). Il nodo presenta un'antenna a 2.4GHz integrata sullo stampato che permette di trasmettere fino a 125m

in campo aperto. Ogni nodo è inoltre dotato di 1Mbyte di memoria flash esterna e di alcuni sensori (umidità, temperatura ed intensità luminosa). Inoltre ciascun nodo è identificato tramite un indirizzo IEEE 802.15.4 a 64 bit univoco ed è dotato di un convertitore seriale-USB per facilitarne la programmazione ed il debug. In Figura 2.6 si mostra il modello TMoteSky.

Figura 2.6: Nodo sensore TMoteSky

BTnode. I BTnode (Figura 2.7) sono sensori che utilizzano un processore ATMega128L e una scheda radio CC1000, che opera nella banda ISM 433-915 MHz. Progettato dall'ETH di Zurigo, è stato realizzato in tre versioni: la più recente, chiamata semplicemente BTnode rev3, è la piattaforma che ha avuto più successo. La particolarità del di questa piattaforma consiste nel possibilità di usufruire di un doppio sistema di trasmissione radio, disponendo anche di una radio Bluetooth.
Il sistema Bluetooth integrato può supportare 4 *Piconet*, ognuna delle quali può comandare al massimo 7 slave. Il BTnode non presenta alcun sensore integrato ma la scheda supporta l'inserimento di sensori di luminosità, temperatura, accelerazione e suono costruiti da TAOS (Texas Advanced Optoelectronic Solutions).

Figura 2.7: Nodo sensore BTNode

EyesIFX Family. Il modello EyesIFX, come si può vedere in Figura 2.8, ha dimensioni molto ridotte. Il nodo monta un processore MSP430 (MSP430F1611) e un modulo radio TDA5250 della Infineon, che opera

nella banda degli 868 MHz e offre una velocità massima di trasmissione di 64 Kbit/s. La piattaforma integra sensori di luminosità e di temperatura. Il sensore di temperatura lavora in un range molto ampio, che va da −30 a +100 C° con un errore di circa ±2 C°. L'hardware è stato progettato e costruito per offrire una alta efficienza energetica, i consumi sono ridotti ma a discapito di una velocità di trasmissione più bassa, limitata dal firmware a circa 20Kbit/s.

Figura 2.8: Nodo sensore EyesIFX

SquidBee. I nodi Squidbee hanno dimensioni piuttosto elevate (Figura 2.9). Si basano su un processore Atmega168 e il modulo radio è l'XBee prodotto dalla DiGi. La piattaforma Squidbee è progettata per la costruzione di WSN che rilevano temperature, luminosità e umidità grazie a sensori esterni, non integrati nella scheda principale. Il nodo sensore è formato da una scheda madre (modello *Arduino*)e da un circuito di interfacciamento tra il modulo radio XBee, progettati dalla Libelium. Il modulo radio XBee può operare nelle bande di frequenza 2.4 GHz, 915 MHz e 868 MHz e ha un data-rate variabile da 20 Kbits/s ad un massimo di 250 Kbits/s. Esiste anche la versione del modulo radio XBee-Pro, che estende il raggio d'azione della radio ad 1 Km. I progetto dei nodi è open, ovvero ogni parte del nodo è accessibile e può essere studiata, cambiata e personalizzata utilizzando gli schemi circuitali e il codice sorgente. Questo tipo di piattaforma non usa alcun tipo di sistema operativo: il codice viene compilato e caricato direttamente all'interno della EEPROM per essere caricato all'avvio direttamente dal bootloader.

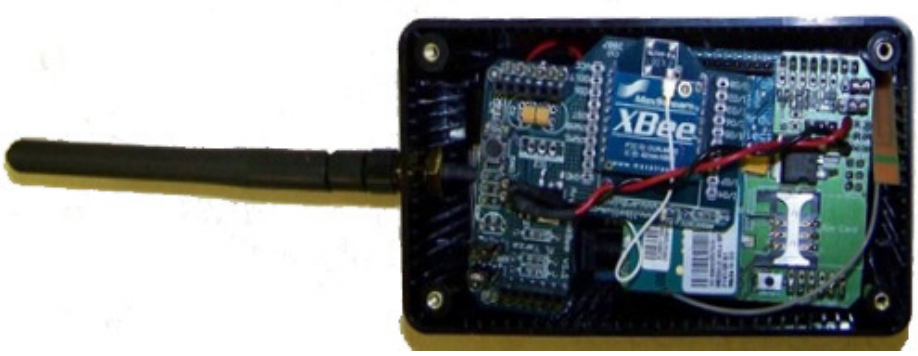


Figura 2.9: Nodo sensore SquidBee

Firefly. Il nodo sensore Firefly, visibile in Figura 2.10, è sviluppato dalla Carnegie Mellon University ed è dotato di un processore Atmega128L e un modulo radio CC2420. Al suo interno sono integrati in una scheda i sensori di luminosità, temperatura, audio e accelerazione. I consumi del nodo sono estremamente ridotti: la

durata prevista delle batterie è di quasi due anni. Questo è reso possibile da complessi meccanismi di sincronizzazione tra i vari nodi dell'intera rete, che permettono di poter mandare i sensori in fase di sleep molto rapidamente e minimizzando i periodi di idle. Il raggio di azione di questi dispositivi è di circa 50-100 metri e il data-rate massimo è pari a 250 Kbps.

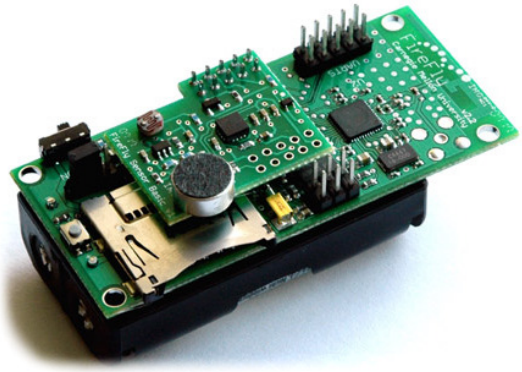


Figura 2.10: Nodo sensore FireFly

EZ430-RF2500. Il prodotto della Texas Instruments EZ430-RF2500 (Figura 2.11), si basa sul microcontrollore MSP430 (MSP430F2274) che integra 32KB di memoria Flash, 1KB di Ram e si basa su un'architettura RISC a 16bit. Integra inoltre al suo interno un sensore di temperatura ambientale. La comunicazione radio è affidata al dispositivo CC2500, operante nelle frequenze di 2.4 GHz. La velocità di comunicazione è programmabile, con velocità di picco di 500kbps. La normale velocità configurata per ridurre l'assorbimento elettrico è di 250kbps. Peculiarità di questo dispositivo è l'utilizzo di un protocollo di rete a bassissimo consumo e sviluppato dalla stessa Texas Instruments: SimpliciTI (si rimanda alla sezione 2.5). Il dispositivo non fa uso di un vero e proprio sistema operativo, ma il codice del firmware (in linguaggio C standard) è sviluppato e compilato tramite l'ambiente IAR Embedded Workbench e caricato sul nodo, utilizzando l'apposita interfaccia USB.

L'eZ430-RF2500 è il nodo sensore utilizzato in questo lavoro di tesi, si rimanda alla sezione 2.6, per una dettagliata trattazione del dispositivo fisico.

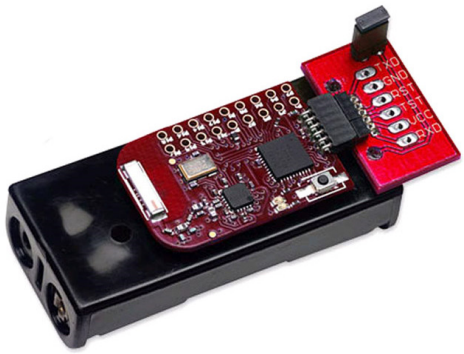


Figura 2.11: Nodo sensore EZ430-RF2500

2.3 I sistemi operativi per i nodi sensore

Un sistema operativo per sensori deve avere alcune caratteristiche di base: deve avere ridotte dimensioni, basso consumo durante l'elaborazione, consumo pressoché nullo durante lo stato di idle del dispositivo, deve gestire la concorrenza, deve implementare protocolli di rete a seconda della periferica di rete utilizzata e tali protocolli devono essere poco dispendiosi in termini di energia (il TCP/IP e' in genere inapplicabile); il sistema operativo, inoltre, deve fornire un'astrazione per i dispositivi hardware (sensori e attuatori) montati sul sensore.

Si distinguono due approcci allo sviluppo di sistemi operativi per sensori:

- Sviluppare un sistema i cui componenti vengono compilati insieme all'applicazione (come TinyOS 1.x [14]).
- Sviluppare un sistema che includa i tradizionali strati di software dei sistemi general purpose in versione ridotta (ad esempio Contiki [15]).

Nel primo caso si ha una sola singola applicazione in esecuzione in un dato momento. Tuttavia tale sistema permette di avere bassissimi consumi (non esistendo in genere *context switch* dal momento che gli scheduler seguono una politica *run to completion*) e sistemi molto piccoli (essendo realmente inserite nel sistema solo le funzionalita' richieste). Lo svantaggio derivante da tale approccio è la limitata versatilità e i seri vincoli di riconfigurabilità dell'applicazione.

Nel secondo caso è difficile tenere i consumi e le risorse impiegate sotto controllo, ma si guadagna in versatilità potendo eseguire più applicazioni in contemporanea.

2.3.1 TinyOS 1.x

TinyOS [14] è il sistema operativo sviluppato per la tecnologia *Motes* dall'università californiana di Berkeley. E' scritto in linguaggio nesC [16] (si vede sezione 2.3.2) e offre un modello di programmazione basato su eventi (l'applicazione è composta da diversi componenti la cui elaborazione viene avviata al verificarsi di un particolare evento). Questo approccio si contrappone al tradizionale paradigma basato su *stack* e *switching* del contesto di esecuzione. Quando il sistema è in stato di *idle* non esegue alcuna operazione, consumando minime quantità di energia. In questo modo pur permettendo la concorrenza, evita l'attività di *contextswitching* (che consuma energia) tipica dell'approccio tradizionale.

Il componente principale (l'unico sempre presente in ogni applicazione) è lo *scheduler*, questo manda in esecuzione i task dei diversi componenti secondo una politica FIFO *run to completion* (un task non può interrompere un altro task). Lo scheduling ha due livelli di priorità: normale, per i task, e quello più alto per gli eventi, che possono interrompere i task.

Componenti. Il modello a componenti di TinyOS è dotato di *command handlers* e *event handlers*. I componenti comunicano tra loro invocando *comandi* (gestiti dal command handlers) e sollevando *eventi* (gestiti dall'event handlers). Comandi ed eventi vengono eseguiti al livello alto di priorità dello scheduler. A questi si aggiunge una serie di task, che vengono eseguiti a livello basso di priorità.

Ogni componente inoltre contiene un *frame*. Questa è l'area dati del componente e viene allocata staticamente. Tipicamente gli eventi vengono sollevati da componenti più vicini all'hardware verso componenti meno vicini, mentre i comandi vengono invocati in verso opposto.

I componenti possono essere suddivisi in tre categorie:

Hardware abstractions, questi componenti mappano le funzionalità fornite via software sulle funzionalità fornite dall'hardware creando un'astrazione dello stesso utilizzabile dai moduli superiori;

Synthetic hardware, questi moduli simulano il comportamento di hardware più sofisticato di quello realmente presente sul sensore;

High level software component, questi componenti sono quelli di livello più alto e si occupano di eseguire algoritmi che prescindono dal particolare hardware.

Ad esempio un componente che legge/scrive un bit su un canale radio è un'astrazione hardware, mentre un componente che compone 8 bit e li invia a quelli di livello superiore è un hardware sintetico, mentre il componente che implementa il protocollo di routing è un componente di alto livello.

Active Messages. E' la tecnologia di rete utilizzata da TinyOS, si tratta di un'architettura che si avvicina molto a quella a comandi e componenti di programmazione del singolo sensore. Questo sistema permette di inviare messaggi a tutti o a un singolo nodo tra i nodi vicini (protocolli di routing sono implementati ai livelli superiori, quindi ha solo un meccanismo di indirizzamento verso i nodi vicini). Active messages una tecnologia molto leggera, infatti non specifica meccanismi *connection oriented*, ogni pacchetto è un'entità indipendente. Esso contiene l'identificatore di un handler da richiamare sulla macchina di destinazione e il payload del pacchetto. Questo sistema permette di evitare il complesso utilizzo di buffer delle implementazioni del protocollo TCP/IP.

Il principale svantaggio dell'approccio è che tutti i nodi comunicanti devono avere lo stesso software o quantomeno implementare un componente che definisca lo stesso handler.

2.3.2 NesC

NesC [16] è una variante del linguaggio C sviluppato per la programmazione dei motes, sviluppato all'università di Berkeley ed utilizzato per la scrittura del sistema operativo TinyOS. Per alcuni versi questo linguaggio è un'estensione del C (implementa un *modello a eventi*), mentre per altri restringe il linguaggio C (limita diverse operazioni riguardanti i puntatori).

In accordo con la politica di utilizzo delle risorse di alimentazione del sensore (sleep per la maggior parte del tempo, awake solo durante la fase di elaborazione) nesC permette di definire un'elaborazione *event-driven*: i componenti di un'applicazione vengono mandati in esecuzione solo quando si verificano gli eventi associati a ciascun componente.

Modello a componenti. Un'applicazione nesC è un insieme di *componenti* collegati tramite *interfacce* (tra i componenti che compongono un'applicazione ci sono anche quelli del sistema operativo stesso). Questo approccio separa la costruzione dei componenti dalla composizione degli stessi.

Ogni componente è specificato dalle *interfacce* che pubblica e da quelle che utilizza. Ogni interfaccia è bidirezionale e modella un servizio offerto/utilizzato dal componente. Le interfacce sono composte da un insieme di comandi e da un insieme di eventi. Per ogni interfaccia fornita da un componente, quest'ultimo implementa i comandi, mentre l'utilizzatore implementa il comportamento relativo agli eventi.

Per ogni interfaccia utilizzata da un componente, quest'ultimo implementa gli eventi e invoca i comandi. Questo meccanismo è ridondante, tuttavia raggruppare comandi ed eventi relativi a un servizio in interfacce rende il codice più leggibile.

Comandi ed eventi al momento del linking statico vengono tradotti in chiamate a funzioni, quindi semanticamente la differenza è minima.

Esistono due tipi di implementazione di componenti: i *moduli* e le *configurazioni*. I moduli implementano le funzionalità delle interfacce del modulo attraverso codice C con alcune estensioni. Il modulo implementa inoltre gli eventi e i comandi in modo molto simile a come vengono implementati i sottoprogrammi in C, ad esclusione del fatto che, prima della definizione dell'evento o del comando, bisogna inserire il nome dell'interfaccia relativa, e per alcune macro utilizzate come valori di ritorno.

All'interno di un'implementazione è possibile sollevare eventi (tipicamente verso moduli di livello più alto) tramite la parola chiave *signal*, mentre è possibile invocare comandi (tipicamente su componenti più vicini all'hardware) tramite la parola chiave *call*. Ogni modulo contiene il suo stato sotto forma di variabili locali dichiarate alla stessa maniera delle variabili in C.

Le configurazioni implementano componenti dichiarando una serie di sottocomponenti e definendo i collegamenti tra le interfacce di questi sottocomponenti. Ogni applicazione è composta da una configurazione globale che definisce la connessione dei vari componenti.

2.3.3 TinyOS 2.x

TinyOS 2.x [17] si presenta come una riscrittura completa del TinyOS 1.x al fine di riorganizzarlo ed ottimizzarlo, alla luce dell'esperienza maturata durante la stesura della versione precedente. Di conseguenza TinyOS 2.0 offre, oltre alle funzionalità della versione 1.x, svariate migliorie architetturali. E' importante specificare che le due versioni non risultano compatibili a livello di software eseguibile, pertanto programmi scritti per la versione 1.x non funzionano sulla nuova versione, esattamente come per la situazione inversa. In particolare:

1. Lo *scheduler* del sistema operativo è ora un componente modulare che può pertanto essere sostituito o modificato con facilità

2. La struttura architetturale basata su *livelli di astrazione* è stata rivisitata, al fine di ottimizzare il supporto all'hardware utilizzato

3. La sequenza di *Bootstrap* è stata modificata 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 con successo

4. Le funzionalità *radio* sono normalmente non operative dopo il boot del sistema. L'interfaccia SplitControl, che gestisce le comunicazioni wireless, richiede uno start esplicito ed esclusivo. Si evitano in questo modo sprechi energetici, in caso di non utilizzo delle funzionalità radio

5. I *timer* sono ora solo di tipo assoluto. E' stato rimosso la possibilità di implementare timer relativi, perché concettualmente scorretto reimpostare il clock hardware interno con un valore differente di quello impostato in fase di boot

6. La struttura dei *buffer dei messaggi* è stata modificata, creando una modalità di accesso alle differenti modalità di comunicazione, come ad esempio radio e seriale, simile, così da favorire la semplicità di programmazione

2.3.4 Contiki

Contiki [15] è un sistema operativo multitasking progettato per i dispositivi embedded e per le vecchie architetture a 8 e 16 bit incluso il processore 6510 della Commodore 64. E'sviluppato per sistemi di rete con limitata capacità di memoria. La sua occupazione è di 2 KByte di memoria RAM e 40 di ROM. Contiki offre un kernel che permette di poter allocare dinamicamente i programmi. L'esecuzione dei processi è basata su thread molto leggeri e usa messaggi per la gestione degli eventi.

Questo sistema operativo può funzionare su microprocessori TI MSP430 e Atmel AVR. Opzionalmente Contiki offre una GUI grafica per la connessione con i terminali della rete.

Caratteristiche (alcune opzionali):

- Multitasking kernel
- Optional per-application pre-emptive multithreading
- Protothreads
- TCP/IP networking, including IPv6
- Windowing system and GUI
- Networked remote display using Virtual Network Computing
- Web browser
- Personal web server
- Simple telnet client
- Screensaver

2.4 Protocolli di rete per le Wsn

L'obiettivo primario di ogni nodo è quello di inviare i propri dati verso un centro di raccolta, all'interno della Wireless Sensor Network, che viene definito *Gateway*: quest'ultimo ha il compito di inviare tutti i dati pervenuti ad un sistema centrale, generalmente un server, che funge da database, tramite comunicazione wired (Ethernet, USB, LAN, etc.), oppure wireless (es. GPRS). Nelle Wireless Sensor Network più evolute il flusso dati può non limitarsi a quello monodirezionale descritto (dai nodi al Gateway), ma può essere omnidirezionale, ovvero possono essere trasmessi comandi da nodo a nodo, o dal server centrale (quindi dall'utente) ai nodi.

La possibilità di scambio dati tra vari nodi in una WSN è garantita dalla infrastruttura di comunicazione, che viene a crearsi tra i nodi, grazie ad un protocollo implementato sui nodi stessi: i protocolli di gestione della rete si dividono principalmente in due classi, quelli *flat* e quelli *gerarchici* (Figura 2.12).

I protocolli flat prevedono che tutti i nodi siano tra loro "allo stesso livello di importanza", eccetto uno che funge da *master* o coordinatore, che può o meno coordinare le trasmissioni degli altri (*slave*), ma comunque ha il compito di trasferire i dati della WSN verso il Gateway. La configurazione, che generalmente si viene a creare in questo tipo di rete, con pochi nodi, è detta "a stella", perché tutti i nodi comunicano col master in maniera diretta.

I protocolli gerarchici, invece, prevedono l'attribuzione di livelli diversi tra i singoli nodi: in base alle metriche scelte, ogni nodo, che riceva un segnale di *beacon* da un altro, diventa lo slave e l'altro il master; a sua volta il nodo slave, generando un segnale di beacon, può diventare il master per un altro nodo.

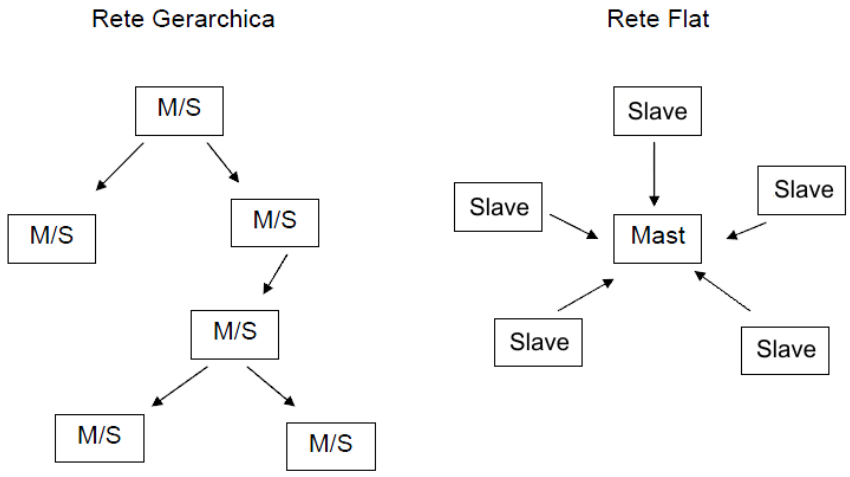


Figura 2.12: Topologie di reti wireless

2.4.1 La formula di Friis

Dati due punti A,B, a distanza R l'uno dall'altro, e data la trasmissione di un'onda elettromagnetica, che supponiamo irradiarsi in forma sferica, si dimostra che la potenza di trasmissione (PT) necessaria per raggiungere il punto B da A, è proporzionale alla potenza ricevuta (PR) e ad un fattore R, che rappresenta il raggio dell'onda sferica, elevato ad un coefficiente m, intero, pari al numero di trasmissioni intermedie tra A e B:

$$PT \propto (Rm * PR)$$

Quindi, minore è il raggio, minore sarà la potenza da trasmettere : se lo stesso tragitto A-B, invece di coprirlo con un'unica trasmissione, viene effettuato con un'insieme di trasmissioni a potenza più bassa lungo il tragitto stesso, si ottiene che la potenza totale necessaria in trasmissione è notevolmente inferiore (Figura 2.13).

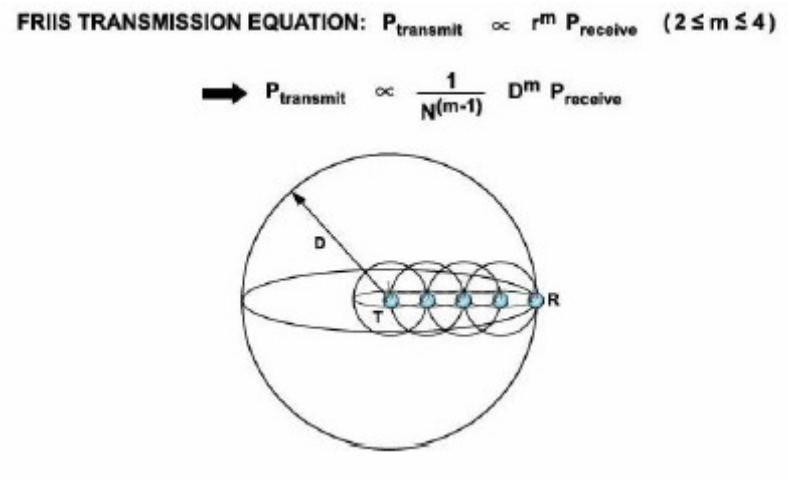


Figura 2.13: Formula di Friis

2.4.2 Protocolli di comunicazione: IEEE 802.15.4

Lo standard IEEE 802.15.4 [18] è pensato per comunicazioni wireless di basso costo, bassa velocità e basso consumo energetico. E adatto a reti W-PAN a basso bit rate costituite da dispositivi alimentati tramite batterie che non possono essere sostituite frequentemente, come, ad esempio, i nodi sensori. Le sue principali caratteristiche sono la capacità di coprire un raggio di azione di poche decine di metri offrendo fino ad un data-rate massimo di 250 Kbps. Offre una bassa potenza in trasmissione implicando un basso consumo energetico. Permette ai livelli superiori la presenza di livelli di rete che possono effettuare routing. Offre altresí la possibilità di gestire le ritrasmissioni dei dati tramite una modalità di Acknowledgment in caso di mancato ricevimento o di errore di trasmissione.

Livello fisico. I servizi offerti intervengono sull'accensione e sullo spegnimento del modulo radio, questa opzione permette al sistema un risparmio d'energia, effettua, inoltre, la scelta del canale di comunicazione

calcolando una stima sull'occupazione del canale ed analizza e calcola il rapporto segnale/rumore di un canale per scegliere il migliore diminuendo la probabilità di errori in trasmissione. Ovviamente modula e demodula i segnali in ricezione e in trasmissione.

Il livello fisico opera nelle frequenze ISM utilizzando tre possibili bande libere:

- 868-868.6 Mhz: banda utilizzata nella maggior parte dei paesi europei con un data-rate di 20kbps disponendo di un solo canale per la trasmissione;
- 902-928 Mhz : banda utilizzata nel continente oceanico e nel continente americano, offrendo un data-rate di 40Kbps e 10 canali disponibili;
- 2400-2483.5 : banda utilizzabile in quasi tutto il globo con un data-rate massimodi 250 Kbps e 16 canali a disposizione.

La modulazione del segnale più diffusa è il DSSS utilizzando tecniche BPSK, O-QPSK, anche se recentemente sono state introdotte nuove modulazioni in aggiunta al DSSS come PSSS.

Livello MAC. Il secondo livello (MAC) offre servizi quali la possibilità di creare una rete PAN, la trasmissione dei beacon e l'accesso al canale tramite il protocollo CSMA/CA. Questo livello supporta algoritmi di cifratura basata su AES-128 per la sicurezza dei dati, gestisce, inoltre, l'*handshake*, cioè gli Acknowledge per la ritrasmissione dei dati in caso di mancata o erronea ricezione. Calcola e verifica l'integrità della PDU. Può supportare reti fino ad un massimo di 65536 nodi poiché utilizza un sistema di indirizzamento fino a 16 bit.

2.4.3 Lo standard Bluetooth

Lo standard Bluetooth (IEEE 802.15.1) [19], offre un metodo economico e sicuro per scambiare informazioni tra dispositivi diversi attraverso una frequenza radio a corto raggio. La tecnologia Bluetooth opera anch'essa nella banda di frequenze tra 2,4 e 2,5 GHz (ISM), usando una modulazione FSK con un data-rate di 720Kbps. Le reti topologicamente più semplici che possono essere formate usando dispositivi Bluetooth vengono dette *piconet*. Esse sono composte al massimo da otto elementi. In ogni piconet si distingue un nodo, detto master, mentre gli altri sono detti slave. Il master funge da coordinatore e concorda la sincronizzazione. Lo slave funge da nodo passivo, accetta le condizioni dettate dal master. Un dispositivo può trovarsi in stadi intermedi, meno reattivi, ed è utilizzata la trasmissione adattiva per il risparmio delle batterie.

2.4.4 ZigBee

Basato sulle specifiche dello standard 802.15.4, la tecnologia ZigBee [20] implementa protocolli fino a livello applicativo. In Figura 2.14 viene proposto un parallelo con la classica architettura ISO/OSI. Viene definita a basso costo perché può essere utilizzata in svariate applicazioni e a basso consumo offrendo alte prestazioni sulla conservazione delle batterie. ZigBee Alliance è l'organismo che definisce i profili delle applicazioni accessibili gratuitamente.

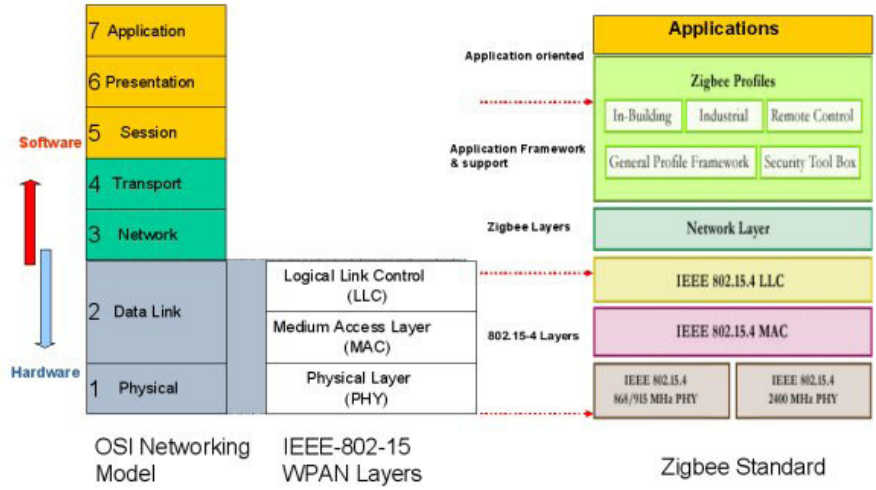


Figura 2.14: Stack protocollo ZigBee

2.5 Interrogazione di una rete di sensori

E' auspicabile una metodologia basata su *middleware* per l'interrogazione di una rete di sensori. L'obiettivo è poter utilizzare meccanismi di più alto livello rispetto a quelli forniti dai tool per la programmazione di sistema.

Un middleware deve essere in grado di dare una visione unitaria della rete di sensori, ovvero deve prescindere dal singolo sensore, deve fornire delle *api* di alto livello all'utente della rete (tipicamente esterno) e queste *api* devono astrarre la topologia e la composizione della rete. Un approccio di questo tipo permette di integrare facilmente una rete di sensori all'interno di altri sistemi informativi più complessi ed eventualmente di integrare le funzionalità di più reti di sensori tra di loro.

Tipicamente un middleware per reti di sensori è composto da due parti: una in esecuzione sui singoli sensori (sviluppata con le tecniche offerte dalla programmazione del software di sistema) e una in esecuzione sulla/sulle base station che offre l'interfaccia con la rete agli utilizzatori.

2.5.1 L'approccio di TinyDB

TinyDB [21] è un middleware per l'accesso ai dati misurati dai sensori di una rete astraendo l'intera rete che viene vista come un database. L'accesso avviene attraverso un linguaggio dichiarativo molto simile a SQL. L'interazione da parte dell'utilizzatore della rete avviene mediante un'interfaccia grafica o testuale.

Il concetto di query è lievemente diverso da quello dei DBMS tradizionali: qui la query viene iniettata nella rete e rimane attiva per un tempo specificato. Durante 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 invece che una sola volta.

Il sistema è sviluppato per sensori Motes sotto forma di applicazione TinyOS sviluppata in nesC.

Caratteristiche:

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

2. **Query multiple**. E' 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

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

TinyDB è composto da due macrosezioni: il software eseguito sui singoli sensori (sviluppato in nesC) e l'interfaccia client (sviluppata in Java).

Il tipico processo di funzionamento inizia quando il client invia una query al nodo collegato fisicamente ad esso (tipicamente un Mote collegato via cavo a un computer). La query a questo punto viene diffusa a tutti i sensori della rete costruendo una struttura di routing ad albero che ha il nodo collegato a un computer come radice. Ogni nodo provvede a rilevare i dati dai propri sensori, propagare le informazioni che arrivano dai nodi a valle ed eventualmente prima di propagare le informazioni aggregarle. Quando tutte le informazioni sono giunte al nodo radice la query termina e viene visualizzata o passata all'interfaccia Java.

Il software contenuto sul singolo sensore è logicamente composto da diversi componenti:

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

2. **Query processor**. Questo è il componente chiave durante l'esecuzione di una query, infatti esso 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

3. **Network topology manager**. Questo componente fornisce un'interfaccia per interagire con diversi componenti sottostanti per il routing, esso fornisce i comandi per inviare una query e inviare i risultati e gli eventi per ricevere query e risultati

Il progetto TinyDB è stato attualmente abbandonato.

Una delle problematiche fondamentali consiste nella mancanza di un controllo efficace sulla durata e la persistenza delle query, causata della tipologia di routing adottato e rischiando il potenziale sovraccarico dei nodi, soprattutto più vicini alla radice. Questo si traduce, in problematiche nel parametro fondamentale di una rete di sensori, ovvero la gestione energetica.

Il sistema sviluppato e descritto in questo lavoro di tesi prende ispirazione dall'approccio adottato da TinyDB. Si tratta di un sistema centralizzato, con livelli diversi di query (utente e nodo sensore). I dati sono raccolti in modo semicentralizzato in sottosistemi dislocati, ai quali è direttamente connessa una rete di sensori fisica, di tipologia a stella. Nessun dato è memorizzato sui nodi sensori, i quali hanno solo il compito di campionare e inviare i dati al gateway secondo le modalità impostate dalle query. Per una descrizione approfondita del sistema si rimanda alle successive sezioni 4 o 5.

Capitolo 3

Texas Instruments EZ430-RF2500

Il dispositivo utilizzato per sviluppare la rete di sensori wireless e il sistema basato su database per l'interrogazione della stessa è prodotto dalla Texas Instruments e si chiama EZ430-RF2500. Esso è costituito da una piccola scheda elettronica (*target board*) nella quale sono collegati i due integrati fondamentali della piattaforma: la CPU MSP430 e il Transceiver CC2500. L'EZ430-RF2500 può essere programmato tramite l'apposito adattatore USB (Figura 3.1), utilizzato inoltre per la comunicazione tramite il connettore UART (comunicazione di tipo seriale). L'alimentazione è fornita dall'adattatore stesso, prelevata direttamente dalla porta USB. L'EZ430-RF2500 nel caso in cui non sia connesso direttamente al Personal Computer, può essere alimentato tramite due batterie stilo tipo "AAA", tramite l'apposito holder (Figura 3.2).

Sulla target board sono collocati, oltre ai due integrati principali, due led di colore rosso e verde, collegati ad altrettante uscite digitali del microcontrollore. Sono inoltre disponibili ulteriori 18 pin per connettere altrettante porte di I/O analogiche e digitali. E' inoltre disponibile un pulsante a pressione per l'interazione con l'utente. Le funzionalità radio sono garantite dal transceiver grazie alla presenza di un antenna a 2.4 GHz integrata nel circuito stampato.

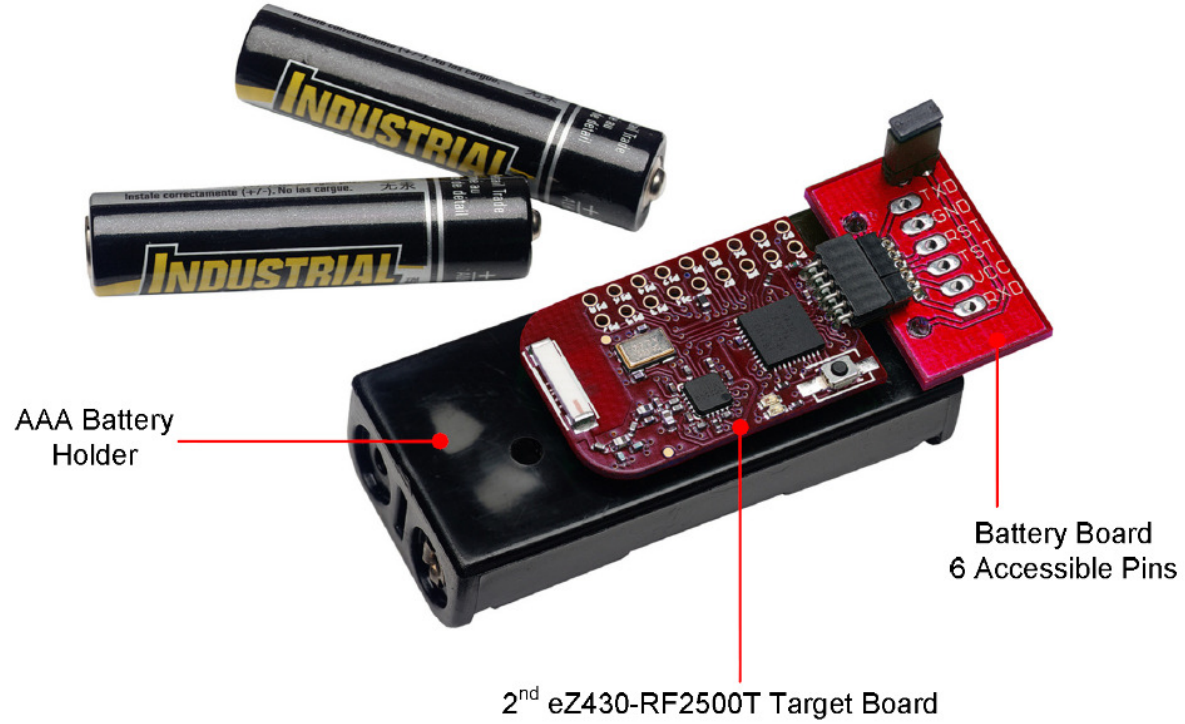


Figura 3.1: End Device EZ430-RF2500

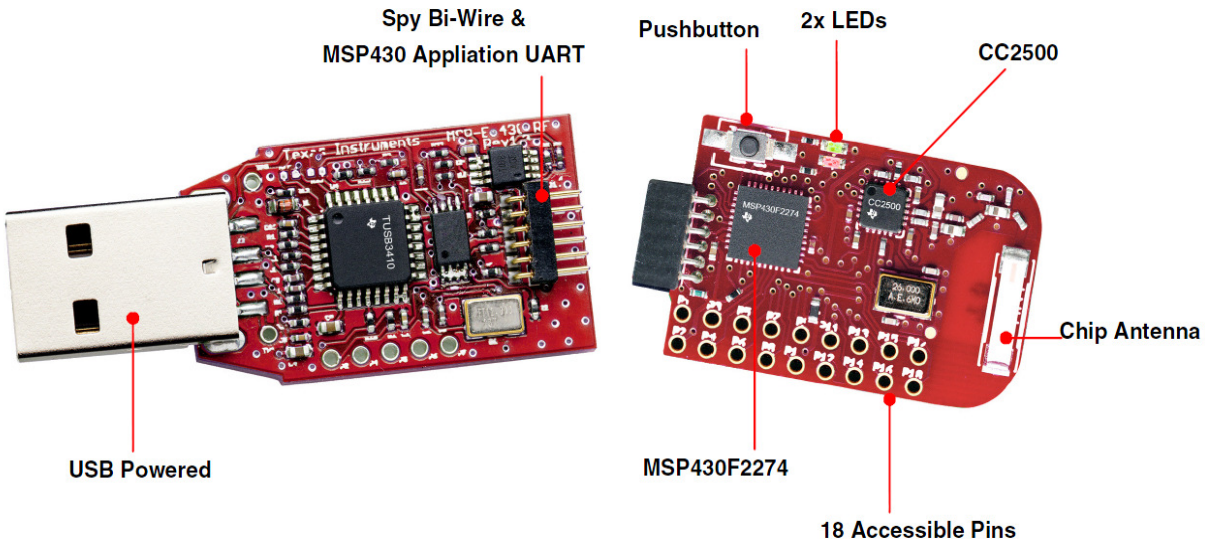


Figura 3.2: Access Point EZ430-RF2500

3.1 Il microcontrollore MSP430

Il microcontrollore MSP430 [22] incorpora una CPU a 16 bit RISC, interconnessa tramite un bus dati (MDB) e un bus indirizzi (MAB) a 16 bit con le memorie di sistema (RAM e Flash/ROM) e le periferiche di I/O (Figura 3.3). Una peculiarità di questo dispositivo è la disponibilità di diversi clock interni pilotanti differenti parti del dispositivo. In questo modo è possibile offrire fino a cinque differenti modalità di funzionamento a basso consumo (LPMx), che attivano/disattivano combinazioni diverse dei clock disponibili.

Il sistema di clock è stato specificatamente pensato per le applicazioni che richiedono l'alimentazione a batteria. Il clock ausiliario a bassa frequenza e consumo (ACLK) è generato direttamente da un oscillatore a cristallo da 32KHz. Questo clock con funzionamento in background è usato per gestire le funzionalità di wake-up del circuito quando il clock primario è disabilitato. Questo (MCLK) è generato internamente da un oscillatore digitale a 1MHz (DCO). Il clock primario è usato sia dalla CPU che dalle periferiche.

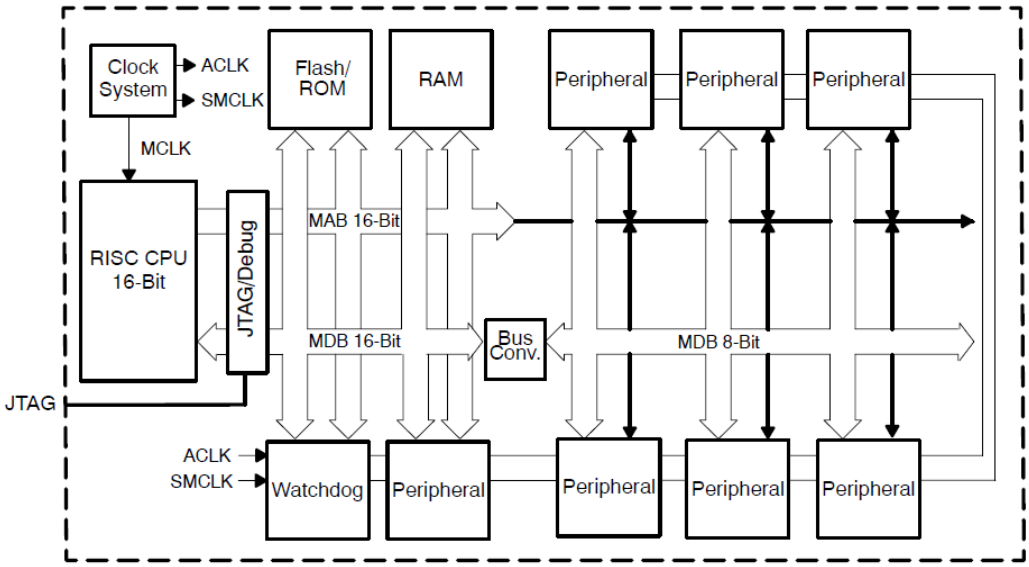


Figura 3.3: Architettura microcontrollore MSP430

Lo spazio di indirizzamento è comune per i registri di sistema, periferiche, RAM e Flash/ROM ed è pari a 128KB, per tutte le versioni del microcontrollore. La dimensione delle memorie di sistema varia in base al modello di microcontrollore, commercializzato in diverse versioni e caratteristiche.

L'EZ430-RF2500 monta la versione **MSP430F2274** [22] (Figura 3.4) che dispone di 32KB di memoria Flash programmabile, per ospitare il codice dei programmi e 1 KB di RAM, per l'esecuzione degli stessi.

Le caratteristiche del dispositivo sono le seguenti:

- Basso voltaggio di alimentazione da 1.8V a 3.6V
- Consumi estremamente ridotti: da 270 µA a 1MHz in Active Mode, fino a 0.7 µA in modalità StandBy
- Architettura RISC a 16bit
- 2 timer a 16bit (Timer_A, Timer_B)
- Interfaccia di comunicazione seriale (USCI UART)
- Convertitore Analogico / Digitale a 10 bit – 200 ksps
- 2 amplificatori operazionali configurabili
- 32 I/O pins port
- Gestore interno del bootstrap

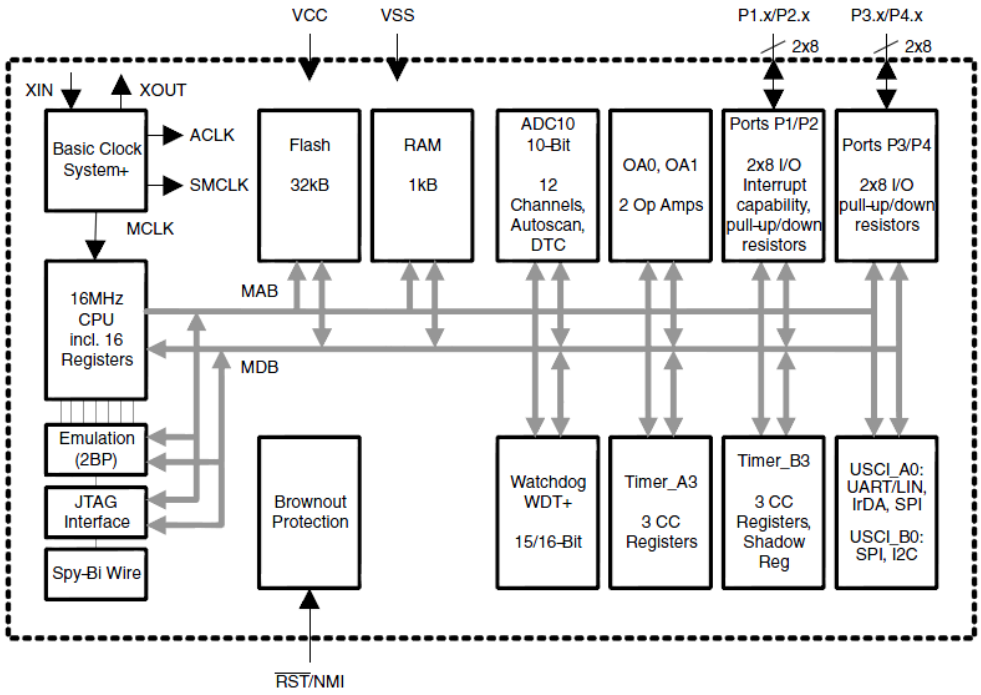


Figura 3.4: Architettura MSP430F2274

L'oscillatore controllabile digitale (DCO) permette il passaggio del dispositivo dalle modalità di StandBy alla modalità Active in circa 1 µs.

3.2 Il transceiver CC2500

Si tratta di un transceiver prodotto dalla Chipcon per la Texas Instruments [23] , a basso costo e a bassissimo consumo energetico. Funziona nel range di frequenza ISM (Industrial, Scientific e Medical) e per applicazione a piccolo raggio di azione (SRD). La banda di frequenza è 2400-2483.5 MHz. Supporta diverse modalità di modulazione del segnale radio e raggiunge una velocità di trasmissione massima di 500 kbps. Caratteristiche:

- Dimensioni ridotte: 4 x 4 mm, 20 pins di connessione.
- Alta sensibilità: -101 dBm a 10 kbps
- Basso consumo di corrente: 13.3 mA in RX a 250kbps
- Funzionamento a multi-canale
- Buffer in modalità FIFO da 64Byte, sia in ricezione che in trasmissione
- Informazioni RSSI, ovvero sulla potenza del segnale radio ricevuto
- Wake-on-radio, il dispositivo può attendere la ricezione di un frame in uno stato di sleep a basso consumo, passando nello stato attivo per l'effettiva ricezione del frame.

Le caratteristiche sopra descritte rendono il CC2500 un ottimo dispositivo per l'implementazione in un nodo sensore di una WSN.

In Figura 3.5 si mostra lo schema di interconnessione del CC2500 col microcontrollore MSP430 e in particolare il collegamento di alcuni interrupt che permettono al transceiver di "svegliare" il microcontrollore in caso di messaggi radio in arrivo.

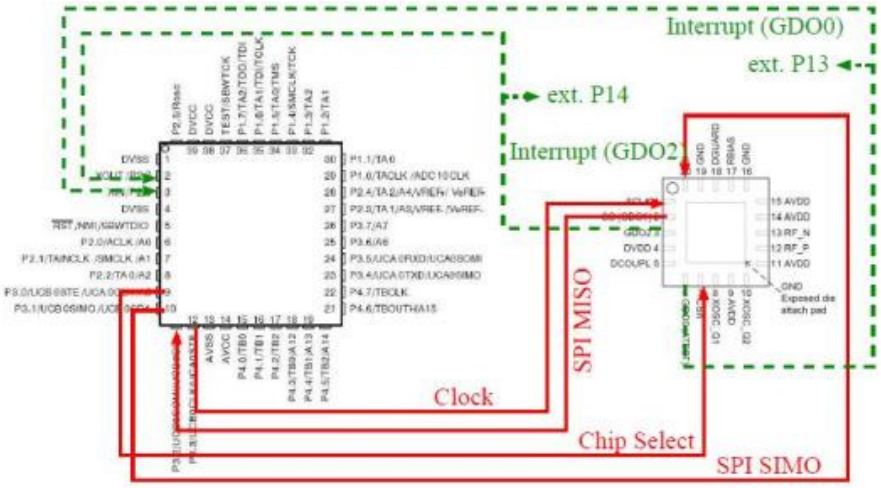


Figura 3.5: Schema connessione MSP430 – CC2500

3.3 Il protocollo di rete SimpliciTI

Il protocollo di rete a radiofrequenza SimpliciTI, sviluppato dalla stessa Texas Instruments per l'utilizzo abbinato alle famiglie di transceiver CC2500/CC1100. E' un protocollo proprietario ma liberamente utilizzabile, definito come *low power* e *low cost* , in quanto supporta dispositivi a risparmio energetico e i meccanismi per attivare e disattivare tali dispositivi. A basso costo poiché richiede poche risorse di sistema, tipicamente il protocollo utilizza meno di 4-8KB di Flash per la memorizzazione del codice e meno di 512-1024B di RAM per il funzionamento.

Il protocollo definisce diversi dispositivi logici della rete e differenti topologie per la comunicazione. SimpliciTI supporta la cifratura a chiave simmetrica dei dati. Disponibile in modo nativo se supportata dal dispositivo hardware, oppure implementabile via software.

Il protocollo fornisce un meccanismo automatico per il cambio di frequenza radio (secondo una tabella preconfigurata) per la comunicazione. Può essere utile nel caso in cui il canale dove si trasmette, risulti in un determinato momento disturbato, in questo modo la continuazione della comunicazione è garantita. Questo meccanismo è chiamato *Frequency Agility*.

3.3.1 Dispositivi logici

Il protocollo SimpliciTI definisce tre tipologie di dispositivi logici collegabili alla rete:

- *End Device (ED)*

- *Access Point (AP)*
- *Range Extender (RE)*

L'ED è la modalità standard di nodo connesso alla rete, generalmente alimentato a batterie. E' obbligatorio nel senso che ogni rete ne deve implementarne almeno uno. L'ED può essere attivo solo in trasmissione oppure in trasmissione/ricezione, inoltre può essere sempre attivo o dormiente, attivandosi solo per la comunicazione.

Il RE è un particolare tipo di nodo sempre attivo e generalmente alimentato direttamente. Il dispositivo trasmette unicamente i messaggi ricevuti. Svolge quindi funzionalità di ripetitore di segnale. In particolari configurazioni può svolgere funzionalità accessorie tipiche degli ED.

L'AP è un tipo di nodo sempre attivo e pensato per essere alimentato direttamente. Ogni rete permette al più un AP e non ne è obbligatoria la presenza. Tra le funzioni dell'AP vi sono: la gestione degli indirizzi di rete e la memorizzazione e inoltro dei messaggi agli ED dormienti. Un AP può implementare anche funzionalità tipiche degli ED (sensori e attuatori) o svolgere il compito del RE.

3.3.2 Topologie di rete supportate

SimpliciTI offre due differenti topologie di rete:

- *Rete a Stella*
- *Rete Peer-to-peer*

Nella rete a Stella gli ED, sia attivi che dormienti, costituiscono i nodi periferici comunicanti, eventualmente attraverso un RE, con il solo AP. L'AP rappresenta il *centro stella*, ovvero il gateway dell'intera rete wireless.

Nella configurazione Peer-to-peer invece gli ED sono sempre attivi e pronti per la ricezione. La trasmissione avviene quindi in modo diretto tra due ED.

SimpliciTI è un protocollo, come dice il nome stesso, incentrato sulla semplicità e pensato per architetture di rete con un numero limitato di nodi. Una lacuna, volutamente non implementata per mantenere le caratteristiche spiegate sopra e su cui si basa in protocollo stesso, è l'assenza di meccanismi di *routing esplicito*. Ogni nodo della rete può comunicare esclusivamente con i nodi direttamente connessi, poiché sono gli unici di cui è a conoscenza.

Alcuni esempi di architetture di rete Peer-to-peer:

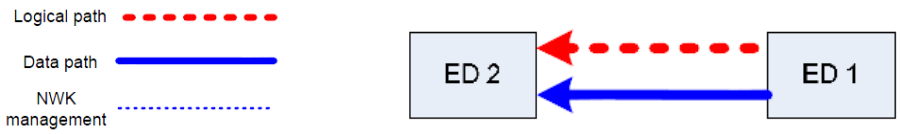


Figura 3.6: Peer to peer diretto

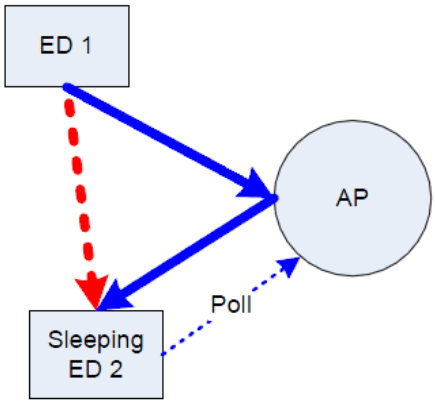


Figura 3.7: Peer to peer con AP

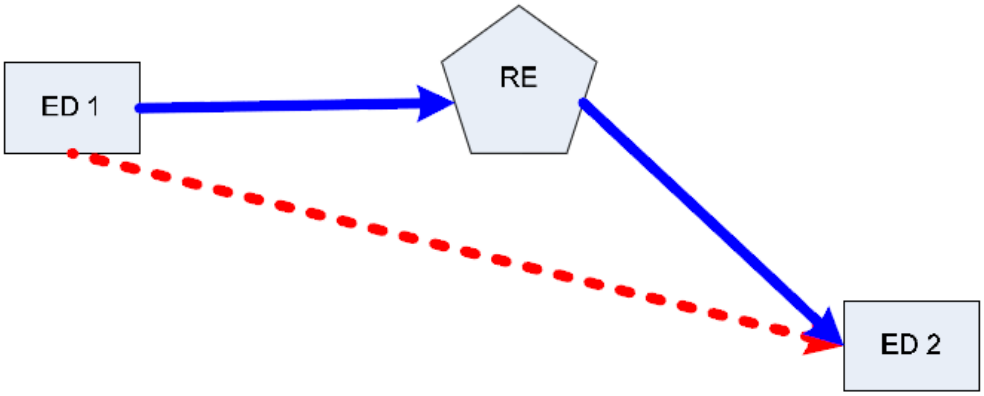


Figura 3.8: Peer to peer con AP e RE

Le Figure 3321, 3322, 3323 rappresentano rispettivamente tre differenti topologie di rete Peer-to-peer, in particolare:

- Peer-to-peer diretto
- Peer-to-peer attraverso un AP con modalità store and forward verso un ED dormiente
- Peer-to-peer attraverso un AP con modalità store and forward verso un ED dormiente con l'utilizzo di un RE.

3.3.3 Architettura del protocollo

SimpliciTI è concettualmente costituito da tre livelli software (Figura 3.9). Il livello più alto di applicazione è l'unico programmabile dall'utilizzatore.

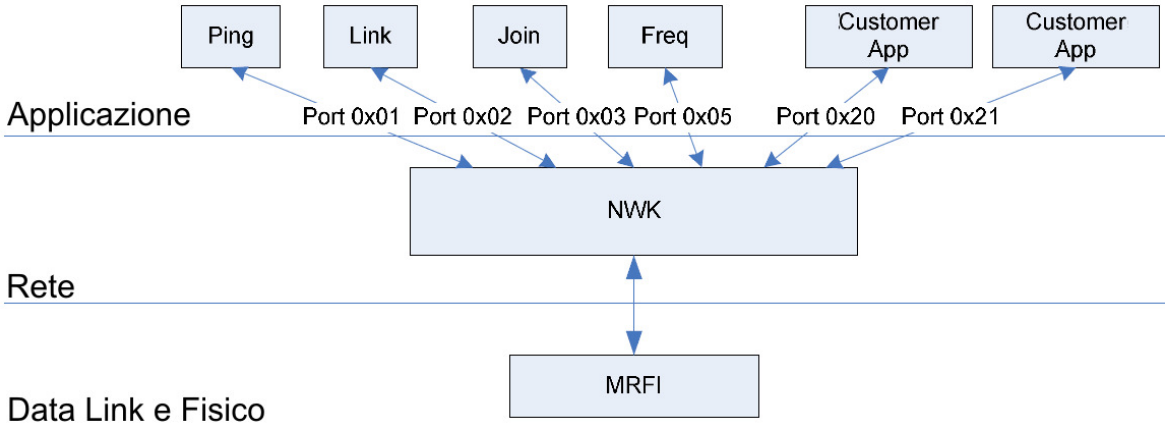


Figura 3.9: Livelli software protocollo SimpliciTI

Livello Applicazione. Fornisce alcune funzioni su determinate porte per gestire l'interazione con la rete. Il programmatore può sviluppare proprie applicazioni utilizzando le due porte disponibili. Il protocollo fornisce apposite API per svolgere all'interno dell'applicazione tutte le funzionalità messe a disposizione dal protocollo.

Livello di Rete. Questo livello si occupa della gestione di svariati parametri per la comunicazione. Alcuni di essi posso essere modificati a livello applicazione tramite l'interfaccia *ioctl()*. Questi riguardano: la frequenza base per la trasmissione, il supporto al cambio di canale (*frequency agility*), tipo di modulazione del segnale e data rate della trasmissione, cifratura dei messaggi e indirizzi dei nodi.

Livello Fisico. E' gestito dall'interfaccia MRFI (Minimal RF Interface), si occupa del collegamento tra il livello di rete e l'hardware a disposizione. L'MRFI è sviluppato per funzionare su diversi dispositivi, implementanti differenti modalità di trasmissione radio a livello data link e fisico.

3.3.4 L'interfaccia di programmazione (API)

La programmazione di un'applicazione di rete risulta piuttosto semplice, in virtù delle poche funzioni implementate dall'API del protocollo:

Inizializzazione: *smplStatus_t SMPL_Init(void);*

Collegamento (bidirezionale): *smplStatus_t SMPL_Link(linkID_t *linkID);*
*smplStatus_t SMPL_LinkListen(linkID_t *linkID);*

Comunicazione: *smplStatus_t SMPL_Send(lid, *msg, len);*
*smplStatus_t SMPL_Receive(lid, *msg, *len);*

Configurazione: *smplStatus_t SMPL_Ioctl(object, action, *val);*

3.3.5 Struttura del frame di trasmissione

Ogni frame trasmesso col protocollo SimpliciTI è suddivisibili in tre porzioni, in base al livello dello *stack* del protocollo che lo genera. Vi è una piccola differenza sulla struttura del frame in base all'utilizzo o meno di cifratura della trasmissione (Figura 3.10 e 3.11).

Il significato di ogni campo del frame è specificato in Tabella 3352.

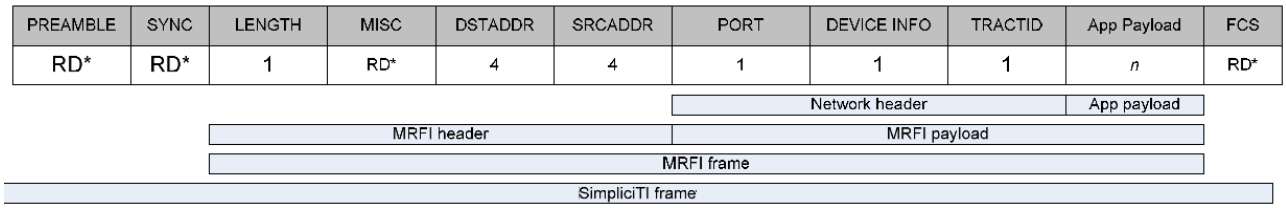
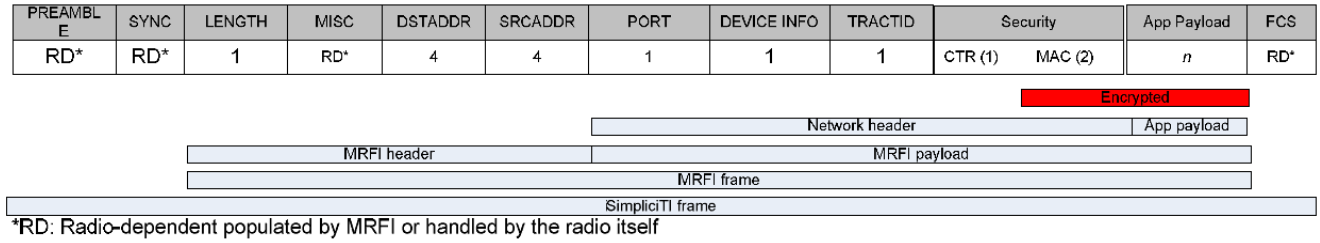


Figura 3.10: Frame protocollo SimpliciTI



*RD: Radio-dependent populated by MRFI or handled by the radio itself

Figura 3.11: Frame protocollo SimpliciTI con cifratura payload

Field	Definition	Comments
PREAMBLE	Radio synchronization	Inserted by radio HW
SYNC	Radio synchronization	Inserted by radio HW.
LENGTH	Length of remaining packet in bytes	Inserted by FW on Tx. Partially filterable on Rx.
MISC	Miscellaneous frame fields	Differ for different radios. May be absent.
DSTADDR	Destination address	Inserted by FW. Filterable depending on radio.
SRCADDR	Source address	Inserted by FW.
PORT	Forwarded frame (7), Encryption context (6) Application port number (5-0).	Inserted by FW. Port namespace reserves 0x20-0x3F for customer applications and 0-1F for `NWK` management.
DEVICE INFO	Sender/receiver and platform capabilities	Inserted by FW. Details below.
TRACTID	Transaction id	Inserted by FW. Discipline depends on context. Need not be sequential.
APP PAYLOAD	Application data	$0 \leq n \leq 50$ for non-802.15.4 radios; $0 \leq n \leq 111$ for 802.15.4 radios
FCS	Frame Check Sequence	Usually a CRC appended by the radio hardware.

Tabella 3.1: Campi del frame del protocollo SimpliciTI

Capitolo 4

Sistema di interazione con WSN basato su query

Nel seguente capitolo si tratterà l'architettura e il funzionamento del modello di sistema sviluppato per interrogare la rete di sensori. L'idea di base è poter richiedere informazioni o inviare istruzioni ai sensori con un modello generico basato su query. La query risulta essere l'elemento fondamentale che interfaccia l'utente con il sistema. Essa ha una struttura SQL-like, ovvero basata su una sintassi simile allo standard SQL. Il vantaggio nell'utilizzo di un modello di interrogazione generica, indipendente alla struttura fisica della rete sottostante, rende il sistema potenzialmente in grado di gestire flussi di dati provenienti da sorgenti eterogenee e quindi totalmente svincolato dalle piattaforme hardware presenti. Scopo primario è l'interazione dell'utente al più alto livello possibile di astrazione.

Il sistema è modulare e costituito da sottosistemi dislocati e distribuiti, comunicanti con un sistema centralizzato. Il sistema centralizzato, chiamato Query Manager (QM), gestisce completamente l'interazione con l'utente, l'acquisizione di richieste e la visualizzazione dei risultati. I sottosistemi distribuiti, chiamati Host Wrapper (HW), sono l'interfaccia di comunicazione fisica con la (o le) diverse WSN presenti. Essi accettano le configurazioni inviate dal QM interagendo con la propria WSN connessa. Hanno inoltre il compito di recupero, aggregazione e memorizzazione dei dati rilevati dai sensori, effettuando controlli sulla conformità degli stessi alle specifiche imposte. Vi sono meccanismi di rilevazione e comunicazione al sistema centrale in caso di errori sui dati raccolti.

Il sistema si basa sull'utilizzo di un Data Base Management System (DBMS) Relazionale sia per la memorizzazione delle richieste e query degli utenti, che per la memorizzazione dei dati rilevati. Sia Query Manager che Host Wrapper ospitano il proprio DBMS, visibili da entrambi i sistemi.

In Figura 4.1 è mostrato lo schema a blocchi del sistema nel suo complesso.

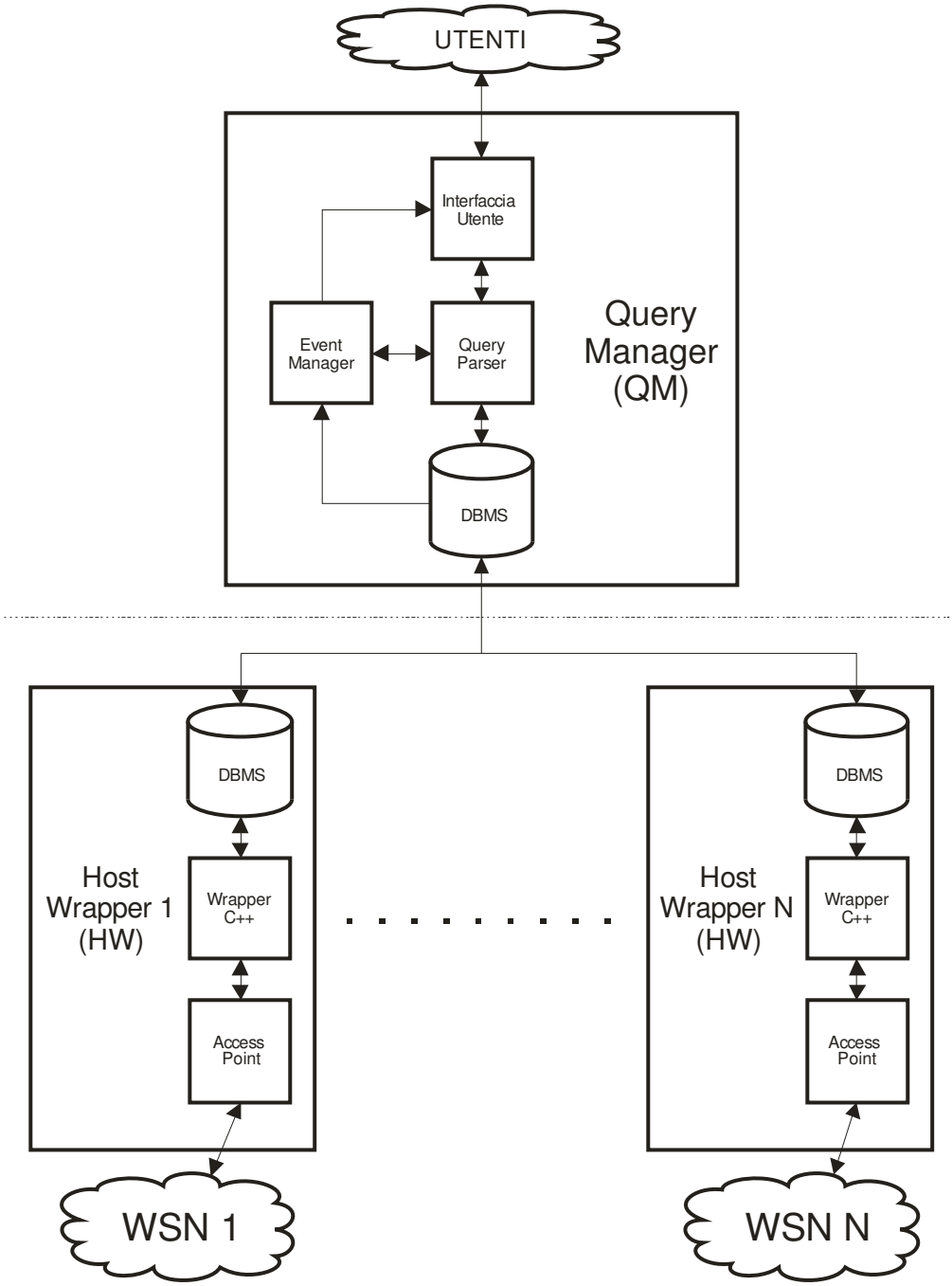


Figura 4.1: Sistema complessivo

4.1 Query Manager: Interrogazione della rete da parte di un utente

Gli utenti possono interagire col sistema, tramite un'apposita interfaccia (*Interfaccia Utente*), grafica o testuale, nella quale inserire le query che andranno in esecuzione sulla rete. Ogni query avrà una struttura standard che comprenderà il tipo di interrogazione richiesta, la frequenza di rilevamento dei dati, la modalità

di restituzione e visualizzazione degli stessi. Potrà essere specificata una durata di esecuzione della query stessa. L'utente potrà in ogni modo intervenire sulle query attive, modificandole o rimuovendole dal sistema. L'interfaccia utente avrà inoltre il compito di restituire i risultati delle query immesse dall'utente, nella modalità di visualizzazione e aggregazione dei dati preferita (Figura 4.2).

Figura 4.2: Esempio interazione utente

All'immissione di una query da parte dell'utente nello pseudo linguaggio SQL, essa viene memorizzata e passata dal modulo di interfaccia al modulo *Query Parser*. Quest'ultimo è la parte principale del QM, poiché svolge le operazioni più importanti. Innanzitutto ha il compito di tradurre, ed eventualmente correggere, la query immessa dall'utente. Essa può essere piuttosto complessa e richiedere informazioni aggregate da più reti di sensori, collegate fisicamente ad HW differenti. Il Query Parser (Figura 4.3) tramite il modulo *Analizer Module* controlla la query immessa, verifica nel database la corrispondenza effettiva con le reti wireless collegate, elabora la query passandola ad un secondo modulo, il *Composer Module.* Esso conoscendo la struttura fisica delle WSN, deve tradurre la query utente complessa in più query semplici (Query Wrapper), inoltrandole agli opportuni HW per l'esecuzione.

Si riporta un esempio: Monitoraggio della temperatura di un edificio. Ogni stanza è coperta nella varie zone da una rete con più sensori, collegati allo stesso HW. L'edificio nel suo complesso risulta quindi monitorato da *n* wsn distinte, pari al numero di stanze, ognuna collegata al proprio HW. L'utente può inoltrare una query per monitorare la temperatura di una singola stanza, oppure richiedere una temperatura aggregata (come la media) di tutto l'edificio. La query inviata dall'utente al sistema potrebbe essere la seguente:

```
SELECT AVG(temp) AS query_name FROM NET edificio EVERY 10 IN sec
```

Una volta inoltrata al Query Parser, sarà da esso tradotta in *n* Query Wrapper ognuna inviata al proprio Host Wrapper. Sempre il Query Parser avrà il compito di recuperare i dati generati nei vari HW secondo la specifica temporale di 10 secondi, aggregandoli e restituendoli all'interfaccia utente per la visualizzazione.

Di questo compito è incaricato il *Data Module*. La struttura della query eseguita sull'Host Wrapper è la seguente:

```
INSERT INTO CONFIGURATION (idq, iddev, freq) VALUES (x,y,10)
```

Dove Idq è l'identificativo univoco assegnato alla query utente (associato al query_name inserito dall'utente), Iddev è l'identificativo del sensore interessato e Freq è la frequenza di campionamento del dato. In questo caso saranno inviate *m* query distinte, pari al numero (m) di sensori presenti nella stanza da monitorare.

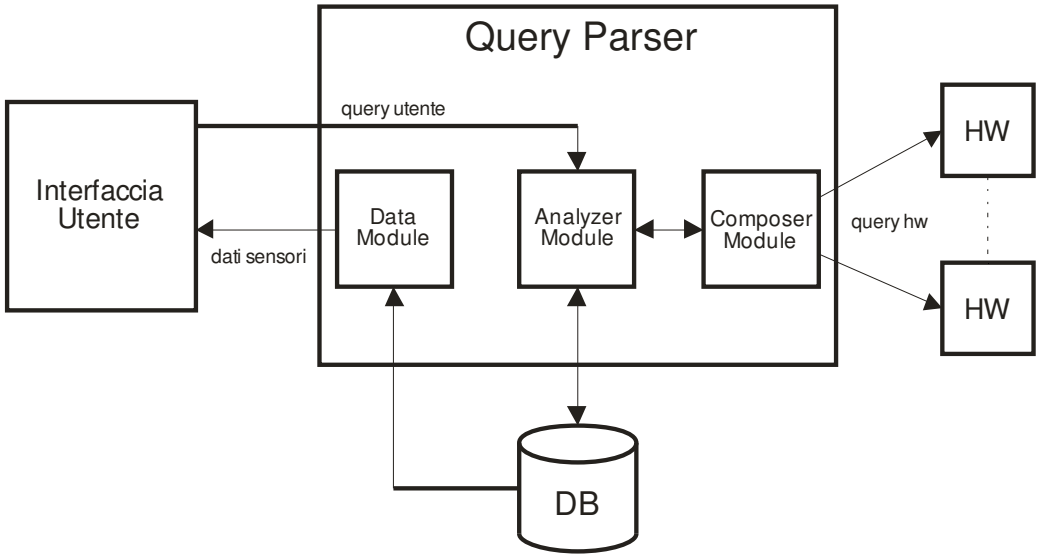


Figura 4.3: Query Parser

4.1.1 Le Query Utente

Nell'esempio precedente si è proposta la tipologia di query SELECT, utilizzabile per la sola interrogazione della rete di sensori per il recupero periodico dei dati. La struttura generica della query SELECT è la seguente:

```
SELECT data [AGGREG(data)] AS query_name FROM device_addr [NET net_name]
[EVERY timevalue IN (SEC|MIN|HOUR|DAY|WEEK|MONTH|YEAR)]
```

Non si tratta però dell'unica tipologia di query inviabile al sistema. Vi sono altre cinque tipologie di query: STORE, ALTER, DROP, START e STOP.

La più importante tra queste è la query STORE. Tramite questo comando si immette e si memorizza nel sistema una query persistente che oltre a specificare la frequenza di recupero dei dati, precisa opportuni

parametri da verificare sui dati acquisiti. Per il sistema in esame la query STORE avrà una struttura come la seguente:

```
STORE data [AGGREG(data)] AS query_name FROM device_addr [NET net_name]
AT timevalue TIMES IN (SEC|MIN|HOUR|DAY|WEEK|MONTH|YEAR)
EVENT AT ([MAX maxvalue] [MIN minvalue] [MAXDELTAP maxdeltapvalue]
[MINDELTAP mindeltapvalue] [MAXDELTAN maxdeltanvalue] [MINDELTAN
mindeltanvalue])
```

Questo tipologia di query verrà interpretata del Query Parser e genererà diverse Query Wrapper per gli opportuni Host Wrapper. I valori specificati dopo EVENT produrranno l'inserimento delle condizioni di controllo che gli HW dovranno monitorare, segnalando il mancato rispetto dei vincoli imposti.

Le condizioni di controllo sui dati sono inserite negli HW, con l'apposita query (vedi 3.1.2).

Ogni Host Wrapper dovrà controllare i dati in ricezione dalla WSN, verificando che rispettino le condizioni specificate. In caso contrario l'HW segnalerà al QM la situazione di anomalia, inviando le differenze dai parametri stabiliti. Sarà compito del modulo Event Manager, gestire la situazione di errore, attuando le opportune operazioni di controllo.

Le tipologie di query ALTER e DROP sono usate rispettivamente per la modifica di una query memorizzata (SELECT o STORE) nel sistema, specificandone i nuovi parametri e per l'eliminazione di una query dal sistema. La struttura di ALTER applicata a query di tipo STORE, è la seguente:

```
ALTER STORE query_name SET data [AGGREG(data)] EVERY timevalue TIMES IN
(SEC|MIN|HOUR|DAY|WEEK|MONTH|YEAR)
EVENT AT ([MAX maxvalue] [MIN minvalue] [MAXDELTAP maxdeltapvalue]
[MINDELTAP mindeltapvalue] [MAXDELTAN maxdeltanvalue] [MINDELTAN
mindeltanvalue])
```

Oppure in caso di query di tipo SELECT:

```
ALTER SELECT query_name SET data [AGGREG(data)] [EVERY timevalue IN
(SEC|MIN|HOUR|DAY|WEEK|MONTH|YEAR)]
```

La struttura di DROP è la stessa per entrambi i tipi di query:

```
DROP query_name
```

Le ultime due tipologie START e STOP, permettono di attivare e disattivare l'esecuzione di una query senza rimuoverla fisicamente dal sistema. Possono essere usate per fermare momentaneamente determinati rilevamenti, che si vorranno riprendere in breve tempo. La struttura è la medesima per entrambe:

```
STOP query_name

START query_name
```

4.1.2 Le Query dell'Host Wrapper nel sistema in esame

Le operazioni effettuate dal Composer Module, per istruire i vari Host Wrapper sul recupero e l'analisi dei dati della Wsn collegata, consistono nell'inserimento di determinate tuple in alcune tabelle nel database dell'HW.

Il primo inserimento riguarda la tabella CONFIGURATION, che contiene tutte le query sottese a nodi della propria WSN. Il secondo inserimento si ha nella tabella CONDITIONS, che ospita i vincoli di controllo sui dati per ogni query attiva. L'ultimo inserimento avviene nella tabella CHANGECONFIG, che è la tabella "operativa", ovvero con la quale si ordinano all'HW le operazioni da eseguire sulle query (Inserimento, Cancellazione, Disattivazione, Attivazione).

Per una trattazione più esauriente sul struttura del database dell'Host Wrapper nel sistema esaminato, si rimanda alla Sezione 5.2.

La struttura delle tre diverse query eseguite sull'HW, del sistema in esame è la seguente:

```
INSERT INTO CONFIGURATION (idq, iddev, freq) VALUES (idqvalue,iddevvalue, freqvalue)

INSERT INTO CONDITIONS (idq, max, min, pdh, pdl, ndh, ndl) VALUES (idqvalue, (maxvalue | null), (minvalue | null), (maxdeltapvalue | null), (mindeltapvalue | null), (maxdeltanvalue | null), (mindeltanvalue | null))

INSERT INTO CHANGECONFIG (idq, ins, del, act, disact) VALUES (idqvalues, (1|0), (1|0), (1|0), (1|0))
```

4.2 Query Manager: gestione monitoring dei dati, rispetto dei vincoli stabiliti

Il modulo adibito al controllo delle condizioni e i dei vincoli sulle query immesse nel sistema è l'Event Manager. Esso agisce sulle query di tipo STORE, per la verifica sul rispetto dei vincoli. In particolare viene verificato, tramite l'*Event Trigger Module* (Figura 4.4), l'invio da parte degli Host Wrapper di segnalazioni di errori, scansionando periodicamente un'apposita tabella dati (DIFFERENCE), scrivibile da ogni HW. Gli HW segnalano sulla suddetta tabella, il valore di scostamento dai parametri specificati per la determinata Idq attiva. Al rilevamento del valore differenza, l'Event Manager, tramite il *Decision Module*, attua opportune politiche di gestione dell'errore, in base alle specifiche dell'utente.

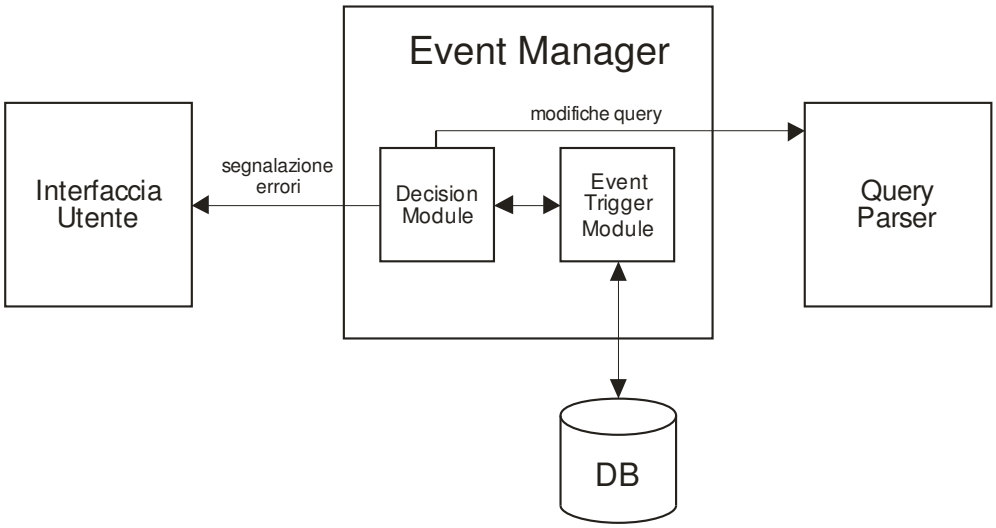


Figura 4.4: Event Manager

Le politiche di gestione degli errori possono essere svariate. Vi sono specifiche per la *segnalazione diretta all'utente* del parametro fuori vincolo oppure per la *modifica dei parametri della query* o ulteriormente per l'*attivazione di un'altra query* con parametri più restrittivi.

Si riporta un esempio con l'ultima situazione.

Vengono memorizzate due query di tipo STORE, la prima (Query1) attiva e la seconda (Query2) disattivata. La Query1 specifica un periodo di campionamento della temperatura della WSN di una stanza di un edificio ogni 30 secondi e una soglia di temperatura massima di 25 C°. La Query2 setta invece un periodo di rilevazione di 10 secondi e una soglia massima di 30 C°. L'Event Manager può essere istruito per disattivare, in caso di mancato rispetto delle condizioni di Query1, la prima query e rendere attiva la seconda. Nel caso di superamento anche della soglia impostata da Query2 l'Event Manager deve segnalare la situazione all'utente. I vantaggi di una configurazione siffatta sono notevoli: monitorare un parametro ambiente a una

frequenza bassa fa risparmiare batteria ai nodi, aumentandone la durata. Se la temperatura raggiunge il primo valore di attenzione, vi è allora la necessità di aumentare il periodo di rilevazione per avere il sistema più "rapido" nel verificare il superamento della seconda soglia, ovvero quella che merita la segnalazione all'utente. L'Event Manager può essere configurato per ritornare alla situazione iniziale, solo Query1 attiva, nel caso in cui la temperatura ritorni sotto la prima soglia.

4.3 Host Wrapper: recupero, analisi e inoltro dei dati acquisiti

Il sistema Host Wrapper è costituito da un calcolatore connesso tramite una rete LAN o WAN, con il Query Manager. La comunicazione dei dati, parametri ed eventi avviene tramite il DBMS MySQL. In particolare per la connessione bidirezionale tra i due sistemi è richiesta l'apertura verso "l'esterno" della sola porta TCP 3306, utilizzata come default dal DBMS MySQL.

L'Host Wrapper svolge principalmente due funzioni: la prima riguarda l'esecuzione di un software (*Wrapper C++*) incaricato di gestire la comunicazione bidirezionale con il database mentre la seconda è quella di ospitare, collegato a un porta USB, *l'Access Point* (AP) della WSN, ovvero il gateway, o centro stella, della rete wireless. E' verso l'AP che gli *End Device* (ED), ovvero i veri sensori della rete, trasmettono e ricevono i dati rilevati e di configurazione tramite la comunicazione radio.

Lo sviluppo del sistema Host Wrapper e dei software per i nodi della WSN sono stati lo scopo primario di questo lavoro di tesi e saranno analizzati in dettaglio nelle successive sezioni.

Capitolo 5

Architettura e software Host Wrapper

Il seguente capitolo spiegherà in dettaglio l'architettura e il funzionamento dell'Host Wrapper, i componenti ad esso collegati, il software in esecuzione e i meccanismi per la comunicazione, verso la rete di sensori e verso i database MySQL, locale e remoto.

5.1 L'architettura

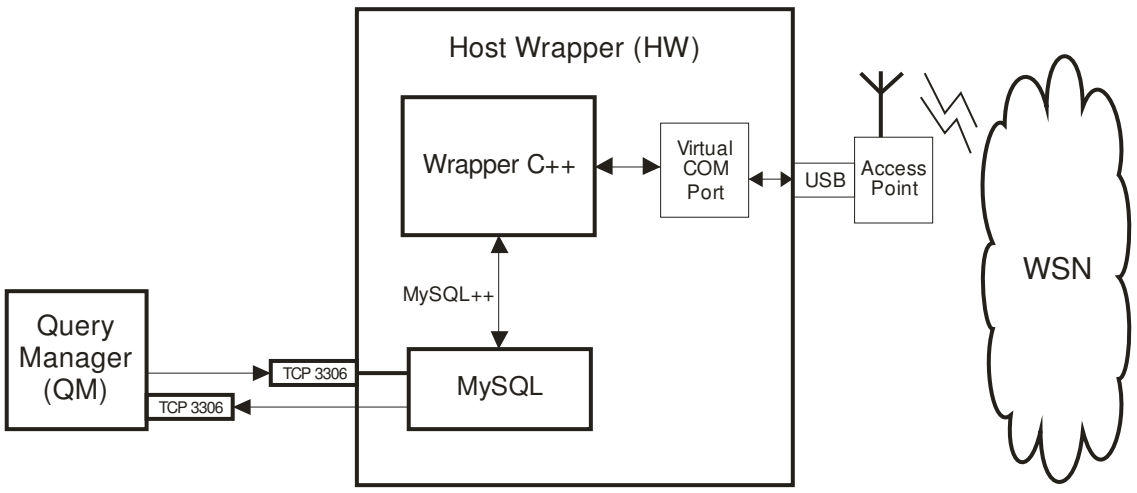


Figura 5.1: Architettura Host Wrapper

La Figura 5.1 mostra lo schema a blocchi dell'Host Wrapper costituito fisicamente da un calcolatore e da un EZ430-RF2500 collegato tramite un adattatore USB. La connessione è bidirezionale utilizzando una porta seriale "virtuale" (COM), creata dal driver software del dispositivo TUSB3410 "EZ430-RF USB Debugging Interface". Quest'ultimo permette, tramite la connessione USB al calcolatore, l'interfacciamento e

l'alimentazione di un sensore EZ430-RF2500, utilizzando l'apposito connettore a 6 pin**.** L'EZ430-RF2500 collegato all'interfaccia Usb, sarà di seguito chiamato Access Point (AP), poiché eseguirà un software apposito, differente da quello in esecuzione sugli EZ430-RF2500 utilizzati come sensori distribuiti nell'ambiente. Essi saranno di seguito chiamati End Device (ED).

La comunicazione dell'AP, tramite porta COM, avviene verso un il software definito "Wrapper C++". Esso è il fulcro del sistema Host Wrapper, in quanto gestisce la comunicazione da e verso la WSN, la memorizzazione e il recupero dei parametri dal DBMS locale e la comunicazione con il DBMS remoto.

La comunicazione col database remoto in funzione sul Query Manager (QM), avviene tramite protocollo TCP/IP e connessione alla porta TCP 3306, utilizzata dal DBMS MySQL. Utilizzando la sola connessione di MySQL, si garantisce un livello di flessibilità elevato, poiché l'architettura su cui è in funzione il QM e il database remoto non pregiudica il funzionamento del sistema, non necessitando di alcuna riconfigurazione della connessione. Nello stesso modo, la comunicazione del QM verso i vari Host Wrapper, avviene sempre tramite protocollo TCP/IP, verso il DBMS MySQL.

Nella parte seguente saranno considerati e analizzati i singoli componenti dell'Host Wrapper

5.2 MySQL e lo schema relazionale

La scelta del Data Base Management System (DBMS) da utilizzare è ricaduta sul prodotto di Sun Microsystem: MySQL [26]. La versione utilizzata nel sistema sviluppato è la 5.1.

MySQL è un DBMS relazionale, composto da un client con interfaccia a caratteri e un server, entrambi disponibili sia per sistemi Unix come GNU/Linux che per Windows, anche se prevale un suo utilizzo in ambito Unix. Dal 1996 supporta la maggior parte della sintassi SQL e si prevede in futuro il pieno rispetto dello standard ANSI. Possiede delle interfacce per diversi linguaggi, compreso un driver ODBC, due driver Java e un driver per Mono e .NET.

Il codice di MySQL viene sviluppato fin dal 1979 dalla ditta TcX ataconsult, adesso MySQL AB, ma è solo dal 1996 che viene distribuita una versione che supporta SQL, prendendo spunto da un altro prodotto: Msql. Il codice di MySQL è di proprietà della omonima società, viene però distribuito con la licenza GNU GPL oltre che con una licenza commerciale. Fino alla versione 4.0, una buona parte del codice del client era licenziato con la GNU LGPL e poteva dunque essere utilizzato per applicazioni commerciali. Dalla versione 4.1 in poi, anche il codice dei client è distribuito sotto GNU GPL. Esiste peraltro una clausola estensiva che consente l'utilizzo di MySQL con una vasta gamma di licenze libere.

Nel sistema sviluppato il DBMS svolge un ruolo fondamentale. Esso è utilizzato, oltre che per la naturale memorizzazione dei dati provenienti dalla WSN, anche per la memorizzazione delle query attive e delle condizioni di controllo che esse definiscono. Inoltre, sempre tramite database, è strutturata la comunicazione tra i due sistemi dislocati: il Query Manager e l'Host Wrapper. L'invio dei dati da e verso i due sottosistemi avviene tramite apposite tabelle. La comunicazione avviene così sempre in modo "indiretto", consentendo scalabilità a entrambi i sottosistemi. Si delega direttamente al DBMS, tramite le apposite API, l'instaurazione e la gestione della comunicazione e di eventuali connessioni concorrenti.

In dettaglio si mostra in Figura 5.2, lo schema relazionale del database installato localmente sull'Host Wrapper e in Figura 5.3 la tabella del database remoto sul server del Query Manager, interessata alla scrittura da parte del wrapper.

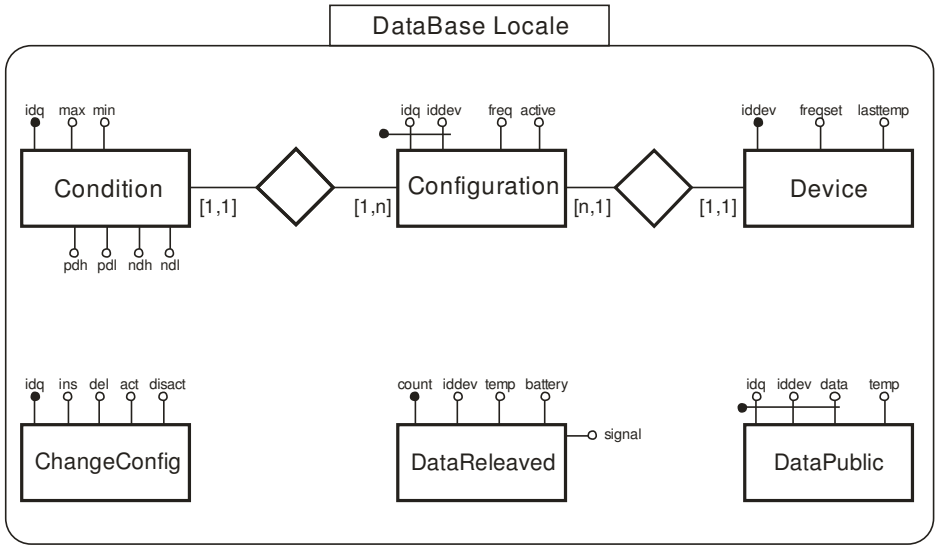


Figura 5.2: Schema database HW

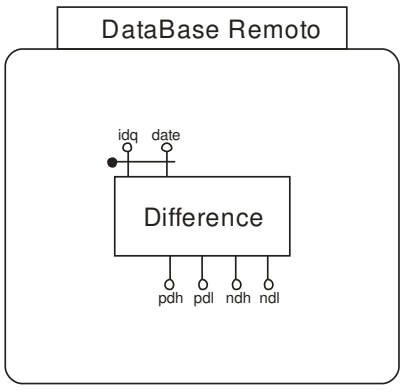


Figura 5.3: Schema database QM

5.2.1 Il flusso dei dati

Nelle seguenti figure sono rappresentati (Figura 5.4), le azioni che il Query Manager esegue sui database collegati, mentre in Figura 5.5, le azione compiute dall'Host Wrapper. Le operazioni eseguite saranno trattate in dettaglio nella Sezione 5.3.

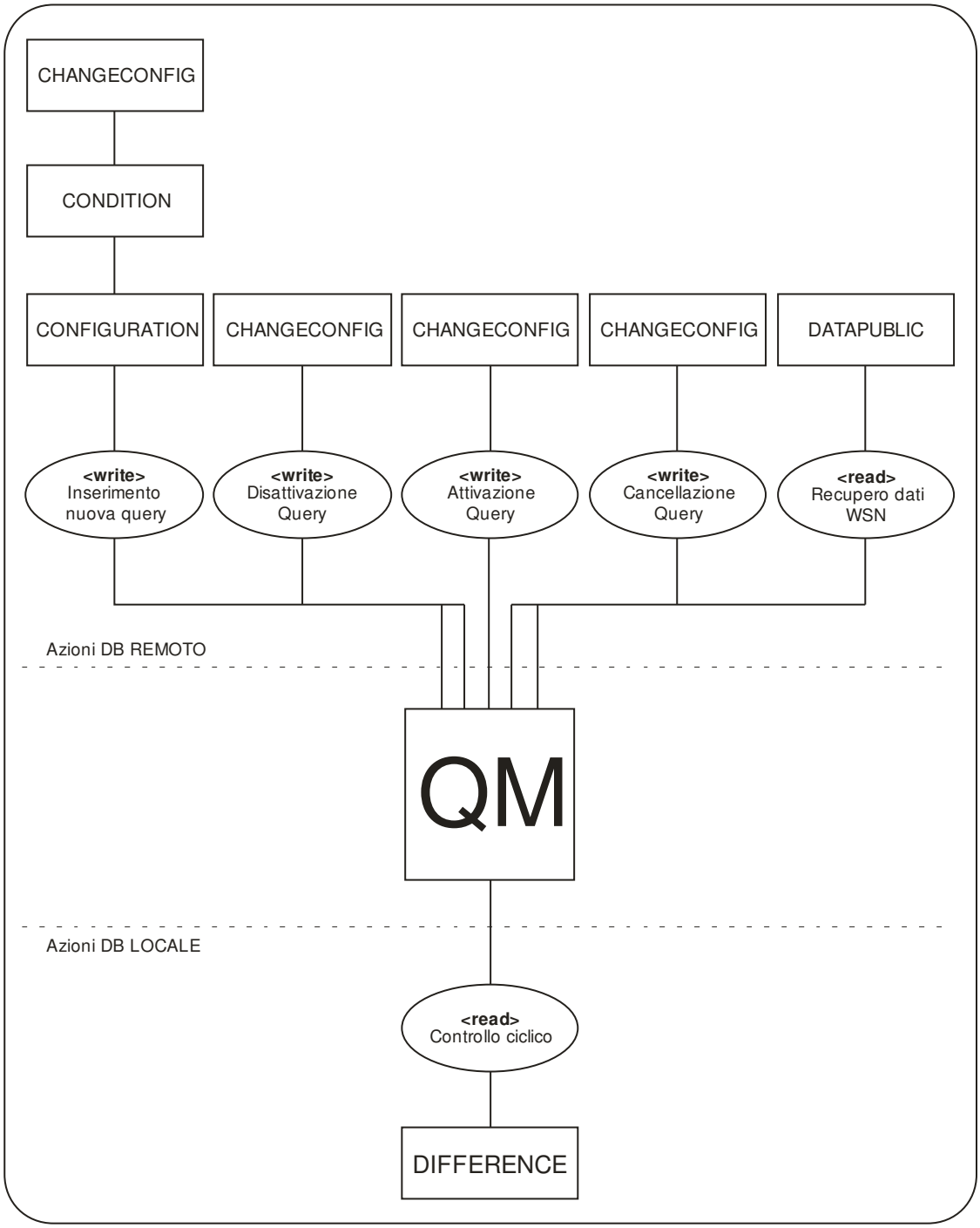


Figura 5.4: Flusso dati QM

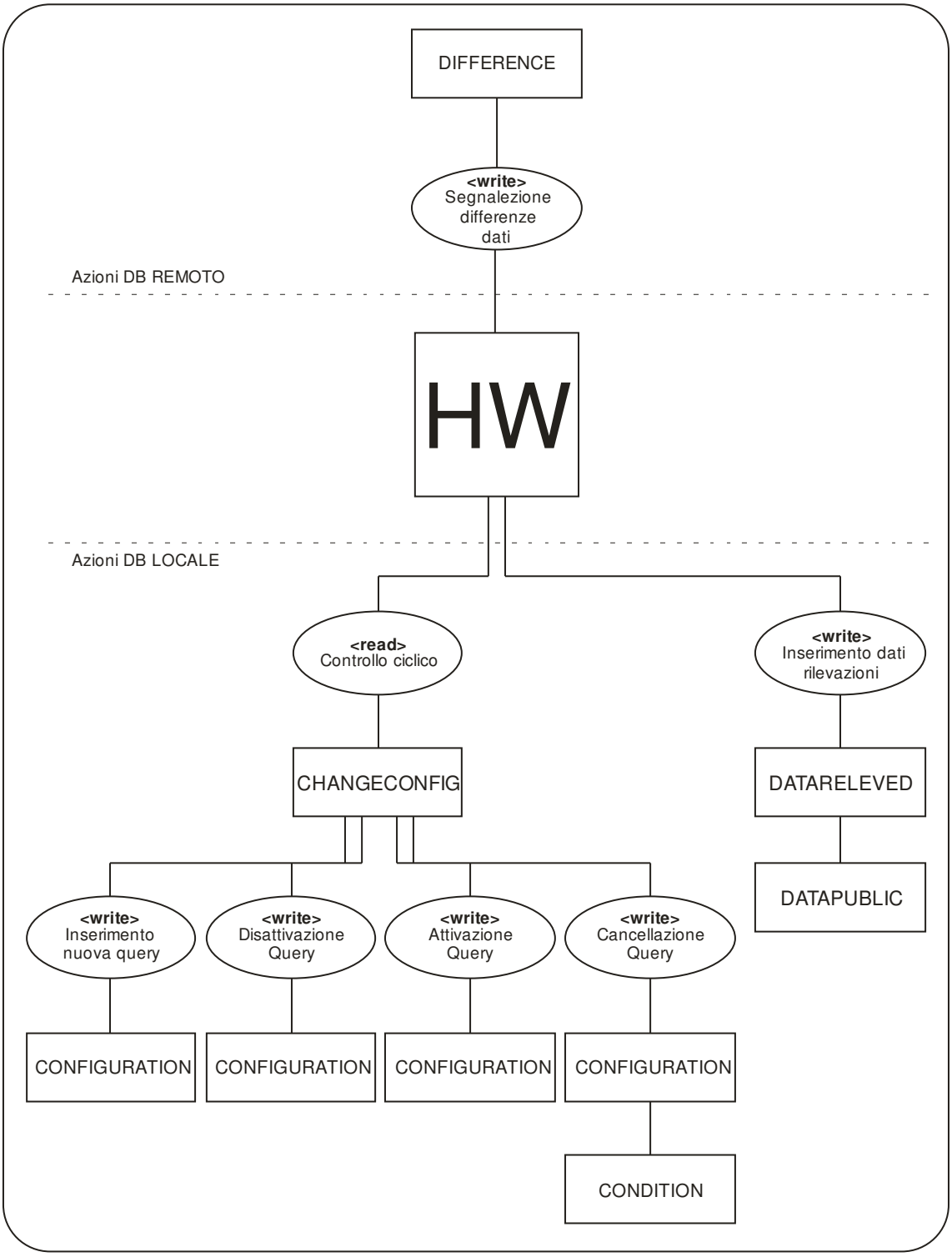


Figura 5.5: Flusso dati HW

5.2.2 Le API fornite da MySQL++

Per la scrittura del codice del wrapper, in particolare per le funzioni e le connessioni col DBMS, si è optato per l'utilizzo di un apposito sistema di interfaccia con le API native di MySQL, chiamato MySQL++ [27]. Questo software viene fornito dei soli sorgenti da compilarsi per l'architettura di utilizzo. Si basa su di una libreria dinamica ("dll" in Windows) e alcune librerie statiche, da includere nel codice del software.

MySQL++ è un potente wrapper per le API C di MySQL, scritto in linguaggio C++. Ha lo scopo di rendere semplice l'esecuzione di query e la manipolazione dei dati del database. I dati del database sono astratti e utilizzati come se fossero strutture dati standard del linguaggio C++ (STL Containers). Creato nel 1998 da Kevin Atkinson, MySQL++ mirava a diventare un software indipendente dal database interfacciato, infatti il primo nome fu SQL++. L'impossibilità di uno sviluppo efficace di un sistema così eterogeneo, fece propendere il programmatore per lo sviluppo esclusivo rivolto al database open source MySQL.

Il funzionamento di MySQL++ è molto simile alla maggior parte delle API per l'accesso ai DB. Il tipico schema di utilizzo è il seguente:

1) Apertura connessione
2) Costruzione ed esecuzione della query
3) Se eseguita con successo, iterazione dei risultati
4) Altrimenti, gestione degli errori.

La connessione al database avviene tramite la classe *Connection*, che gestisce l'intero processo di connessione al server MySQL. Questa classe offre differenti modalità per la connessione, quella tipicamente utilizzata è tramite il metodo TCPConnection.

La classe *Query* si occupa della creazione e dell'esecuzione della query sul DB. L'utilizzo più tipico di un oggetto query è tramite il passaggio di stringa (l'effettiva query SQL) e lanciarne l'esecuzione con l'apposito metodo. Esistono modalità di query più complesse, ad esempio tramite le Template Query è possibile creare una struttura standard di query con appositi tag all'interno e riutilizzarla più volte nello stesso programma. Un ulteriore metodo per costruire query è utilizzando le *Specialized SQL Structure* (SSQLS). Tramite SSQLS è possibile creare strutture dati C++ che rispecchiano la struttura di tabelle del DB. Ogni azione sui dati della struttura SSQLS corrisponde all'opportuna operazione di INSERT, UPDATE o DELETE sulla tabella del database. La modalità di interrogazione e manipolazione dei dati tramite SSQLS introduce un livello di astrazione ulteriore, rispetto alle modalità utilizzate dalle API standard di MySQL. Può risultare utile in situazioni in cui è necessario manipolare grandi quantità di dati tramite algoritmi che lavorano già con strutture dati C++, in questo caso la presenza o meno del database al livello software risulta mascherata.

Nel progetto affrontato si è optato per l'utilizzo delle SSQLS e il relativo codice è riportato in Appendice A.11.

5.3 Il Wrapper C++

Il codice del software wrapper è stato scritto in linguaggio C++, tramite l'ambiente di sviluppo e compilazione Microsoft Visual C++ 2008 Express Edition. Il programma è costituito da due thread. Il principale è perennemente in ascolto sulla porta COM a cui è virtualmente collegato l'AP gestendo così la ricezione e la manipolazione dei dati in arrivo dalla rete di sensori. Il thread secondario si occupa invece di monitorare il database, controllando la presenza o la modifica nelle configurazioni dei sensori e gestendone l'inoltro verso la WSN. Nel caso di studio, la configurazione settabile sugli ED, riguarda l'impostazione della frequenza di acquisizione e invio del dato di temperatura.

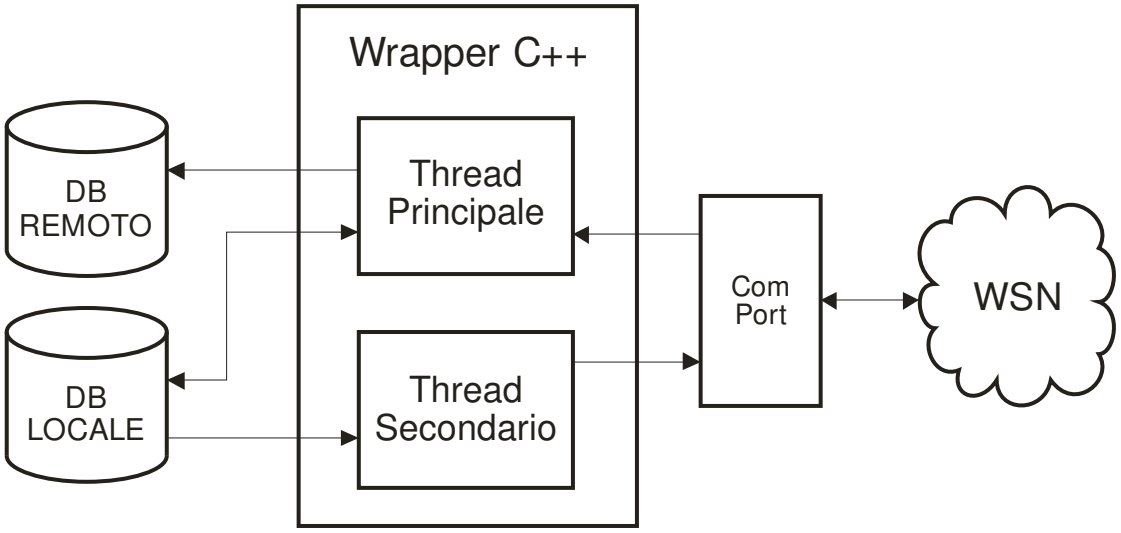


Figura 5.6: Schema software Wrapper c++

5.3.1 Thread Principale

Il thread principale si occupa primariamente della ricezione dei dati dall'AP e quindi dall'intera rete di sensori ad esso collegata. Alla ricezione di dati corretti dalla porta seriale viene scatenata l'esecuzione di varie funzioni per la memorizzazione e il controllo sul rispetto dei parametri del dato acquisito.
Il thread principale è rappresentato come schema a blocchi nella seguente Figura 5.7

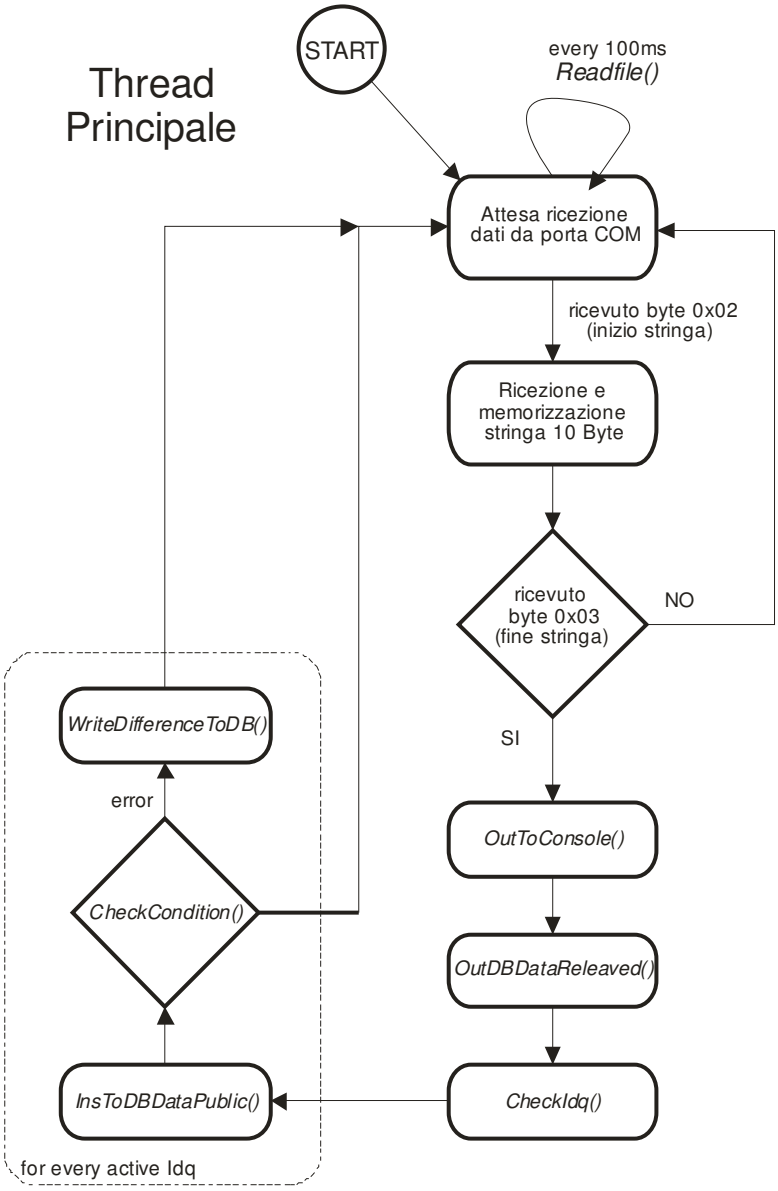


Figura 5.7: Diagramma di flusso Thread Principale

Il thread è basato su un ciclo senza fine *"while (true) {…} "* che esegue la funzione *Readfile()* di lettura dalla porta seriale. Questa funzione è una funzione "bloccante", ovvero ritorna il controllo solo alla ricezione del numero di Byte specificato, è stato quindi necessario configurare opportuni parametri di *"timing"* della porta COM che terminano la funzione, dopo un tempo fissato se non vi è stata ricezione di dati. L'aspetto di timing diventa fondamentale per rendere possibile al thread secondario l'utilizzo condiviso della porta seriale. Esso utilizza la stessa connessione per inviare i dati all'AP tramite la funzione *Writefile()*.

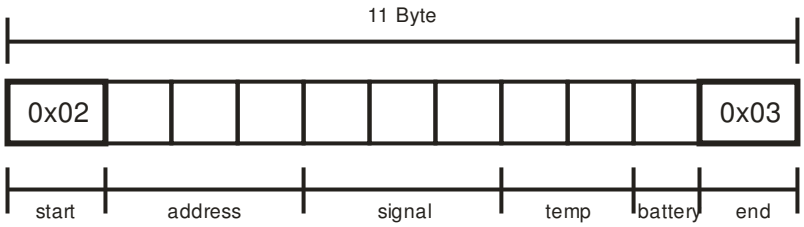


Figura 5.8: Stringa dati AP

Per garantire un certo livello di affidabilità dei dati ricevuti, essi sono formattati in un'apposita stringa di dimensione prefissata identificata con un byte di inizio e un byte di fine stringa (Figura 5.8).

Nel caso in esame il payload della stringa, contiene 9 byte, così suddivisi:

3B → indirizzo sensore
3B → potenza segnale radio
2B → temperatura rilevata
1B → stato batteria (voltaggio)

La stringa correttamente ricevuta è passata alla prima funzione *OutToConsole()*, la quale elabora i dati e genera in uscita sulla console un'apposita stringa testuale con le informazioni ricevute. Esistono due visualizzazioni in console: *verbose* (default) e *no-verbose* (Figura 5.9). E' possibile selezionare la modalità no-verbose, passando come parametro all'esecuzione del programma "noverbose". Un'ulteriore parametro specificabile a riga di comando è la visualizzazione del dato temperatura in *C*° (default) oppure *F* . In quest'ultimo caso sarà necessario lanciare l'eseguibile del wrapper con il parametro "f".

I parametri specificati a riga di comando influiscono solo sulla visualizzazione a console e non sulle successive memorizzazioni nel database.

[Verbose ON] `Node:HUB0,Temp: 91.2F,Battery:3.5V,Strength:000%,RE:no`
[Verbose OFF] `$HUB0, 89.6F,3.5,000,N#`

Figura 5.9: Stringa dati su console

Una volta terminata la visualizzazione su console, viene lanciata la funzione *OutToDataReleaved()*, essa effettua il primo inserimento dei dati nel database, in particolare nella tabella locale DataReleaved, dove sono memorizzati per ogni sensore oltre al dato di temperatura, i valori di batteria e segnale. Questa tabella è svincolata dal sistema di query centralizzato e può essere considerata come una tabella di controllo, utilizzabile per analisi sullo stato fisico della WSN, come ad esempio per monitorarne la copertura radio o lo stato delle batterie dei sensori.

Dopo l'inserimento in DataReleaved, inizia il controllo sulle query memorizzate. Per ogni query associata all'indirizzo del sensore, viene eseguita la funzione *InsToDBDataPublic()* , tabella principale che fornisce le informazioni specificate nelle query. Nel caso in esame essendo la temperatura l'unico dato rilevabile, nel

caso di più query attive sullo stesso sensore vi può essere una ridondanza dei dati, l'unicità della tupla è comunque garantita dalla chiave, includente oltre all'indirizzo del sensore e al datetime anche l'id univoco della query.

L'ultima funzione eseguita prima di ritornare a inizio ciclo è *CheckCondition()*. Essa verifica le condizioni associate ad ogni query attiva sul sensore, e verifica il rispetto dei vincoli stabiliti. Nel caso vi siano parametri non soddisfatti è scatenata *InsToDBDifference()*, la funzione si occupa di memorizzare sulla tabella Difference del database remoto, ipoteticamente sullo stesso calcolatore del Query Manager, le differenze, tra i vincoli stabiliti e i dati rilevati. La tabella Difference sarà poi adeguatamente monitorata dall'Event Manager remoto, che adotterà le opportune politiche stabilite per gestire i dati anormali.

5.3.2 Thread Secondario

Il thread secondario è rappresentato come schema a blocchi nella seguente Figura 5.10

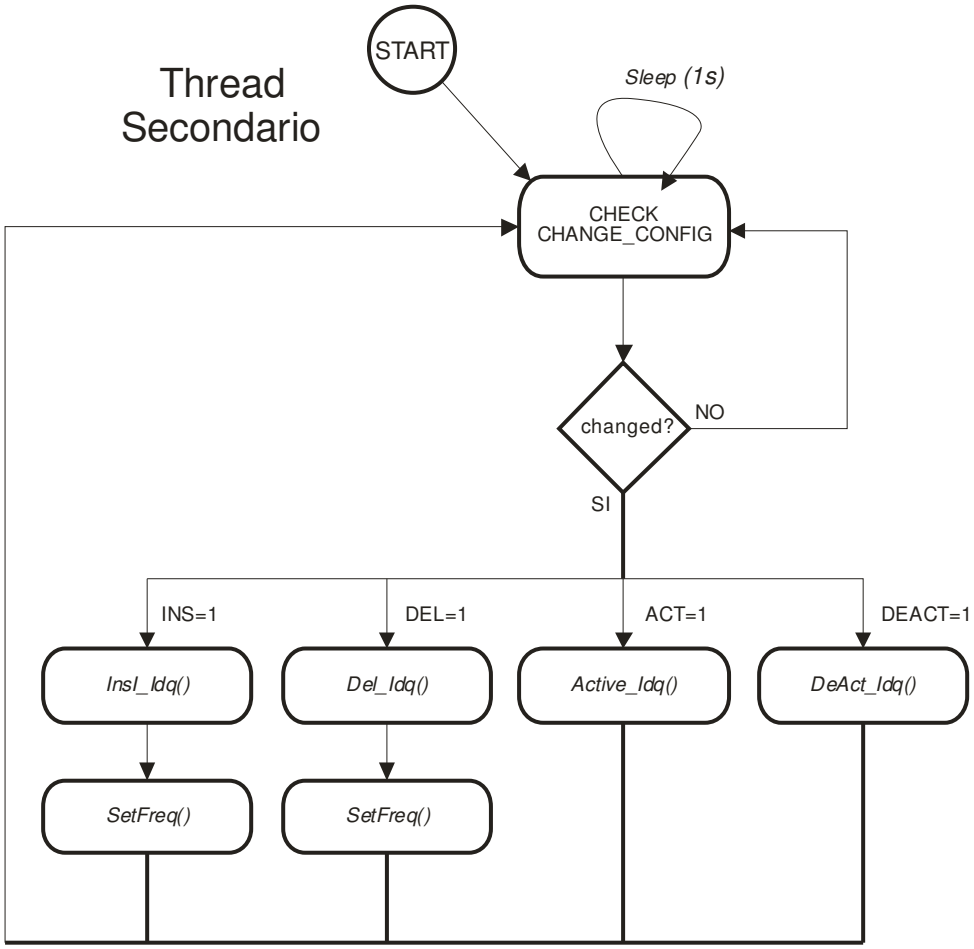


Figura 5.10: Diagramma di flusso Thread Secondario

Scopo principale del thread secondario è il monitoraggio e la gestione delle query in esecuzione sull'Host Wrapper. Ha inoltre il compito di inviare le configurazioni dei parametri ai sensori. Nel caso in esame l'unico parametro che si invia alla WSN è la frequenza di invio del dato di temperatura. Essa è fissata all'accensione a zero, ovvero non vi è rilevazione e invio del dato. La rete nel modello presentato è semplificata come auto configurata, ovvero la conoscenza del numero dei nodi, di ogni indirizzo e delle frequenze di rilevazione di ogni sensore è prestabilita. In un sistema più raffinato, vi saranno meccanismi per gestire ogni stato della rete, indipendentemente dal numero di sensori collegati. Meccanismi per accettare e configurare a tempo di esecuzione l'unione alla rete di nuovi sensori e la gestione dell'avvio e dello spegnimento di ogni nodo.

Il controllo di un cambio di configurazione sulle query, avviene tramite la tabella ChangeConfig che viene scandita periodicamente. La presenza di tuple nella suddetta tabella, presuppone una modifica sulle query di configurazione del sistema: può essere specificata una nuova query da attivare oppure una query da eliminare. Una volta gestite le righe presenti nella tabella esse vengono rimosse, cosicchè la tabella rimanga di dimensione estremamente contenuta e il controllo periodico eseguito su di essa risulti essere poco oneroso in termini computazionali e di velocità. Sono queste ultime le motivazione per le quali si è pensato di utilizzare una tabella separata per il controllo sui cambi di configurazione delle query, piuttosto che una scansione sulla tabella generale Configuration, che per sua concezione potrebbe assumere dimensioni anche elevate, rischiando così di pregiudicare le funzionalità del wrapper nel suo insieme.

Nella tabella ChangeConfig è possibile specificare tramite flag su campi booleani quattro azioni su ogni specifica query:

1) *Inserimento*
2) *Cancellazione*
3) *Disattivazione*
4) *Attivazione*

Ognuna di queste azioni è gestita da un'apposita funzione, lanciata in base al flag settato nella tupla di ChangeConfig.

La funzione *InsIdq()* gestisce l'inserimento di una query nel sistema. Si presuppone che l'inserimento da parte del Query Manager della tupla scatenante la funzione, sia preceduto dall'inserimento dei parametri della query stessa nelle tabelle Configuration e Condition, lasciando settato il campo *Active* su 0. In questo modo la prima operazione eseguita da InsIdq() riguarda l'attivazione della query, portando a 1 l'apposito campo della tabella Configuration. La seconda operazione è controllare i sensori interessati dalla query. In particolare si abilitano i sensori al momento non attivi e si confrontano le frequenze di rilevazione di quelli già abilitati. Si è deciso per dare priorità alla query che intende settare la frequenza di rilevazione maggiore,

decidendo di duplicare il dato (a frequenza più elevata) anche per le query che lo richiederebbe a frequenza inferiore.
Nel caso in cui la frequenza di un sensore debba essere modificata, viene lanciata la funzione *Set_Freq()*. Quest'ultima invia alla Access Point tramite porta COM, l'indirizzo del sensore e il tempo in secondi di ogni rilevazione. La stringa inviata è mostrata in Figura 5.11.

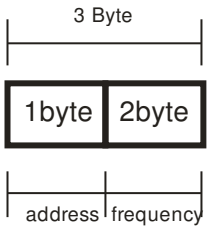


Figura 5.11: Stringa configurazione End Device

La stringa di configurazione, contiene al primo Byte, l'identificativo univoco del sensore cui inviare la configurazione. Questo indirizzo è un intero positivo a 8 bit, quindi con valori compresi tra 0-255, l'indirizzo 0 è riservato all'AP. I restanti 2 Byte della stringa, contengono il valore dell'intero a 16 bit, pari al tempo di rilevazione da settare nel sensore a cui è inviato. Un intero positivo a 16 bit rappresenta un numero fino a 65535 che, corrispondendo a un tempo in secondi, produce un intervallo massimo di rilevazione di circa 18 ore.

Oltre alla funzione di inserimento è presente la funzione duale di cancellazione, *DelIdq()*. La funzione si occupa di disattivare la query specificata, rimuovendo le rispettive tuple dalle tabelle Configuration e Condition. DelIdq() deve inoltre controllare i nodi sensori associati alla query rimossa e fermarne la rilevazione, inviando uno 0 come valore di frequenza, per quelli che non hanno altre query a cui corrispondere il dato. Nel caso in cui gli End Device associati alla query siano presenti su altre query attive, sarà necessario aggiornarne la frequenza di rilevazione, inviando quella inferiore.

Nel caso vi sia la necessità di fermare l'esecuzione della query temporaneamente, senza rimuoverne la presenza nel database e senza modificare la rete di sensori associata, è possibile utilizzare la funzione di disattivazione. Tramite il flag opportuno settabile con un tupla in ChangeConfig, si può eseguire la funzione *DeactIdq()*. Essa modifica il campo "Active" nella tupla corrispondente nella tabella Configuration. Il campo Active viene controllato ad ogni CheckCondition() del thread principale, nel caso in cui esso sia settato "false", non vengono eseguite le azioni successive sulla query, ovvero l'inserimento nel db e il controllo dei vincoli sulle condizioni. Si specifica che la disattivazione di una query è pensata come operazione temporanea, in quanto non modifica la configurazione della WSN e i sensori associati alla query continuano l'invio dei dati, anche se non effettivamente utilizzati. Può essere conveniente utilizzare la disattivazione della query, quando il tempo in cui la query deve cessare la sua azione è breve in proporzione al tempo di riconfigurazione di tutti i sensori in caso di cancellazione della query stessa.

Esiste la funzione duale alla disattivazione per far riprendere l'azione della query: *Act_Idq()* . La funzione si limita a portare a un valore "true" il campo Active della corrispondente riga della query nella tabella Configuration.

5.4 L'Access Point e comunicazione con la WSN

La comunicazione del wrapper verso l'EZ430-RF2500 adibito ad Access Point verso la WSN collegata, avviene tramite la porta COM virtuale creata dal driver dell'interfaccia USB. Il software dimostrativo fornito a corredo con l'AP è rappresentato tramite diagramma di flusso in Figura 5.12. Le modifiche introdotte per l'utilizzo nel sistema in esame, sono descritte nella Sezione 5.4.2.

5.4.1 Il software AP standard

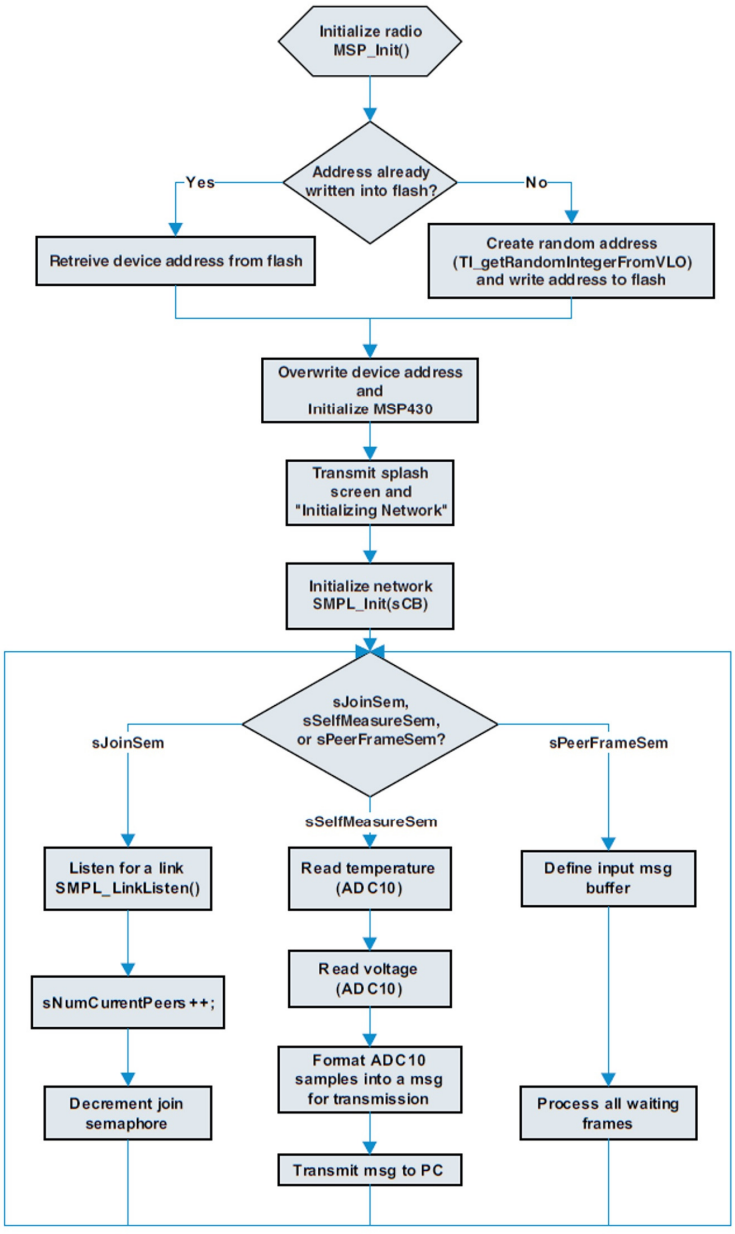


Figura 5.12: Diagramma di flusso software AP standard

L'inizializzazione del dispositivo all'avvio avviene tramite le funzioni BSP_Init(), che si occupa di inizializzare la comunicazione tra il microcontrollore MSP430 e il transeiver CC2500 e la funzione MCU_Init() che nello specifico configura la modalità di funzionamento del MSP430. In particolare viene settata una frequenza di funzionamento di 8Mhz, viene abilitato un interrupt Timer_A eseguito a un intervallo di 1 secondo e viene abilitata la comunicazione seriale tramite interfaccia (USCI) UART, configurando un velocità di 9600 Baud in RX/TX.

Nella prima inizializzazione i sensori, sia AP che ED, generano e memorizzano nella memoria flash interna un indirizzo casuale, utilizzato per l'identificazione univoca del dispositivo nella rete. Questo indirizzo "hardware" viene inoltre utilizzato per riconoscere gli ED, momentaneamente disconnessi o riavviati, permettendo all'AP, di riassegnare il precedente indirizzo "virtuale" (LinkID), utilizzato per la comunicazione col protocollo SimpliciTI.

I diversi rami di esecuzione del programma principale sono condizionati da appositi semafori: sPeerFrameSem, aJoinSem e sSelfMeasureSem.

Il semaforo sPeerFrameSem è attivato quando avviene la ricezione di un frame radio da un ED già connesso all'AP. Il semaforo è incrementato dall'apposita funzione sCB, invocata tramite interrupt dal protocollo di rete, quando un frame è in attesa di ricezione. La ricezione effettiva del frame è delegata al ramo di esecuzione condizionato dal semoforo, che al suo termine viene riportato a zero. Il frame ricevuto dall'ED, contiene le informazioni sul dato di temperatura rilevato e sul voltaggio della batteria. Prima di inoltrare questa informazione alla porta seriale, viene completata con l'informazione sulla qualità del segnale, ottenuta grazie a un'apposita funzione del protocollo SimpliciTI. I dati vengono poi formattati in un'apposita stringa e spediti alla porta seriale.

Il ramo di esecuzione condizionato dal semaforo aJoinSem, anch'esso attivato dalla funzione sCB, viene eseguito nel caso in il frame ricevuto sia il primo frame di connessione di un nuovo ED. In questo caso viene lanciata dal'AP la funzione SMPL_LinkListen() che ritorna il linkID, con cui effettuare le successive comunicazioni con gli End Device.

L'ultimo semaforo implementato è sSelfMeasureSem, esso ha il compito di stabilire il momento in cui effettuare la rilevazione della temperatura da parte dell'AP e di inviarla come stringa alla seriale in modo del tutto analogo a quello degli ED. sSelfMeasureSem è attivato dal Timer_A, con una frequenza di una volta al secondo.

5.4.2 Il software AP modificato

Nello sviluppo del sistema, oggetto di questo lavoro di tesi, è stato necessario modificare il software dell'Access Point per implementare le ulteriori funzionalità richieste.

Sono state rimosse la gestione della formattazione della stringa di output e la conversione dei dati da parte dell'AP, riducendo così i dati trasmessi verso la porta seriale in modo significativo. In questo modo vengono trasferiti solo 9 Byte contenenti i dati rilevati, la conversione dei dati, i caratteri testuali e la formattazione per l'output su console è ora totalmente a carico del software wrapper. Ulteriormente il wrapper deve ora gestire la conversione del dato temperatura nelle diverse unità di misura, gradi Celsius e gradi Fahrenheit.

E' stata modificato in modo significativo la modalità di rilevazione della temperatura, infatti si è introdotto un contatore e un valore di riferimento di frequenza. In questo modo la rilevazione della temperatura è disattivata all'avvio (non più una volta ogni secondo), essendo la frequenza impostata a zero come default. Sarà il wrapper che invierà in modo specifico la frequenza di rilevazione che sarà poi memorizzata in una variabile. Il contatore è gestito dal Timer_A e incrementato di una unità ad ogni ciclo, ovvero una volta al secondo. La stessa modalità di configurazione della frequenza di rilevamento della temperatura è stata implementata anche sugli End Device.

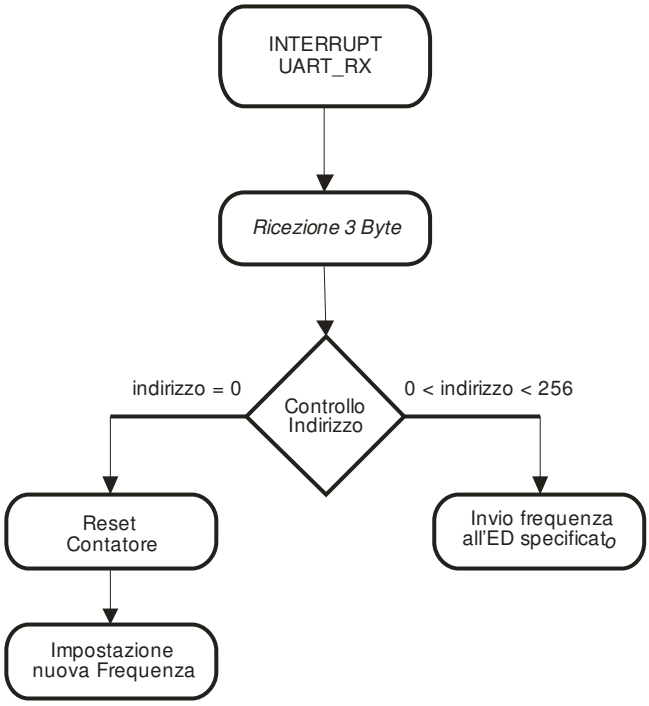


Figura 5.13: Diagramma di flusso modifiche software AP

Modifiche significative nella parte di ricezione di dati dalla porta seriale. La sezione di codice all'interno della routine di interrupt gestente il buffer USCIA di ricezione dalla porta di comunicazione è rappresentata in modo grafico in Figura 5.13. Alla ricezione della stringa di configurazione viene valutato l'indirizzo specificato, se corrisponde a "0" il valore di frequenza è indirizzato all'AP che dovrà resettare il proprio

contatore e impostare la nuova frequenza di rilevazione. Nel caso in cui l'indirizzo specifichi un ED connesso, sarà eseguita la funzione *TrasmitToEd()* che trasmetterà in modalità radio il dato di frequenza all'opportuno End Device.

5.5 Gli End Device e l'acquisizione dei dati ambientali (temperatura)

Gli End Device rappresentano i veri e propri nodi della rete di sensori. Essi sono collegati in modalità "a stella" ad un solo Access Point, al quale comunicano i dati. Il joining con l'AP a distanza di rilevamento è effettuato all'accensione del dispositivo. Il numero massimo di ED collegabili allo stesso AP è fissata dal protocollo di rete SimpliciTI a 30, è un valore comunque modificabili intervenendo sulle costanti che lo definiscono. La portata massima di comunicazione radio può essere estesa tramite l'utilizzo di sensori appositamente programmati per questo scopo, essi prendono il nome di Range Extender (RE). Il RE legge i frame ricevuti e li reinoltra sulla rete a maggiore potenza.

5.5.1 Il software ED standard

Il digramma di flusso del software preinstallato sull'EZ430-RF2500 adibito a End Device è rappresentato in Figura 5.14.

Dopo l'inizializzazione hardware del dispositivo tramite BSP_Init(), viene creato e assegnato l'indirizzo hardware se non è già presente, esattamente come accade per l'AP. Per quanto riguarda invece l'inizializzazione della rete, vi è una sostanziale differenza, in quanto alla funzione SMPL_Init() non è associata alcuna funzione di callback (sCB) avente il compito di gestire la ricezione dei frame radio. Infatti la comunicazione dell'ED è unidirezionale in trasmissione verso l'AP.

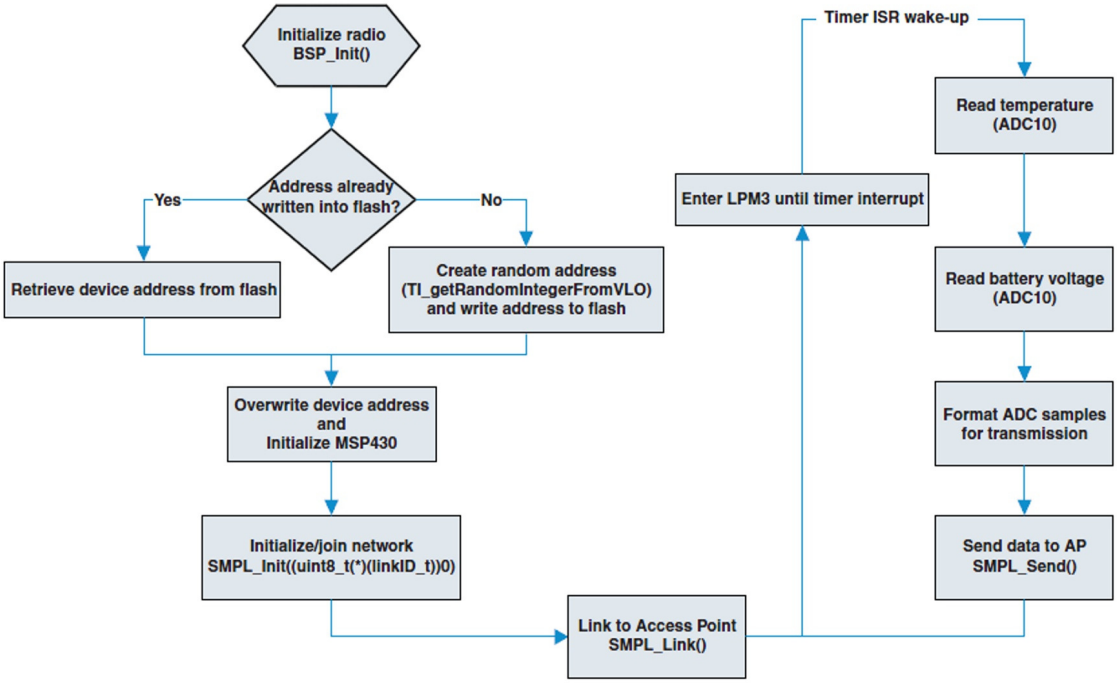


Figura 5.14: Diagramma di flusso software ED standard

L'End Device si trova per la maggior parte del suo tempo di esecuzione in una modalità a basso consumo (LPM3), con spegnimento della cpu e delle funzionalità radio. Sarà il Timer_A ad ogni sua esecuzione ciclica una volta al secondo, a riattivare il dispositivo, riportandolo nella modalità a consumo normale e con la trasmissione radio abilitata. Alla riaccensione del nodo vi è la rilevazione della temperatura e dello stato della batteria, l'invio all'AP del messaggio contente queste informazione e il ritorno alla modalità LPM3.

5.5.2 Il software ED modificato

Le modifiche effettuate al codice dell'End Device sono state sostanziali. Innanzitutto è stata modificata la modalità radio, abilitando la ricezione di frame tramite una funzione di callback sCB, dopodiché è stata modificata la modalità di invio dei dati, ora non più a frequenza costante di una volta al secondo, ma impostabile tramite la ricezione del valore desiderato. In maniera del tutto analoga al caso dell'AP, vi sono un contatore (incrementato ad ogni ciclo del Timer_A) e una variabile frequenza inizialmente settata a zero (Figura 5.15).

E' importante specificare che la necessità di abilitare la ricezione radio, impone l'impossibilità di posizionare nella stato a basso consumo di "sleep" il ricevitore CC2500, esso devo rimanere nello stato "idle". Per quanto riguarda il microcontrollore MSP430 anche per esso è stata rimossa la modalità a basso consumo LPM3. Il caso di studio non ha preso in esame valutazioni e ottimizzazioni sul consumo, ma è bene specificare che è possibile migliorare il funzionamento del dispositivo sotto questo profilo. Infatti il

microcontrollore e il transceiver posso sfruttare alcuni interrupt hardware, con i quali è possibile utilizzare le modalità a risparmio energetico del MSP430. Diventa così a carico del CC2500 l'interruzione tramite interrupt della Low Power Mode e il ritorno al funzionamento normale. Nella sezione successiva sarà fatta un confronto tra le modalità operative e le differenze di consumo nei vari stati dei dispositivi

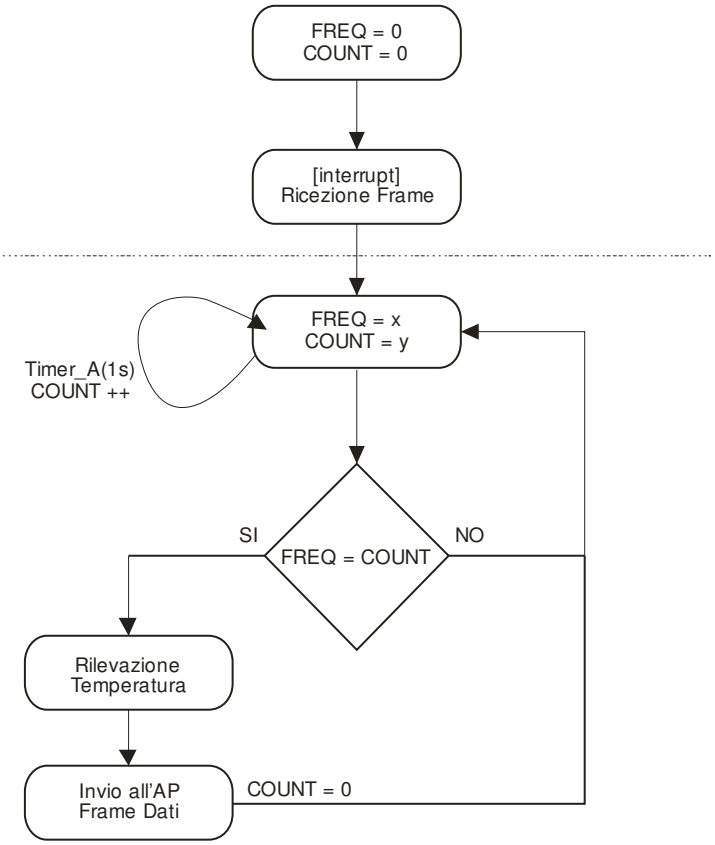


Figura 5.15: Diagramma di flusso modifiche software ED

5.6 Analisi prestazionale e consumi

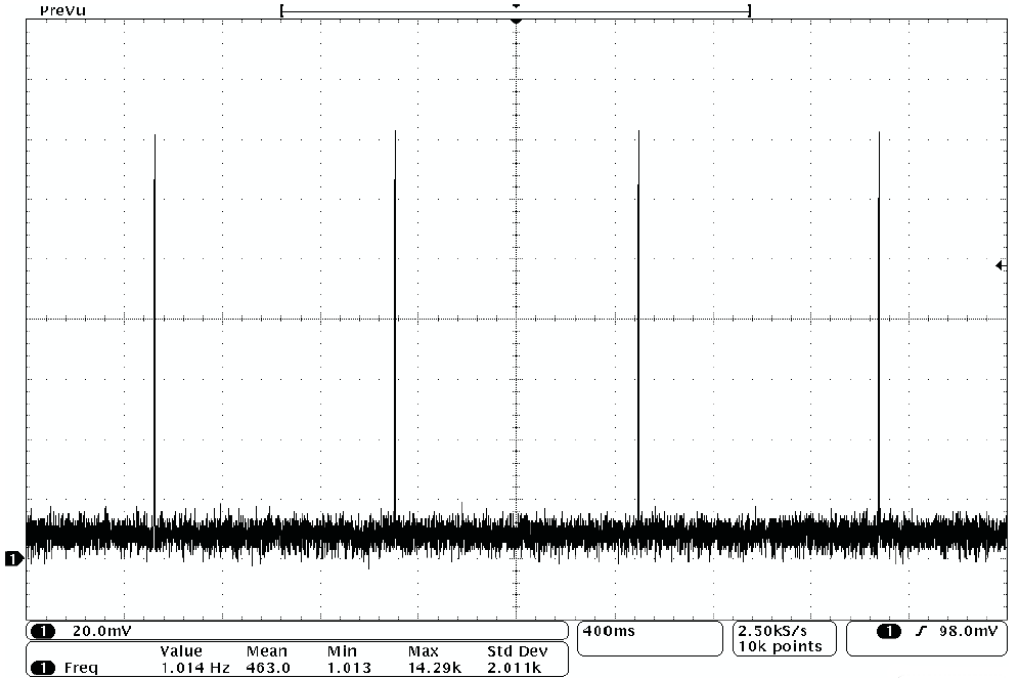


Figura 5.16: Corrente assorbita ED

Gli aspetti sulle prestazioni e il consumo sono di fondamentale importanza per la funzionalità di una rete di sensori. La durata della batteria deve essere un parametro quanto il più possibile ottimizzato.

In Figura 5.16 si mostra in un arco temporale di 4 secondi, l'assorbimento di corrente di un ED, con software standard. Si vede chiaramente come il dispositivo rimanga per la maggior parte del tempo di funzionamento nello stato a basso consumo (LPM3), con picchi di assorbimento solo nel momento in cui si "sveglia", legge i dati li invia. E' chiaro come la durata della modalità a consumo normale sia in proporzione al tempo complessivo di funzionamento assai ridotta. Una modalità di funzionamento come questa garantisce tempi di utilizzo senza manutenzione assai elevati. In Figura 5.17 si mostra nel grafico la vita operativa aspettata senza cambio di batterie in funzione del periodo di rilevazione della temperatura. [13]

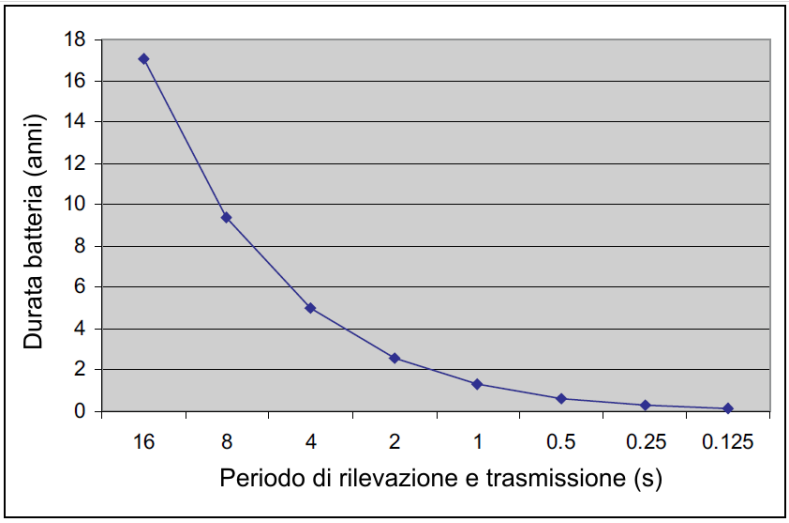


Figura 5.17: Stima vita operativa ED

Per capire in dettaglio la differenza di consumo tra le diverse modalità di funzionamento, si riporta in tabella i valori di assorbimento di corrente per gli integrati del dispositivo (Tabella 5.1).

Analizzando i valori di corrente, risulta un totale per la modalità a basso consumo di:
900 nA + 400 nA = **1.3 µA** (MSP43= LPM3, CC2500=Sleep)
Mentre nella frazione di secondo in cui avviene la rilevazione e l'invio dei dati si ha:
2.7 mA + 21.3 mA = **24 mA** (MSP430=Active, CC2500=TX)

Volendo riportare il consumo in idle dell'ED funzionante col software modificato per il nostro sistema di prova, si ottiene:
2.7 mA + 1.5 mA = **4.2 mA** (MSP430=Active, CC2500=Idle)
Si tratta di un valore di ordine di grandezza superiore rispetta al valore in idle dell'ED standard.

HARDWARE	***CORRENTE***	
	Valore	*Unità*
Sleep Mode		
MSP430 low-power mode 0 (LPM0)	1.1	mA
MSP430 low-power mode 3 (LPM3)	900	nA
CC2500 Sleep State	400	nA
MSP430 Active Mode		
8 MHz = DCO = SMCLK, 3 V	2.7	mA
MSP430 ADC10		
fADC10CLK = 5.0 MHz, ADC10ON = 1, REFON = 1, REFOUT = 0, ADC10DIV = 0x4 (ADC10CLK/5)	850	µA

CC2500 Modes		
Idle	1.5	mA
Receive (RX) (weak input signal, DEM_DCFILT_FILT_OFF = 0, 250 kbps)	18.8	mA
Transmit (TX) (250 kbps, 0-dB output power)	21.3	mA

Tabella 5.1: Consumi EZ430-RF2500

Risulta interessante notare come il maggiore assorbimento di corrente avvenga durante la trasmissione o la ricezione radio. In Figura 5.18 è mostrato il consumo durante l'esecuzione della funzione SMPL_Send() per l'invio di un frame sul canale radio.

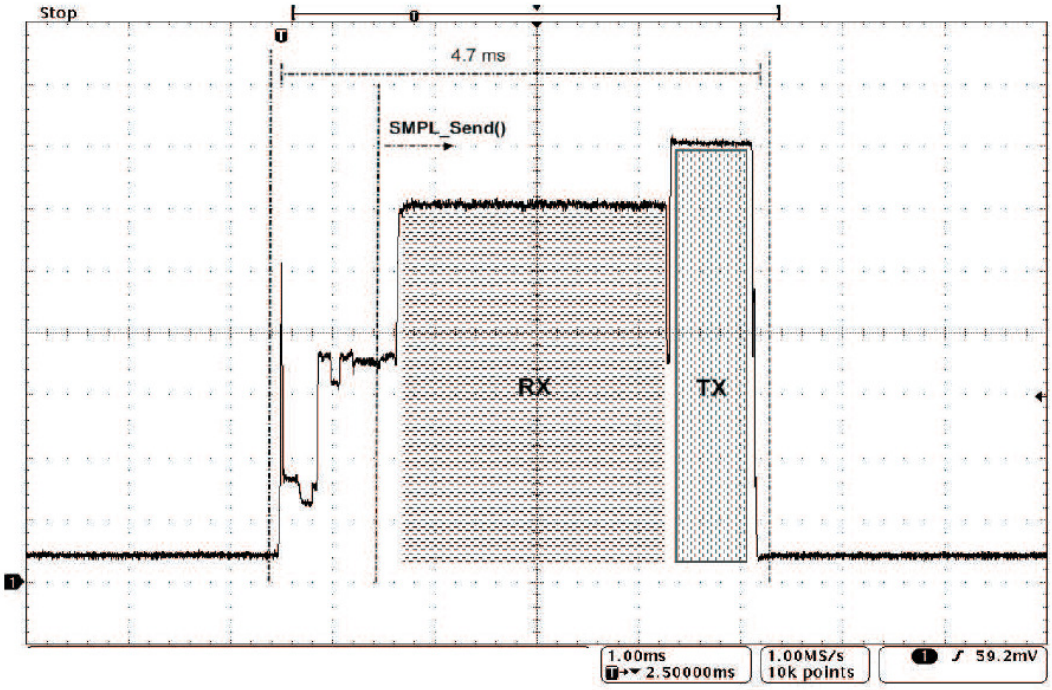


Figura 5.18: Analisi consumo in fase di trasmissione

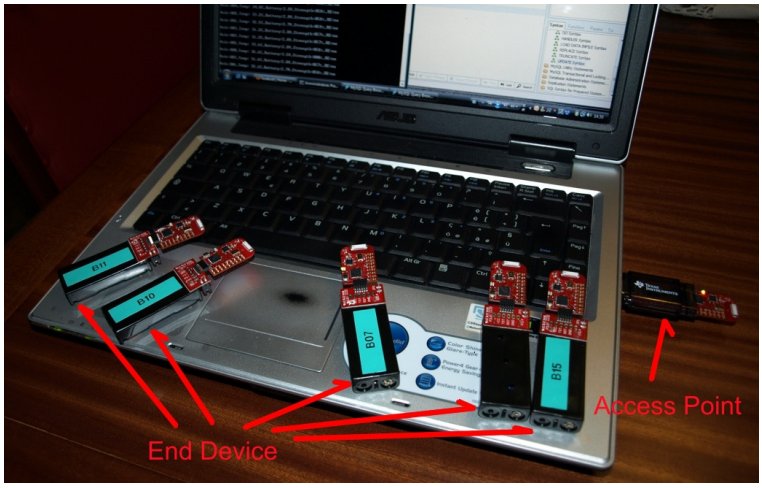
Capitolo 6

Risultati sperimentali

Nel seguente capitolo sono riportati i risultati di alcune prove effettuate sul sistema realizzato. Si propongono alcune situazioni differenti, analizzando le funzionalità di raccolta e memorizzazione dell'Host Wrapper in base alle configurazioni dei nodi wireless e delle query attive nel sistema.

6.1 Ambiente di test

L'ambiente di prova con cui si sono stati effettuati gli esperimenti è costituito da un Notebook su cui è installato il software Wrapper C++ e il DBMS MySQL col quale è possibile interagire utilizzando l'apposito pannello di amministrazione. Tramite il personal computer sono stati programmati gli EZ430-RF2500 e in particolare si sono avuti a disposizione cinque nodi sensori programmati come End Device e un ulteriore dispositivo, programmato come Access Point. L'AP è stato collegato alla porta USB, tramite l'apposito adattatore per la comunicazione dei dati col wrapper.



6.2 Controllo temperatura di un ambiente

La prima prova effettuata è stata svolta per simulare il funzionamento di una ampia rete di sensori volti al monitoring della temperatura in un determinato ambiente. Sono stati utilizzati tutti e cinque gli End Device a disposizione, configurando anche l'Access Point all'invio della propria temperatura, avendo così un totale di sei misurazioni di temperatura.

Per rendere l'esperimento più significativo durante la prova sono stati effettuati (in modo abbastanza casuale) dei raffreddamenti e riscaldamenti dei dispositivi, per ottenere un andamento meno costante delle temperature. Sono stati settati gli stessi vincoli di temperatura massima e minima e sono state riportate le differenze rilevate.

6.2.1 Configurazione sistema

La Figura 6.1 riporta la configurazione delle frequenze sui dispositivi (tabella di sistema *Device*), mentre la Figura 6.2 riporta la tabella (*Configuration*) delle query attive durante la prova. Per ogni nodo vi era una sola query attiva.

Si ricorda che le *condizioni* da verificare erano le medesime per tutte le query, corrispondendo a una temperatura massima (MAX) di 40 C° e una temperatura minima (MIN) di 15 C°.

IDDEV	FREQ_SETTED
0	10
1	9
2	7
3	3
4	17
5	15

Figura 6.1: Frequenze dispositivi

IDQ	IDDEV	FREQ	ACTIVE
1	0	10	1
2	1	9	1
3	2	7	1
4	3	3	1
5	4	17	1
6	5	15	1

Figura 6.2: Query attive

6.2.2 - Risultati

Il tempo complessivo di rilevazione è stato di 8 minuti. Si riportano nelle successive tabelle parte dei risultati ottenuti, in particolare l'output su *Console* (Figura 6.3), l'output sulla tabella *DataReleaved* (Figura 6.4), in cui sono memorizzate anche le informazioni su batteria e segnale, e l'output sulla tabella *DataPublic* (Figura 6.5).

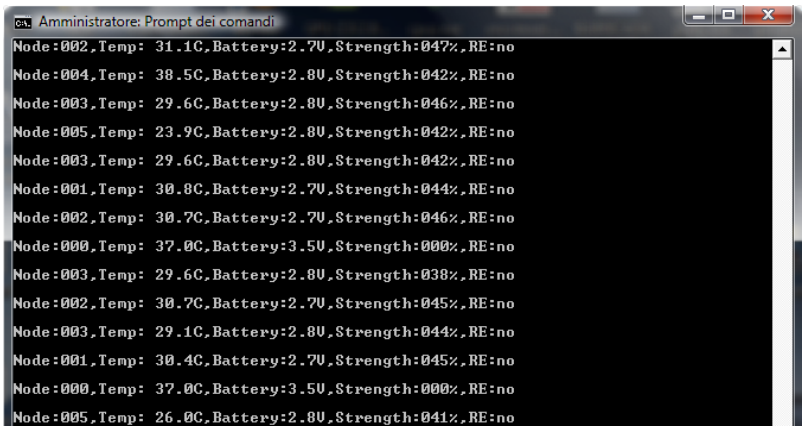


Figura 6.3: Output su console

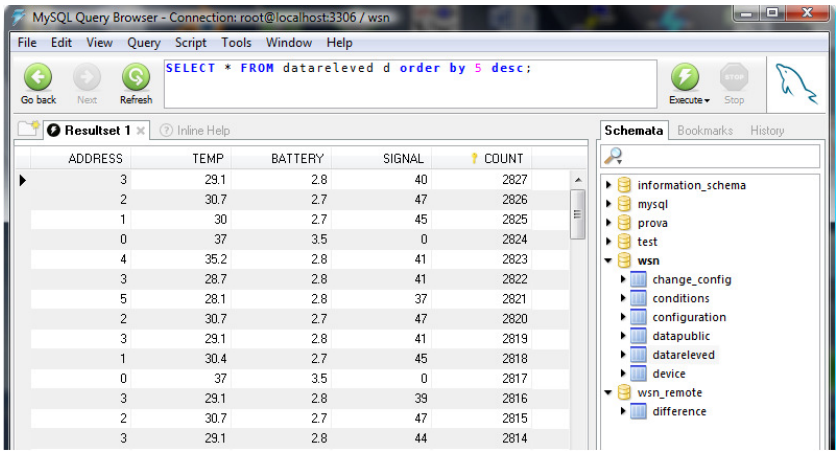


Figura 6.4: Output su DataReleaved

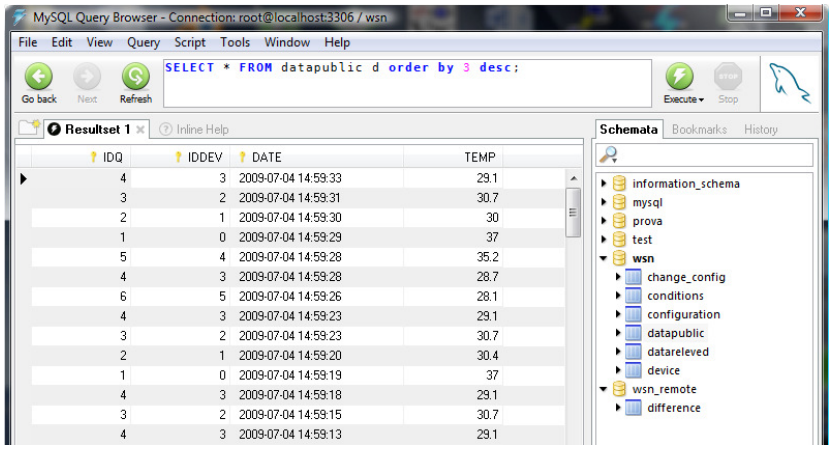


Figura 6.5: Output su DataPublic

In dettaglio l'andamento delle temperature dei singoli nodi della rete è stata la seguente:

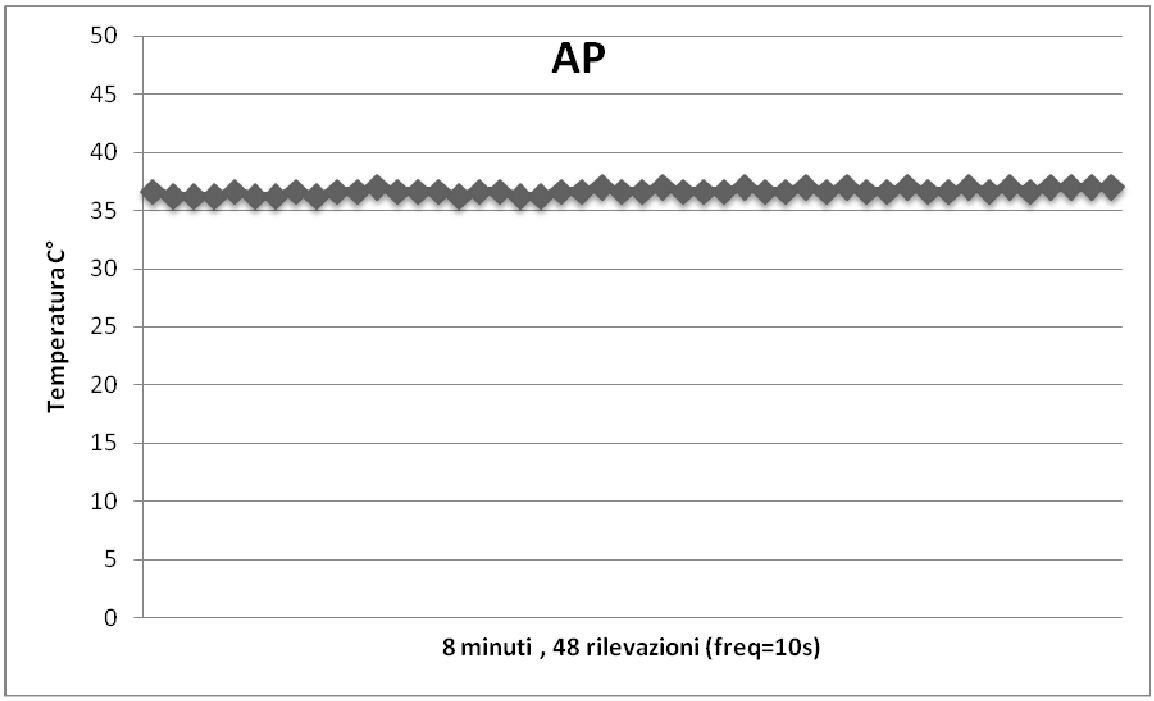


Figura 6.6: Risultati AP

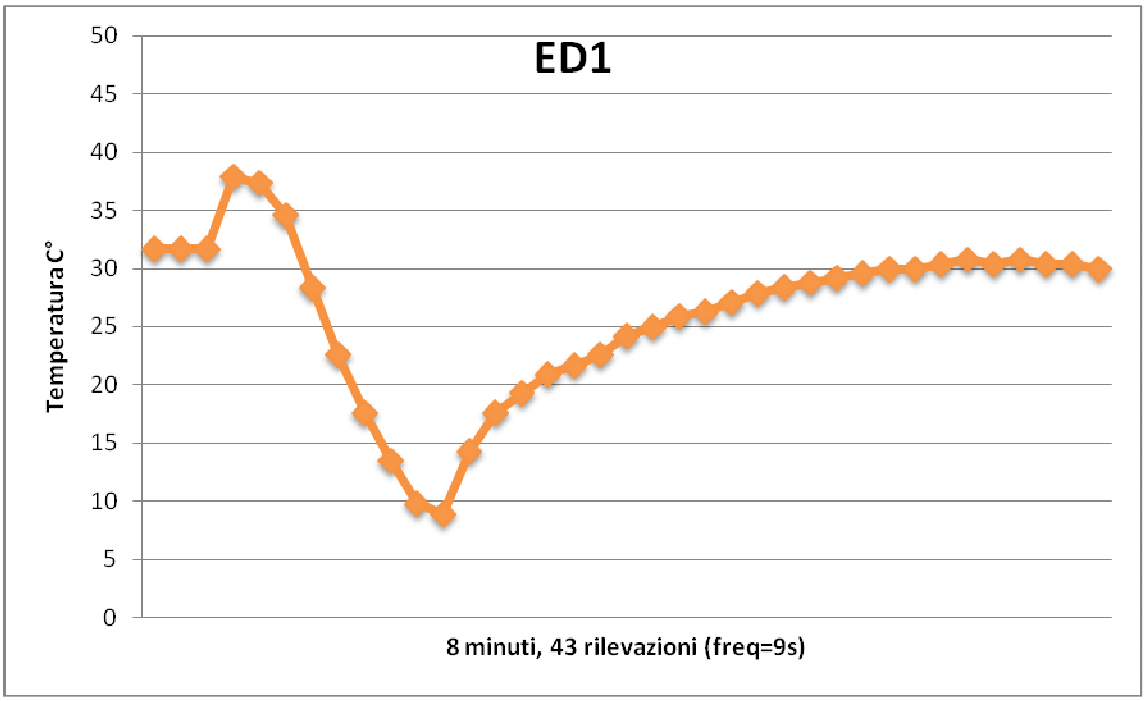


Figura 6.7: Risultati ED1

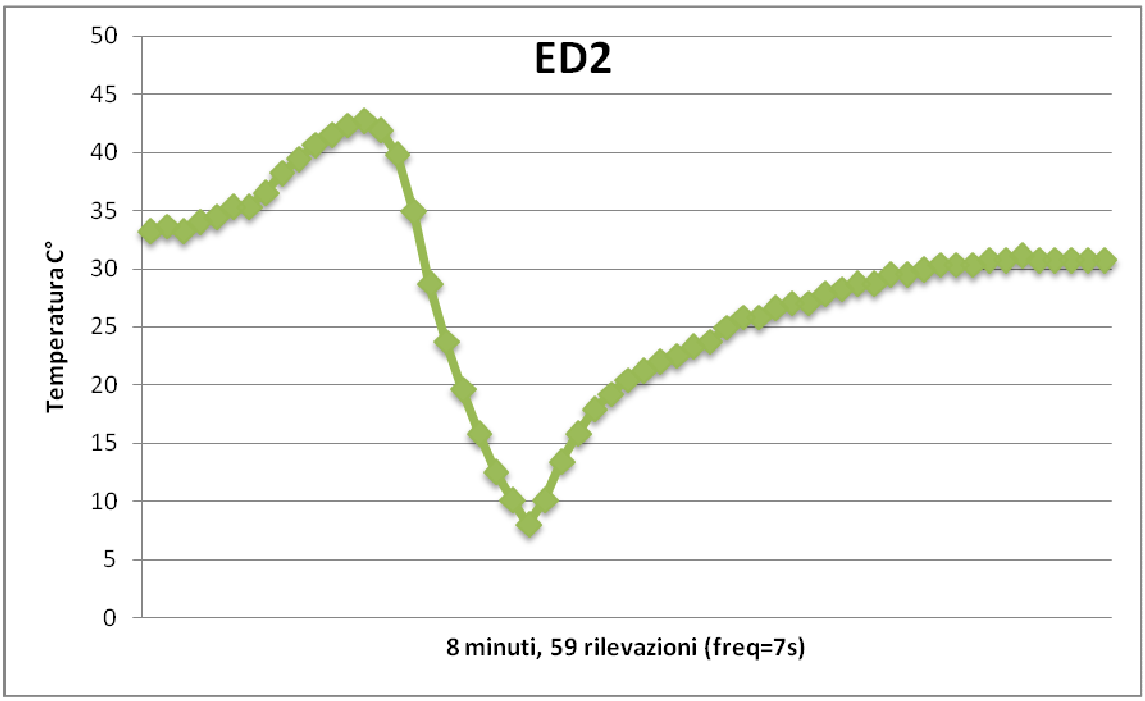


Figura 6.8: Risultati ED2

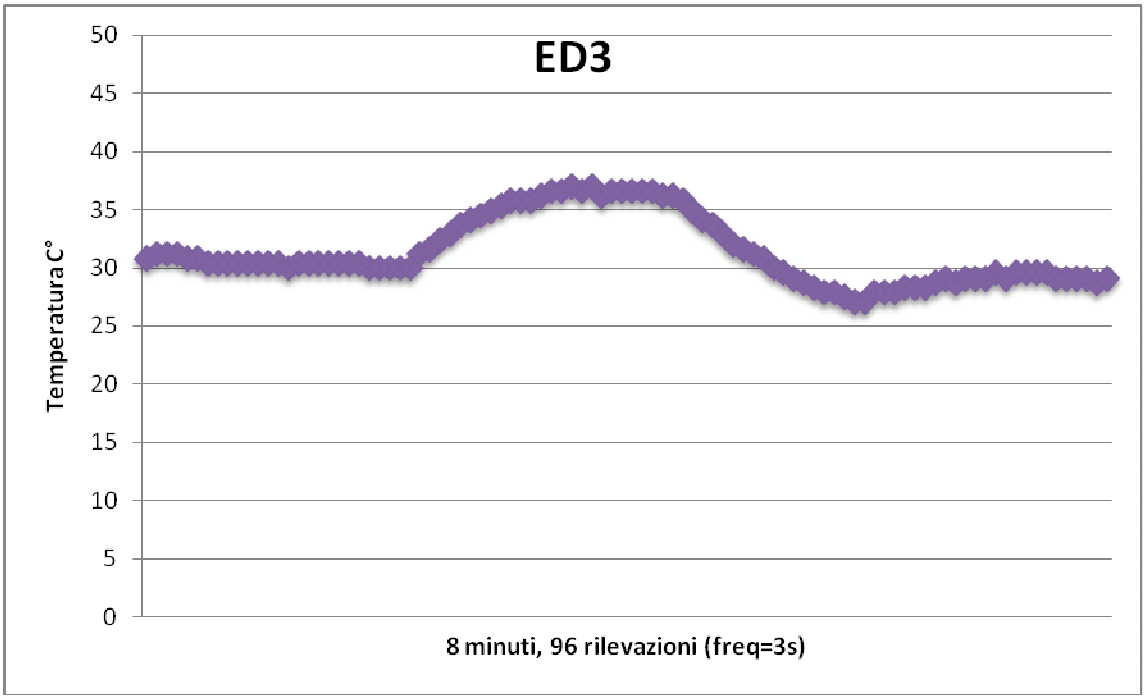


Figura 6.9: Risultati ED3

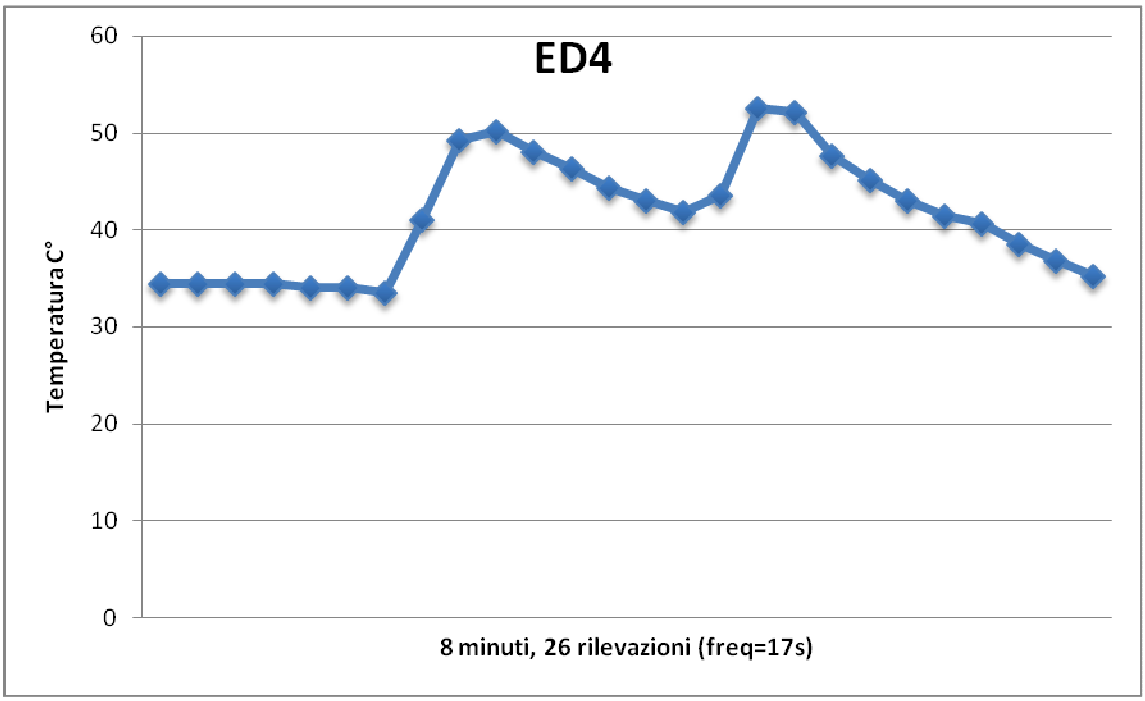


Figura 6.10: Risultati ED4

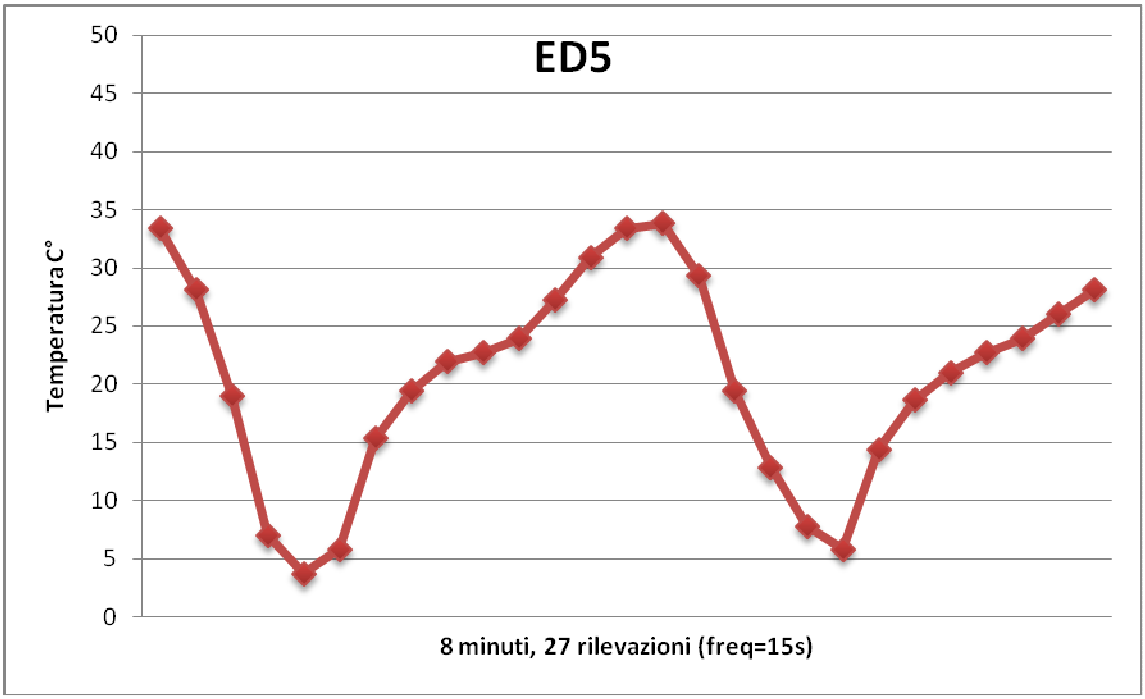


Figura 6.11: Risultati ED5

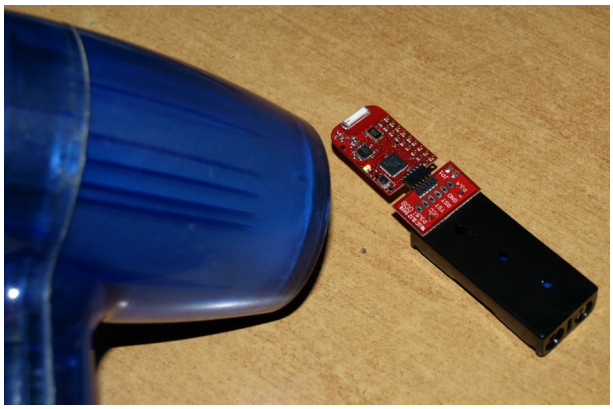
Alcuni End Device hanno superato nel corso delle misurazioni i valori di temperatura massima e minima impostati (in positivo o negativo). La tabella (*Difference*) risultante al termine della prova è la seguente (Tabella 6.1).

IDQ	DATE	MAX	MIN
3	2009-07-04 14:53:11	0,6	
6	2009-07-04 14:53:13		-8
3	2009-07-04 14:53:19	1,5	
3	2009-07-04 14:53:26	2,3	
6	2009-07-04 14:53:30		-11,3
3	2009-07-04 14:53:34	2,7	
3	2009-07-04 14:53:41	1,9	
6	2009-07-04 14:53:48		-9,2
5	2009-07-04 14:54:17	1	
5	2009-07-04 14:54:34	9,2	
2	2009-07-04 14:54:35		-1,5
3	2009-07-04 14:54:37		-2,5
3	2009-07-04 14:54:45		-4,9
2	2009-07-04 14:54:46		-5,2
5	2009-07-04 14:54:51	10,1	
3	2009-07-04 14:54:53		-7
2	2009-07-04 14:54:57		-6,1
3	2009-07-04 14:55:02		-4,9
2	2009-07-04 14:55:08		-0,7
5	2009-07-04 14:55:08	8	
3	2009-07-04 14:55:10		-1,6
5	2009-07-04 14:55:25	6,3	
5	2009-07-04 14:55:43	4,3	
5	2009-07-04 14:56:00	3	
5	2009-07-04 14:56:17	1,8	
5	2009-07-04 14:56:35	3,5	
5	2009-07-04 14:56:52	12,5	
6	2009-07-04 14:56:59		-2,2
5	2009-07-04 14:57:09	12,1	
6	2009-07-04 14:57:16		-7,2
5	2009-07-04 14:57:26	7,6	
6	2009-07-04 14:57:33		-9,2
5	2009-07-04 14:57:43	5,1	
6	2009-07-04 14:57:50		-0,6
5	2009-07-04 14:58:00	3	
5	2009-07-04 14:58:18	1,4	
5	2009-07-04 14:58:35	0,6	

Tabella 6.1: Tabella Difference

6.3 Controllo pendenza positiva nel riscaldamento

In questa prova si è verificato il funzionamento della condizione sulla pendenza positiva nella crescita del dato temperatura (*PDH*, *PDL*). Con questi due parametri è possibile specificare un intervallo in cui deve cadere la variazione di due successive misurazioni. Ad esempio specificando un PDH = 3 e un PDL = 2, si indica al wrapper che la differenza tra una misurazione e la successiva deve essere un valore positivo compreso tra 2 e 3 gradi.



6.3.1 Configurazione sistema

Per la prova si è utilizzato un solo End Device configurato a una frequenza di rilevamento di 2 secondi, con una sola query attiva avente come condizioni PDH = 2 e PDL = 1.

Il periodo di campionamento è stato di circa 1 minuto per un totale di 32 misurazioni. La temperatura del sensore, riscaldato tramite un asciugacapelli, è passata dai 28,4 C° iniziali ai 60,2 C° finali.

6.3.2 Risultati

In Tabella 6.2 sono riportati i dati delle misurazioni, ordinati in successione cronologica. Nella terza colonna è riportato il calcolo del delta T, tra ogni valore e il precedente. Nella quarta, invece, la differenza del delta sulle condizioni impostate. Un quantità positiva indica il valore di superamento di PDH, una quantità negativa indica il valore al di sotto di PDL.

TEMPO	T (C°)	Delta T	Differenza
2009-07-04 09:32:35	28,4	/	/
2009-07-04 09:32:36	28,4	0	-1
2009-07-04 09:32:38	28,8	0,4	-0,6
2009-07-04 09:32:40	31,2	2,4	0,4
2009-07-04 09:32:42	33,3	2,1	0,1
2009-07-04 09:32:44	34,6	1,3	0
2009-07-04 09:32:46	35,8	1,2	0
2009-07-04 09:32:49	36,6	0,8	-0,2
2009-07-04 09:32:51	37,4	0,8	-0,2
2009-07-04 09:32:53	38,3	0,9	-0,1
2009-07-04 09:32:55	39,5	1,2	-0,1
2009-07-04 09:32:57	40,7	1,2	0
2009-07-04 09:32:59	41,6	0,9	-0,1
2009-07-04 09:33:00	43,2	1,6	0
2009-07-04 09:33:02	44,9	1,7	0
2009-07-04 09:33:04	46,5	1,6	0
2009-07-04 09:33:06	48,2	1,7	0
2009-07-04 09:33:08	50,2	2	0
2009-07-04 09:33:10	51,5	1,3	0
2009-07-04 09:33:12	53,1	1,6	0
2009-07-04 09:33:14	55,2	2,1	0,1
2009-07-04 09:33:16	57,7	2,5	0,5
2009-07-04 09:33:18	60,2	2,5	0,5
2009-07-04 09:33:19	62,2	2	0
2009-07-04 09:33:21	62,6	0,4	-0,6
2009-07-04 09:33:23	63,1	0,5	-0,5
2009-07-04 09:33:25	62,6	-0,5	-1,5
2009-07-04 09:33:27	62,6	0	-1
2009-07-04 09:33:29	63,1	0,5	-0,5
2009-07-04 09:33:30	62,6	-0,5	-1,5
2009-07-04 09:33:32	61,4	-1,2	-2,2
2009-07-04 09:33:34	60,2	-1,2	-2,2

Tabella 6.2: Risultati

In Figura 6.12 si riporta graficamente l'andamento della crescita di temperatura insieme ai valori limite di PDH e PDL. Si può così notare i punti in cui il dato di temperatura non rimane all'interno dei due vincoli imposti. Tali punti rispecchiano quelli riportati nella tabella *Difference*.

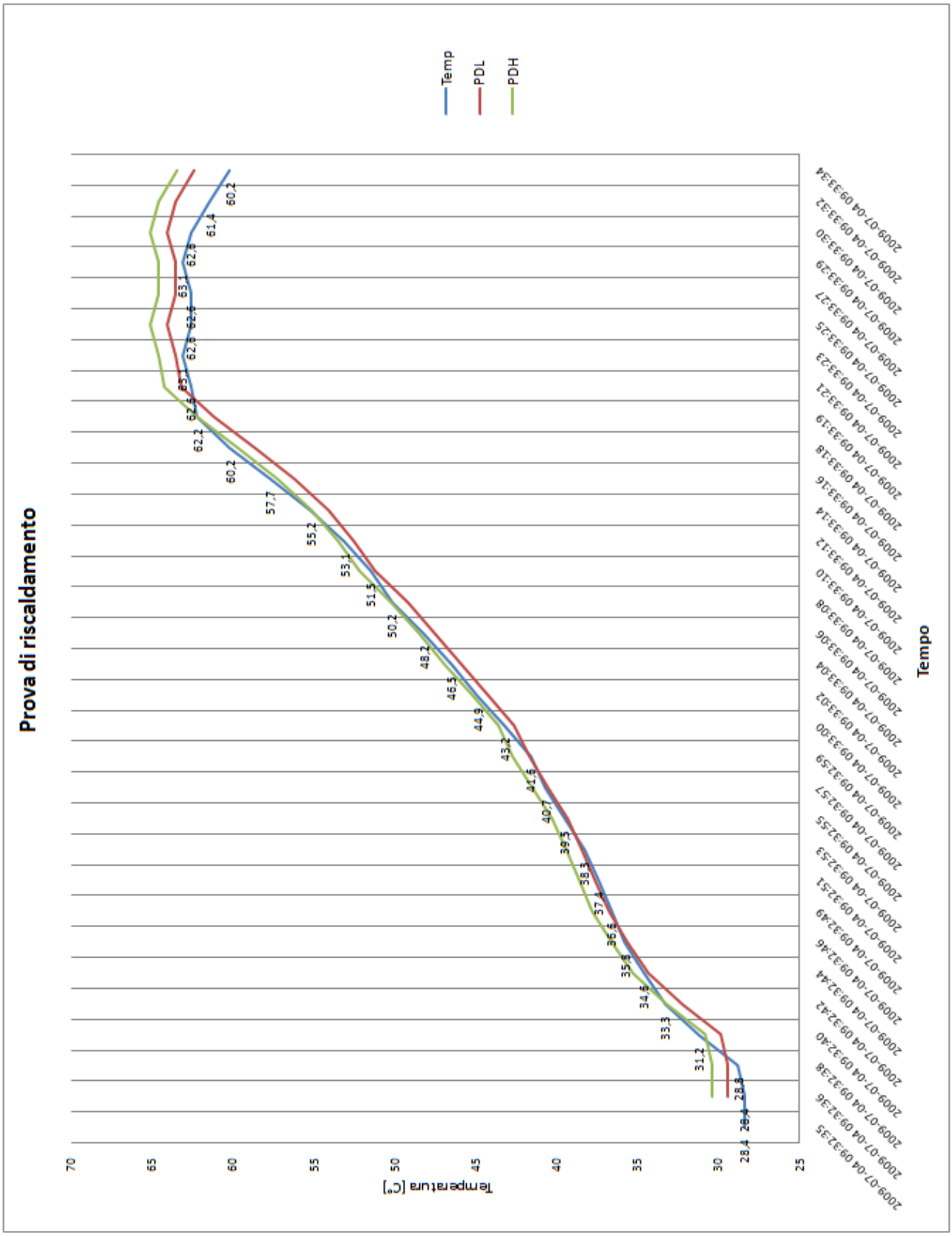


Figura 6.12: Grafico andamento temperatura con vincoli PDL e PDH

La Figura 6.13 seguente mostra l'andamento dei valori riportati nella tabella *Difference* relativi allo scostamento del delta T rispetto ai vincoli. Ad esempio alla terza rilevazione si è registrato un aumento di temperatura di 2.4 C° rispetto alla seconda rilevazione, ovvero 0,4 C° (valore riportato in grafico) superiore al vincolo imposto da PDH, mentre all'ottava misurazione si è registrato un delta T di 0.8 C°, ovvero -0,2 C° inferiori al vincolo PDL.

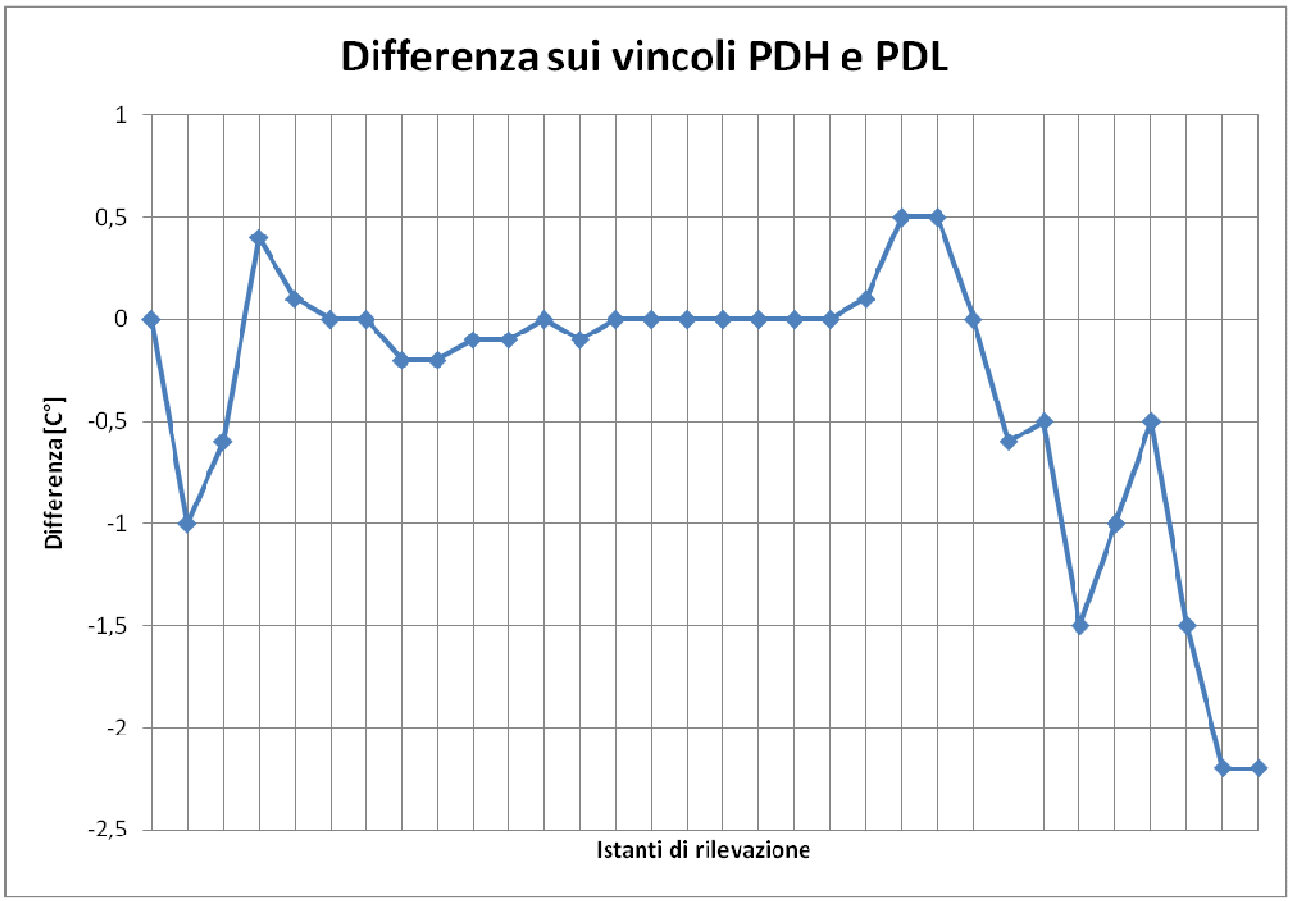


Figura 6.13: Grafico valori Difference

6.4 Controllo pendenza negativa nel raffreddamento

Analogamente alla prova di riscaldamento, in questo prova si è utilizzato un solo End Device configurato con una sola query attiva e vincoli di *NDH* ed *NDL* impostati. Questi misurano la differenza (negativa) tra due misurazioni successive. Nel test sono stati selezionati valori di NDH = -1.6 ed NDL = -1 gradi, così ad esempio, data un misurazione di 23,6 C°, la successiva, per non generare segnalazione sul mancato rispetto dei vincoli, può variare in un range compreso tra 22,6 – 22 C°. Un valore al di sopra o al di sotto di tale range, sarà segnalato, sulla tabella *Difference* remota, riportandone la differenza rispettivamente su NDL ed NDH.

6.4.1 Configurazione sistema

Per la prova si è utilizzato un solo End Device configurato a una frequenza di rilevamento di 5 secondi, con una sola query attiva avente come condizioni NDH = -1,7 e NDL = -1.

Il periodo di campionamento è stato di circa 2 minuti e 20 secondi per un totale di 29 misurazioni. La temperatura del sensore, raffreddato in un frigorifero, è passato dai 42,3 C° iniziali ai 14,6 C° finali.

6.4.2 Risultati

In Tabella 6.3 sono riportati i dati delle misurazioni, ordinati in successione cronologica. Nella terza colonna è riportato il calcolo del delta T, tra ogni valore e il precedente. Nella quarta invece la differenza del delta sulle condizioni impostate. Un quantità positiva indica il valore di superamento di NDL, una quantità negativa indica il valore al di sotto del vincolo inferiore di NDH.

TEMPO	T (C°)	Delta T	Differenza
2009-07-04 11:05:22	42,3	/	/
2009-07-04 11:05:26	41,5	-0,8	0,2
2009-07-04 11:05:31	39,8	-1,7	-0,1
2009-07-04 11:05:36	38,2	-1,6	0
2009-07-04 11:05:40	36,5	-1,7	-0,1
2009-07-04 11:05:45	34,9	-1,6	0
2009-07-04 11:05:50	32,8	-2,1	-0,5
2009-07-04 11:05:55	31,5	-1,3	0
2009-07-04 11:06:00	30,3	-1,2	0
2009-07-04 11:06:05	29,1	-1,2	0
2009-07-04 11:06:10	27,8	-1,3	0
2009-07-04 11:06:14	26,6	-1,2	0
2009-07-04 11:06:19	25,3	-1,3	0
2009-07-04 11:06:24	24,1	-1,2	0
2009-07-04 11:06:29	23,3	-0,8	0,2
2009-07-04 11:06:34	22,5	-0,8	0,2
2009-07-04 11:06:39	21,6	-0,9	0,1
2009-07-04 11:06:44	20,8	-0,8	0,2
2009-07-04 11:06:50	19,6	-1,2	0
2009-07-04 11:06:55	19,2	-0,4	0,6
2009-07-04 11:07:00	18,3	-0,9	0,1
2009-07-04 11:07:05	17,9	-0,4	0,6
2009-07-04 11:07:10	17,1	-0,8	0,2
2009-07-04 11:07:15	16,3	-0,8	0,2
2009-07-04 11:07:20	16,3	0	1
2009-07-04 11:07:25	15,8	-0,5	0,5
2009-07-04 11:07:30	15	-0,8	0,2
2009-07-04 11:07:35	14,6	-0,4	0,6
2009-07-04 11:07:41	14,6	0	1

Tabella 6.3: Risultati

In Figura 6.14 si riporta graficamente l'andamento di decrescita della temperatura unita ai valori limite di NDH ed NDL. Si possono notare i punti in cui le temperature esce dai vincoli imposti.

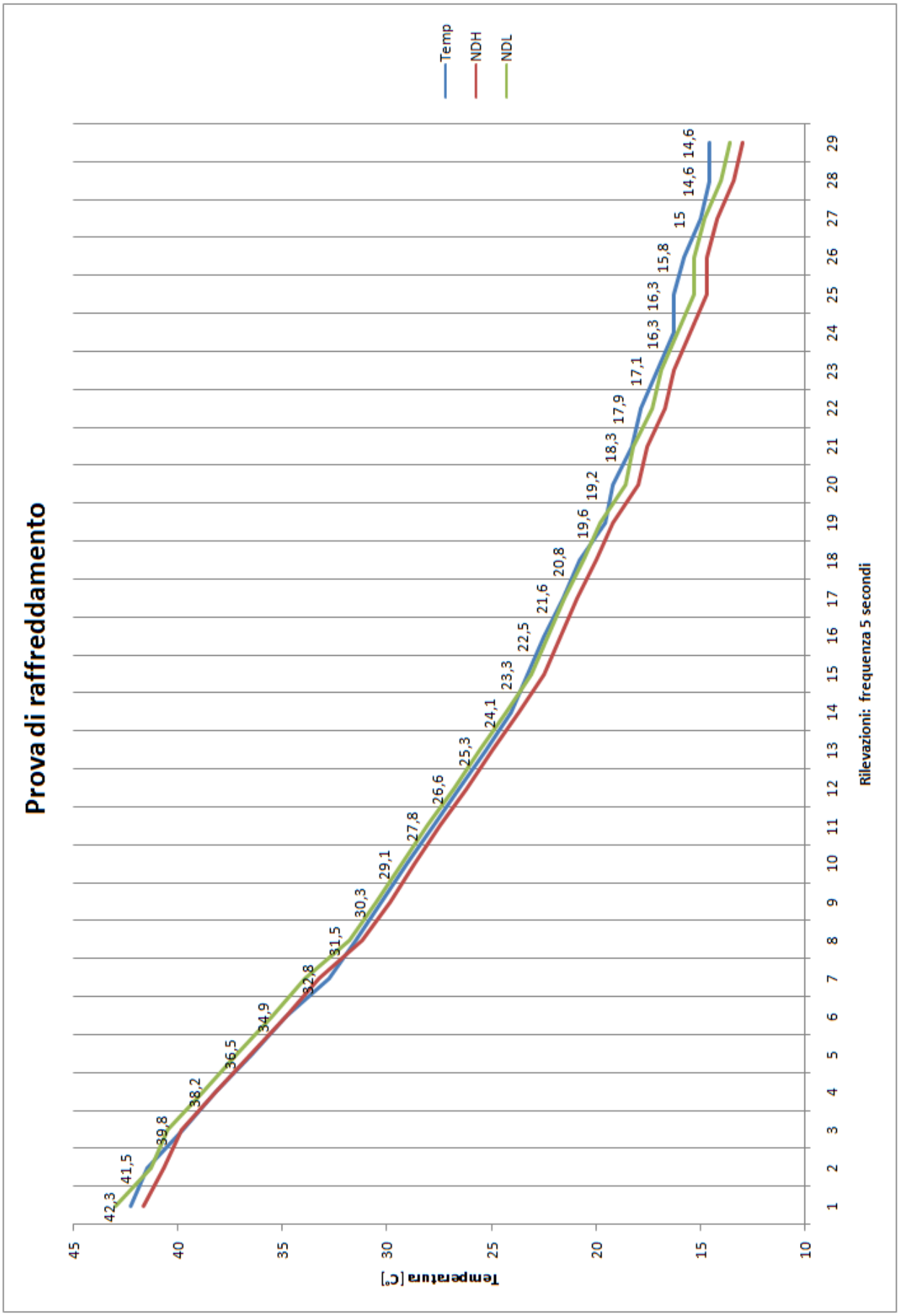


Figura 6.14: Grafico andamento temperatura con vincoli NDL e NDH

La Figura 6.15 seguente mostra l'andamento dei valori riportati nella tabella *Difference* relativi allo scostamento del delta T rispetto ai vincoli. Ad esempio alla settima rilevazione si è registrato una diminuzione della temperatura di -2,1 C° rispetto alla prima rilevazione, ovvero di -0,5 C° (valore riportato in grafico) al vincolo imposto da NDH (-1,6), mentre alla ventesima misurazione si è registrato un delta T di -0.4 C°, ovvero 0,6 C° superiore al vincolo NDL (-1).

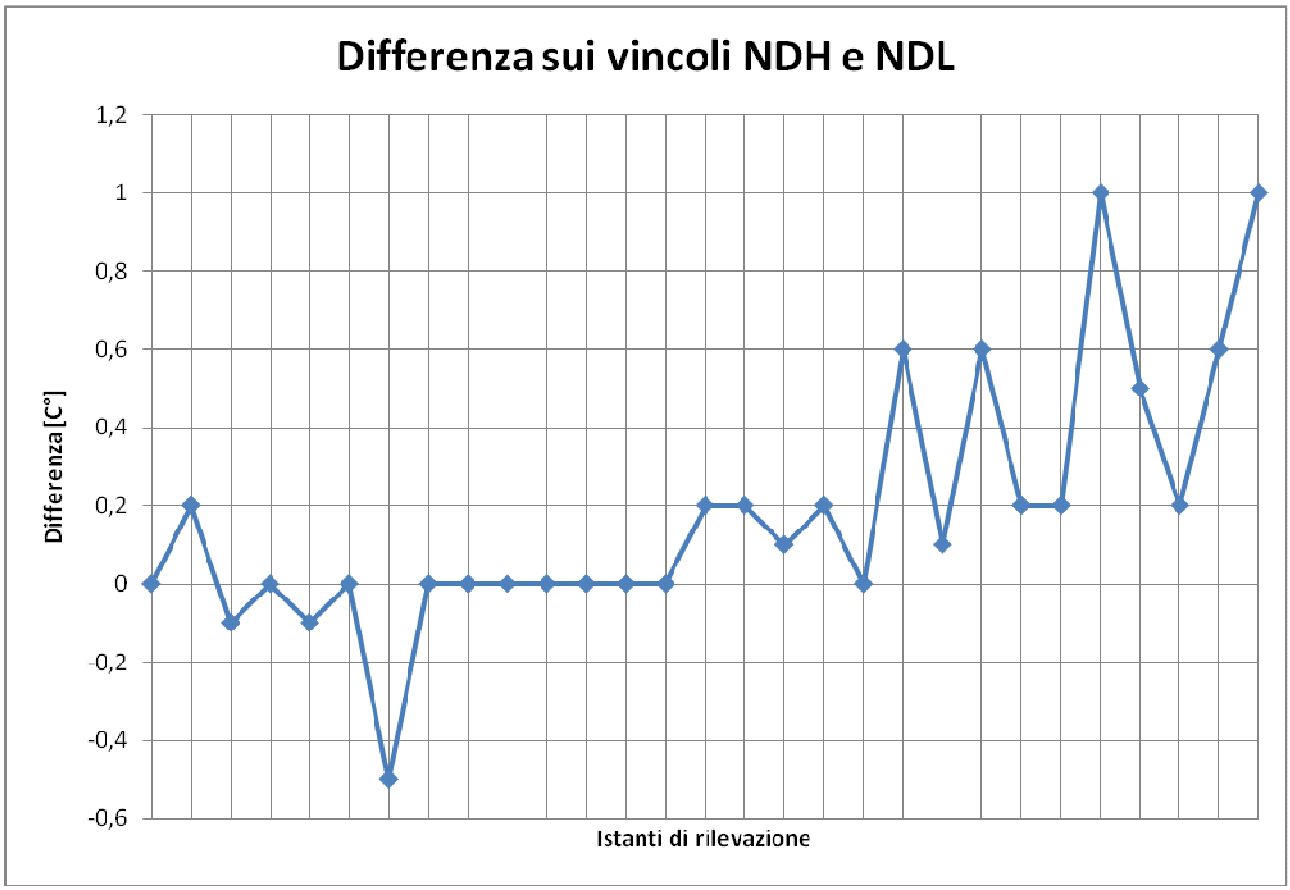


Figura 6.15: Grafico valori Difference

6.5 Query multiple con simulazione controllo Event Manager

Nella seguente prova si è simulata una delle azioni di controllo che il sistema centrale (*Query Manager*) e in particolare l'*Event Manager* può adottare nel monitoring delle rilevazioni.

Si è simulato il controllo di un ambiente riscaldato, impostando diverse "soglie" di allerta. Nel dettaglio sono state configurate tre differenti query ognuna specificante frequenze di rilevamento diverse e crescenti, esattamente come le soglie di temperatura massima nelle condizioni.

L'Event Manager potrebbe avere una configurazione tale per cui al raggiungimento delle determinate soglie, aumenta il livello di attenzione, ovvero abilita query a frequenza di rilevazione superiore. Nel sistema di prova il Query Manager e quindi anche l'Event Manager non sono stati sviluppati, se ne è quindi simulata l'azione controllando manualmente la tabella *Difference* ed all'arrivo della segnalazione di superamento della soglia si è proceduto all'invio manuale del comando per sostituire la query attiva.

6.5.1 Configurazione sistema

Per la prova si è utilizzato un solo End Device riscaldato all'interno di un fornetto elettrico; quindi si è monitorato il riscaldamento del dispositivo e raggiunta una temperatura prossima ai 60 C°, si è rimosso il nodo sensore dal forno e se ne è monitorato anche il raffreddamento.

Sono state impostate tre query, con tre frequenze diversa, ovvero ogni 10 sec, 5 sec, 2 sec (Figura 6.16).

IDQ	IDDEV	FREQ	ACTIVE
1	1	10	1
2	1	5	0
3	1	2	0

Figura 6.16: Query attive

La prima query attivata (IDQ=1), specificava una condizione sulla temperatura massima di 45 C°, raggiunta la quale il wrapper ne segnala al sistema centrale il superamento, con la scrittura nella tabella remota *Difference*. Si è proceduto quindi all'invio al wrapper del comando per disabilitare la query con IDQ=1 e inserire la nuova query con IDQ=2, specificante un periodo di rilevamento di 5 secondi. Al superamento della seconda soglia specificata dalla seconda query di 55 C°, si è disattivata la stessa e inserita la query (IDQ=3), con frequenza di rilevamento ogni 2 secondi. Si ricorda che i comandi inviati al wrapper sono costituiti da un inserimento di tuple nella tabella *Change_config*. Un esempio (superamento della prima soglia) è il seguente:

```
INSERT INTO change_config (idq,ins,del,act,disact)
VALUES (1,0,0,0,1),(2,1,0,0,0);
```

Durante il periodo di raffreddamento si è proceduto con l'operazione inversa, ovvero al cambio della query attiva per abbassare la frequenza di rilevazione.

6.5.2 Risultati

In Tabella 6.4 sono riportati i dati delle misurazioni, ordinati in successione cronologica. Nella prima colonna è specificata l'IDQ della query che ha generato il dato. Si noti la differenza tra le frequenze di campionamento delle diverse query.

IDQ	DATE	TEMP
1	2009-07-05 12:01:57	30,4
1	2009-07-05 12:02:09	30,8
1	2009-07-05 12:02:20	32,1
1	2009-07-05 12:02:31	35,0
1	2009-07-05 12:02:42	38,3
1	2009-07-05 12:02:53	42,4
1	2009-07-05 12:03:04	46,9
2	2009-07-05 12:03:17	54,0
2	2009-07-05 12:03:23	56,4
2	2009-07-05 12:03:29	60,2
2	2009-07-05 12:03:40	61,8
3	2009-07-05 12:04:32	59,7
3	2009-07-05 12:04:35	58,5
3	2009-07-05 12:04:37	56,4
3	2009-07-05 12:04:40	55,6
3	2009-07-05 12:04:43	55,6
3	2009-07-05 12:04:46	56,0
3	2009-07-05 12:04:49	56,0
3	2009-07-05 12:04:51	56,4

3	2009-07-05 12:04:54	56,4
3	2009-07-05 12:04:57	56,9
3	2009-07-05 12:05:00	57,3
3	2009-07-05 12:05:03	56,9
3	2009-07-05 12:05:05	56,9
3	2009-07-05 12:05:08	57,3
3	2009-07-05 12:05:11	57,3
3	2009-07-05 12:05:14	56,9
3	2009-07-05 12:05:17	56,0
3	2009-07-05 12:05:19	55,2
3	2009-07-05 12:05:22	54,4
3	2009-07-05 12:05:25	53,6
2	2009-07-05 12:05:31	52,3
2	2009-07-05 12:05:34	51,5
2	2009-07-05 12:05:36	50,7
2	2009-07-05 12:05:39	49,8
2	2009-07-05 12:05:42	49,0
2	2009-07-05 12:05:45	48,2
2	2009-07-05 12:05:51	47,8
2	2009-07-05 12:05:54	46,5
2	2009-07-05 12:06:00	44,9
2	2009-07-05 12:06:06	43,6
2	2009-07-05 12:06:11	42,4
2	2009-07-05 12:06:17	41,6
2	2009-07-05 12:06:23	40,3
1	2009-07-05 12:06:36	38,7
1	2009-07-05 12:06:47	37,4
1	2009-07-05 12:06:58	36,6
1	2009-07-05 12:07:10	35,8
1	2009-07-05 12:07:21	35,0
1	2009-07-05 12:07:32	34,6
1	2009-07-05 12:07:43	33,7

Tabella 6.4: Risultati

Il grafico dei risultati è rappresentato nella successiva Figura 6.16. Nell'asse delle ordinate sono riportati i valori di temperatura rilevati, mentre nell'asse delle ascisse è riportato il tempo in modo lineare, così da poter facilmente notare come i dati si infittiscano oltre le soglie di attenzione (sotto l'azione delle differenti query) e forniscano un campionamento più preciso del valore rilevato.

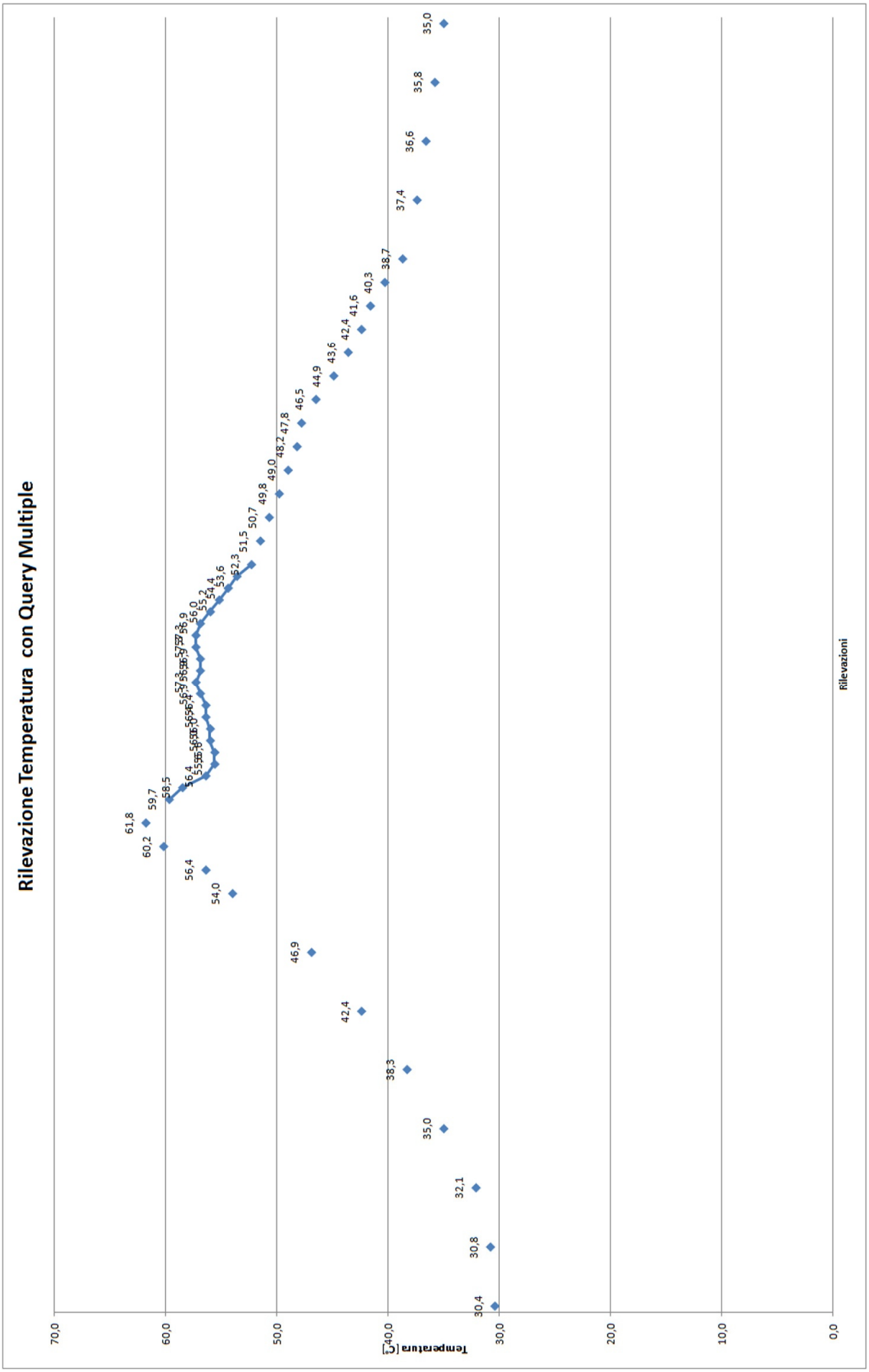


Figura 6.16: Grafico risultati

Capitolo 7

Conclusioni

Le rete di sensori wireless (WSN) possono essere collocate nel contesto del *pervasive computing (everywhere)*, dove l'elaborazione delle informazioni risulta integrata all'interno di oggetti e attività comuni, finora mai destinate a questo tipo di utilizzo. L'ambiente diviene così intelligente, capace di interagire anche in modo autonomo con l'uomo e con gli oggetti e parametri che lo compongono, permettendone un fine adattamento alle esigenze richieste. Settori come la domotica, le biotecnologie, l'ecologia e la prevenzione ambientale potrebbero risultare sempre maggiormente interessati a questo tipo di paradigma e l'utilizzo delle WSN potrà diventare il veicolo per realizzarlo.

L'ipotesi di un utilizzo maggiore delle reti di sensori necessità di modalità e tecnologie adeguate per l'interazione con esse. La mole di dati potenzialmente disponibile tramite le WSN può risultare di dimensioni notevoli, tale da risultare di difficile gestione. Da qui la necessità di utilizzo della tecnologia delle basi di dati e l'interazione con i Data Base Management System (DBMS), in grado di gestire in modo efficace i dati provenienti da tali contesti.

Inoltre l'eterogeneità delle diverse architetture hardware per le WSN e dei differenti protocolli e topologie di rete utilizzate, rendono necessario l'utilizzo di sistemi in grado di raccogliere, ed eventualmente aggregare, questi dati rendendoli disponibili alle applicazioni e all'utente tramite un comune sistema di interazione.

Il lavoro descritto in questa tesi è stato sviluppato cercando di raggiungere tali obbiettivi di aggregazione ed eterogeneità delle WSN. Obbiettivi che si è potuto raggiungere introducendo un livello software intermedio, ovvero il DBMS MySQL.

L'utilizzo di un sistema database sia per la raccolta dei dati che per la gestione e l'invio dei comandi verso la rete di sensori rende il sistema nel suo complesso molto flessibile.

Il sistema centrale e il suo modello di interrogazione e invio delle configurazione dei sensori basato su *Query* permette l'utilizzo di un "linguaggio" comune, indipendente dalle piattaforme hardware sottostanti. L'implementazione fisica delle WSN connesse conseguentemente non modifica la struttura del sistema centrale e del sistema di query.

E' compito della parte distribuita, ovvero degli Host Wrapper, realizzare l'effettiva connessione fisica e la comunicazione con la WSN. Risultando così il *Wrapper* l'unico componente software a necessitare di riprogrammazione in funzione della rete specifica rete WSN ad esso connessa. L'implementazione sviluppata in questa tesi del software wrapper per una rete di sensori basata sul dispositivo EZ430-RF2500, realizza le funzioni descritte dal modello generale.

La WSN implementata in maniera nativa col software standard del dispositivo della Texas Instruments non permetteva una comunicazione adeguata col software wrapper creato. Quindi è stato necessario modificare il firmware originario dei sensori al fine di permetterne il corretto funzionamento col sistema sviluppato. Si è dovuto modificare nei nodi delle rete il software, permettendone così la comunicazione bidirezionale con l'AP, il quale si limitava a ricevere i dati a frequenza costante dagli ED, fornendo in uscita sulla porta seriale delle stringhe testuali contenenti le informazioni. Lo stesso AP è stato modificato, rimuovendone le funzionalità di formattazione e creazione della stringa e consentendogli così un carico computazionale inferiore. In questa situazione è solo il software wrapper ad essere incaricato della totale gestione e manipolazione dei dati provenienti dalla rete.

E' stata modificata, al fine di rendere più flessibile la rilevazione della temperatura, la modalità di invio del dato. Si è optato per l'utilizzo di una variabile settabile in esecuzione su ogni nodo, tramite la ricezione del rispettivo messaggio, che specifica la frequenza di recupero ed invio della temperatura all'AP.

Il software e la rete WSN realizzata in questo lavoro, risulta limitata al recupero e alla gestione dell'unico dato fornito dal sensore del dispositivo, ovvero la temperatura. Il sistema rimane comunque valido per ogni tipologia di dato numerico, adeguando lo schema relazionale delle tabelle, con i campi e vincoli relativi ai dati forniti dalla rete. Anche i parametri di configurazione dei nodi sensori, nel nostro caso solo la frequenza di rilevazione della temperatura, possono essere adeguati alle diverse realizzazioni, modificando il software wrapper e le tabelle del DBMS. Le modalità di interrogazione, di segnalazione e comunicazione col sistema centrale non necessitano di sostanziali modifiche.

7.1 Sviluppi futuri

Alcuni limiti del sistema corrispondono a una mancanza di *dinamicità* e capacità di *autoconfigurazione* del sistema stesso. La rete di sensori necessita di una configurazione statica dei nodi come i moduli del sistema. I nodi sensore devono essere appositamente programmati ed associati, specificandolo nel codice, all'indirizzo

che li renderà identificabili in modo univoco al wrapper. La struttura e la dislocazione dei nodi deve essere conosciuta a priori. Non vi sono meccanismi di associazione automatica ed autoconfigurazione di nuovi nodi alla rete.

Per quanto riguarda la configurazione del sistema centrale e dell'Host Wrapper, anch'essa è definita staticamente. Una strada percorribile per migliorare questo aspetto, potrebbe essere l'introduzione di un apposito modulo, destinato a gestire i meccanismi per l'associazione di nuove WSN e quindi nuovi Host Wrapper al sistema e, rispetto ai nodi sensori, un apposito modulo all'interno del software wrapper che gestisca la dinamicità dei nodi stessi. Un modulo siffatto dovrebbe gestire la richiesta di un nuovo nodo di associazione alla rete, avere una "mappa" geografica dell'ambiente e ricevere le informazione sulla sua dislocazione fisica all'interno dello spazio da monitorare, gestendone l'assegnazione dell'indirizzo univoco. Il modulo dovrebbe inoltre esportare ai livelli superiori le modifiche della rete, implementando meccanismi che permettano al sistema centrale di rimanere aggiornato istante per istante sulla struttura della rete.

Nello specifico della WSN esaminata nel lavoro svolto, vi potrebbero essere miglioramenti sulla topologia della rete stessa. In questa tesi si è trattata una rete totalmente *a stella*, ma la soluzione migliore sotto i profili di dinamicità, estensione ed autonomia, sarebbe una rete di tipo *mesh*. Il protocollo di rete (gestione del routing) e l'hardware (poche risorse in termini di memoria e capacità computazionale) utilizzati non permettono facilmente di implementare questa tipologia di rete. Potrebbe risultare un'alternativa, lo sviluppo di un sistema di *routing* definibile come *asimettrico*, ovvero gestito in parte dal nodo e in parte dal software wrapper. La comunicazione dei nodi verso il wrapper avverrebbe in modo automatico, in quanto ogni nodo è a conoscenza del nodo a lui connesso facente parte del "percorso" determinato verso il centro stella (potenzialmente il messaggio può attraversare più nodi). Invece la comunicazione dal centro stella verso i nodi (oppure da nodo a nodo, passando però dal centro stella) verrebbe elaborata dal wrapper, che conoscendo la struttura della rete, calcola il percorso del messaggio, includendolo al suo interno (payload oppure header di un apposito layer). Ogni nodo che riceve il messaggio da inoltrare, legge l'indirizzo successivo e invia il messaggio al nodo specificato, fino al raggiungimento della destinazione.

Appendice A

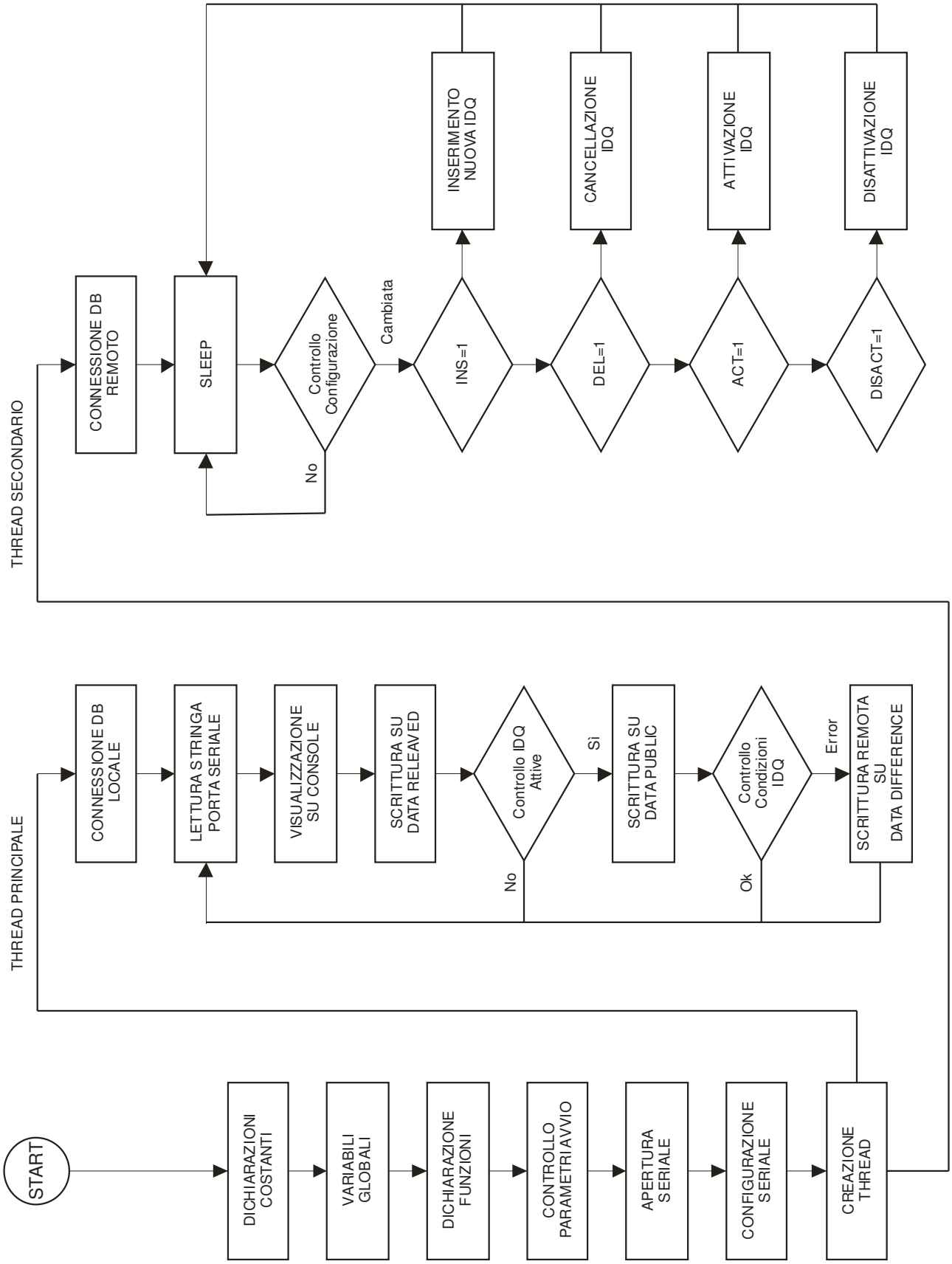
Il codice utilizzato

In questo appendice sono riportati i listati dei programmi creati e descritti in questo lavoro di tesi.

A.1 Wrapper C++

Nella sezione A.11 è riportato il flow-chart relativo al Wrapper C++ sviluppato, mentre nella sezione A.12 il codice del programma.

A.11 Flow-chart



A.12 Codice

```
#include <iostream>
#include <mysql++.h>
#include "windows.h"

//include di mysql++, struttura tabella DB in cui inserire i dati
#include "db_table.h"

//**COSTANTI**
//parametri connessione db
#define DB "wsn"
#define SERVER "localhost"
#define USER "root"
#define PASS "root"

//parametri connessione db remoto
#define DBR "wsn_remote"
#define SERVERR "localhost"
#define USERR "root"
#define PASSR "root"

//porta seriale
#define PORT "COM2"

//lunghezza stringa ricevuta
#define MESSAGE_LENGHT 9

//tempo msec, controllo nuove IDQ in change_config
#define TIME_CHECK 3000

//**DICHIARAZIONI FUNZIONI**

//THREAD PRINCIPALE
void outToConsole(byte stringInput[9]);
void outToDBDataRel (int addr, float temp, float batt, int sign);
void insToDBDataPub(int idq, int addr, float tempC);
void checkCondition(int idq, float last_temp, float tempC);
void insToDBDiff(int idq, float max, float min, float pdh, float pdl,
float ndh, float ndl);
//THREAD SECONDARIO
void insIdq (int idq);
void delIdq (int idq);
void actIdq (int idq);
void deactIdq (int idq);
void setFREQ (byte iddev, unsigned short int freq);



//**VARIABILI GLOBALI**

char verboseMode = 1; //stringa NoVerbose in console se settata a 0
char degCMode = 1; //gradi in C° di default
//SERIALE
HANDLE comPort;
```

```
DWORD bytesread, byteswrite;
BYTE charRead[1], charWrite[1];
//stringa lettura, include come ultimo carattere il "terminatore dati"
byte read[MESSAGE_LENGHT+1];
//CONNESSIONI DB
mysqlpp::Connection connMain(DB,SERVER,USER,PASS);
mysqlpp::Connection connThread(DB,SERVER,USER,PASS);
mysqlpp::Connection connRemote(DBR,SERVERR,USERR,PASSR);


using namespace std;

//thread per l'ascolto sulla porta seriale
DWORD threadRead() {
	std::cout << "Sono il thread in ascolto su COM, correttamente
avviato." << std::endl;

	//**CONTROLLO CONNESSIONE DB**
	try {
		if (!connThread.connected())
connThread.connect(DB,SERVER,USER,PASS);
	}
	catch (const mysqlpp::Exception& er) {
		// Catch for any MySQL++ exceptions
		cerr << "Errore MySQL++: " << er.what() << endl;
	}

	while (true) {
		//resetto il byte letto
		*charRead = '0';
		//inizializzo la stringa di lettura
		for (int j=0; j<MESSAGE_LENGHT+1; j++){
			read[j] = 'z';
		}
		//readfile rimane in attesa di attesa di dati, fino allo
scadere del TimeOut settato
		//in attesa di ricezione del carattere inizio stringa
		while (charRead[0] != 0x02) ReadFile(comPort, charRead, 1,
&bytesread, NULL );
		//ricevuto l'inizio stringa, inizia la ricezione dei dati
		for (int i=0; i<MESSAGE_LENGHT+1; i++) {
			//si suppone che i dati una volta avviato il
trasferimento, arrivino prima di ogni TimeOut
			ReadFile(comPort, charRead, 1, &bytesread, NULL );
			read[i] = (char)(*charRead);
		}
		//controllo la correttezza della stringa ricevuta, verificando
la presenza del terminatore di stringa
		if (read[MESSAGE_LENGHT] == 0x03){
			//**OUTPUT IN CONSOLE**
			outToConsole(read);

			//**PARSING DELLA STRINGA NEI DATI NUMERICI**
			//dati acquisiti
			char addrC[] = ("zzz");
			int addr = 0;
```

```
                    float tempC = 0;
                    float battV = 0;
                    int strenght = 0;

                    //stringa indirizzo
                    addrC[0] = read[0];
                    addrC[1] = read[1];
                    addrC[2] = read[2];
                    addr = atoi(addrC);

                    //valore temperatura sempre in C°
                    char temp_string[] = {" XX.X"};
                    int temp = read[6] + (read[7]<<8);
                    if( temp < 0 ) {
                         temp_string[0] = '-';
                         temp = temp * -1;
                    }
                    else if( ((temp/1000)%10) != 0 ) {
                         temp_string[0] = '0'+((temp/1000)%10);
                    }
                    temp_string[4] = '0'+(temp%10);
                    temp_string[2] = '0'+((temp/10)%10);
                    temp_string[1] = '0'+((temp/100)%10);

                    //conversione in float
                    tempC = atof(temp_string);

                    //valore batteria
                    char batt[] = "X.X";
                    batt[0] = '0'+(read[8]/10)%10;
                    batt[2] = '0'+(read[8]%10);
                    battV = atof(batt); //conversione in float

                    //potenza segnale in percentuale
                    char sign[] = "XXX";
                    sign[0] = read[5];
                    sign[1] = read[4];
                    sign[2] = read[3];
                    strenght = atoi(sign); //conversione in int

                    //**OUTPUT IN DATA_RELEAVED**
                    outToDBDataRel(addr,tempC,battV,strenght);

                    //**Recupero IDQ associate al DEVICE e la LAST_TEMP**
                    mysqlpp::StoreQueryResult resIDQLT;
                    try {
                         string s = "select c.idq, d.last_temp from
configuration c, device d where c.iddev = d.iddev and d.iddev = '";
                         char buff[32];
                         itoa(addr,buff,10);
                         s.append(buff);
                         s.append("' and active=1");
                         mysqlpp::Query query = connThread.query(s);
                         resIDQLT = query.store();

                    }
```

```
                catch (const mysqlpp::Exception& er) {
                      // Catch for any MySQL++ exceptions
                      cerr << "Errore MySQL++: " << er.what() << endl;
                }

                //per ogni IDQ attiva, controllo le condition e
inserisco i valori in DATA_PUBLIC
                for (unsigned int k=0; k < resIDQLT.num_rows(); k++) {
                      //lancio la funzione per l'inserimento della
rilevazione in DATA_PUBLIC
                      insToDBDataPub(resIDQLT[k][0],addr,tempC);
                      //lancio la funzione per il controllo del rispetto
delle condizioni, solo se NON è la prima rilevazione
                      if (resIDQLT[k][1] != mysqlpp::null)
checkCondition(resIDQLT[k][0],resIDQLT[k][1],tempC);
                }

                //aggiorno nella tabella device, il valore di last_temp
                try {
                      string s1 = "UPDATE device SET last_temp='";
                      s1.append(temp_string);
                      s1.append("' where iddev='");
                      char buff[32];
                      itoa(addr,buff,10);
                      s1.append(buff);
                      s1.append("'");
                      mysqlpp::Query query = connThread.query(s1);
                      query.exec();
                }
                catch (const mysqlpp::Exception& er) {
                      // Catch for any MySQL++ exceptions
                      cerr << "Errore MySQL++: " << er.what() << endl;
                }

          }
     }
}

int main(int argc, char* argv[]) {

     //**CONTROLLO PARAMETRI AVVIO**
     //visualizzazione in console C/F e verbose/noverbose
     if (argc == 2) { // con un solo parametro
          if (!strcmp(argv[1],"f") || !strcmp(argv[1],"F")) degCMode =
0;
     }
     if (argc == 3) { // con 2 parametri
          if (!strcmp(argv[1],"f") || !strcmp(argv[1],"F")) degCMode =
0;
          if (!strcmp(argv[2],"noverbose")) verboseMode = 0;
     }

     //** APERTURA PORTA SERIALE **
     comPort = CreateFile(PORT,GENERIC_READ |
GENERIC_WRITE,0,0,OPEN_EXISTING,0,0);
    if (comPort == INVALID_HANDLE_VALUE) {
```

```
        cout << ("Errore apertura porta ") << PORT << ("--> USCITA\n");
        //error opening port exit
           return -1;
    }
    cout << ("COM Aperta\n\n");
     //instanzio oggetto COMMTIMEOUTS, per definire i timeout della COM
e rendere
     //ReadFile() non bloccante, condivisione tra i thread della com
     COMMTIMEOUTS comTimeOut;
     //Specify time-out between charactor for receiving.
     comTimeOut.ReadIntervalTimeout = 0,01;
     //Specify value that is multiplied by the requested number of bytes
to be read
     comTimeOut.ReadTotalTimeoutMultiplier = 1;
     //Specify value is added to the product of the
ReadTotalTimeoutMultiplier member
     comTimeOut.ReadTotalTimeoutConstant = 300;
     //Specify value that is multiplied by the requested number of bytes
to be sent
     comTimeOut.WriteTotalTimeoutMultiplier = 1;
     //Specify value is added to the product of the
WriteTotalTimeoutMultiplier member
     comTimeOut.WriteTotalTimeoutConstant = 1000;
     //set the time-out parameter into device control.
     SetCommTimeouts(comPort,&comTimeOut);

     //**CONTROLLO CONNESSIONE DB**
     try {
          if (!connMain.connected())
connMain.connect(DB,SERVER,USER,PASS);
     }
     catch (const mysqlpp::Exception& er) {
          // Catch for any MySQL++ exceptions
          cerr << "Errore MySQL++: " << er.what() << endl;
     }

     //**CREAZIONE E AVVIO THREAD IN ASCOLTO SU COM**
     DWORD dwThreadId, dwThrdParam = 1;
     HANDLE thread =  CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
threadRead, &dwThrdParam, 0, &dwThreadId);

     //CICLO MAIN
     while (true) {
          Sleep(TIME_CHECK);
          mysqlpp::StoreQueryResult resNEWCONF;
          try {
               string s = "select * from change_config";
               mysqlpp::Query query = connMain.query(s);
               resNEWCONF = query.store();
          }
          catch (const mysqlpp::Exception& er) {
               // Catch for any MySQL++ exceptions
               cerr << "Errore MySQL++: " << er.what() << endl;
          }
          //se ho nuove idq da processare
          if (resNEWCONF.num_rows() != 0) {
```

```
                for (unsigned int i=0; i<resNEWCONF.num_rows(); i++) {
                        //se è una nuova idq da inserire
                        if (resNEWCONF[i][1]) insIdq(resNEWCONF[i][0]);
                        //se è una idq da attivare
                        if (resNEWCONF[i][3]) actIdq(resNEWCONF[i][0]);
                        //se è una idq da disattivare
                        if (resNEWCONF[i][4]) deactIdq(resNEWCONF[i][0]);
                        //se l'idq è da cancellare
                        if (resNEWCONF[i][2]) delIdq(resNEWCONF[i][0]);
                }
        }
        //processate le righe posso cancellarle
        string s1;
        try {
                for (unsigned int j=0;j<resNEWCONF.num_rows(); j++) {
                        s1 = "delete from change_config where idq = '";
                        s1.append(resNEWCONF[j][0]);
                        s1.append("'");
                        mysqlpp::Query query2 = connMain.query(s1);
                        query2.exec();
                }
        }
        catch (const mysqlpp::Exception& er) {
                // Catch for any MySQL++ exceptions
                cerr << "Errore MySQL++: " << er.what() << endl;
        }

    }
}

//funzioni thread
void outToConsole(byte stringInput[MESSAGE_LENGHT+1]) {
    //costruzione parte relativa alla temperatura
    char temp_string[] = {" XX.XC"};
    int temp = stringInput[6] + (stringInput[7]<<8);

    if( !degCMode ) { //se out in F
        temp = (((float)temp)*1.8)+320;
        temp_string[5] = 'F';
    }
    if( temp < 0 ) {
        temp_string[0] = '-';
        temp = temp * -1;
    }
    else if( ((temp/1000)%10) != 0 ) {
        temp_string[0] = '0'+((temp/1000)%10);
    }
    temp_string[4] = '0'+(temp%10);
    temp_string[2] = '0'+((temp/10)%10);
    temp_string[1] = '0'+((temp/100)%10);

  if( verboseMode ) {
    char output_verbose[] = {"\r\nNode:XXX,Temp:-
XX.XC,Battery:X.XV,Strength:XXX%,RE:no "};

    output_verbose[45] = stringInput[5];
```

```
    output_verbose[46] = stringInput[4];
    output_verbose[47] = stringInput[3];

    output_verbose[16] = temp_string[0];
    output_verbose[17] = temp_string[1];
    output_verbose[18] = temp_string[2];
    output_verbose[19] = temp_string[3];
    output_verbose[20] = temp_string[4];
    output_verbose[21] = temp_string[5];

    output_verbose[31] = '0'+(stringInput[8]/10)%10;
    output_verbose[33] = '0'+(stringInput[8]%10);
    output_verbose[7] = stringInput[0];
    output_verbose[8] = stringInput[1];
    output_verbose[9] = stringInput[2];

     //output in console
     cout << output_verbose << endl;
  }
  else {
    char output_short[] = {"\r\n$ADDR,-XX.XC,V.C,RSI,N#"};

    output_short[18] = stringInput[5];
    output_short[19] = stringInput[4];
    output_short[20] = stringInput[3];


    output_short[7] = temp_string[0];
    output_short[8] = temp_string[1];
    output_short[9] = temp_string[2];
    output_short[10] = temp_string[3];
    output_short[11] = temp_string[4];
    output_short[12] = temp_string[5];

    output_short[14] = '0'+(stringInput[8]/10)%10;
    output_short[16] = '0'+(stringInput[8]%10);
    output_short[3] = stringInput[0];
    output_short[4] = stringInput[1];
    output_short[5] = stringInput[2];
    //output in console
     cout << output_short << endl;
  }
}


void outToDBDataRel (int addr, float temp, float batt, int sign) {
     try {
          //creazione della query
          mysqlpp::Query query = connThread.query();

          dataReleaved row(addr, temp, batt, sign);
          query.insert(row);
          query.exec(); //uso exec() in quanto non serve alcun parametro
di ritorno
     }
     catch (const mysqlpp::Exception& er) {
```

```
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }
}


void insToDBDataPub(int idq, int addr, float tempC){
      try {
            //creazione della query
            mysqlpp::Query query = connThread.query();

            dataPublic row(idq, addr, mysqlpp::DateTime(), tempC);
            query.insert(row);
            query.exec(); //uso exec() in quanto non serve alcun parametro
di ritorno
      }
      catch (const mysqlpp::Exception& er) {
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }
}
void checkCondition(int idq, float last_temp, float tempC){
      //recupero le condizioni da verificare dell' IDQ
      mysqlpp::StoreQueryResult resCOND;
      try {
            string s = "select * from conditions where idq = '";
            char buff[32];
            itoa(idq,buff,10);
            s.append(buff);
            s.append("'");
            mysqlpp::Query query = connThread.query(s);
            resCOND = query.store();
      }
      catch (const mysqlpp::Exception& er) {
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }
      //inizializzo le variabili condizione con gestione dei "null"
      float maxC;
      if (resCOND[0][1] == mysqlpp::null) maxC = 0;
      else maxC = resCOND[0][1];
      float minC;
      if (resCOND[0][2] == mysqlpp::null) minC = 0;
      else minC = resCOND[0][2];
      float pdhC;
      if (resCOND[0][3] == mysqlpp::null) pdhC = 0;
      else pdhC = resCOND[0][3];
      float pdlC;
      if (resCOND[0][4] == mysqlpp::null) pdlC = 0;
      else pdlC = resCOND[0][4];
      float ndhC;
      if (resCOND[0][5] == mysqlpp::null) ndhC = 0;
      else ndhC = resCOND[0][5];
      float ndlC;
      if (resCOND[0][6] == mysqlpp::null) ndlC = 0;
      else ndlC = resCOND[0][6];
```

```
	//inizializzo le variabili differenza
	float max = 0;
	float min = 0;
	float pdh = 0;
	float pdl = 0;
	float ndh = 0;
	float ndl = 0;
	//check dei valori min e max
	if ((maxC != 0) && (tempC > maxC)) max = tempC - maxC;
	if ((minC != 0) && (tempC < minC)) min = tempC - minC;
	//calcolo il delta tra l'attuale valore e il precedente
	float d = tempC - last_temp;
	cout << "delta " <<d <<endl;
	//check dei delta, controllo separatamente la gestione crescente o
decrescente
	cout << "pdhC " <<pdhC <<endl;
	cout << "pdlC " <<pdlC <<endl;
	if (d > 0) {
		if ((pdhC !=0) && (d > pdhC)) pdh = d - pdhC;
	}
	if ((pdlC !=0) && (d < pdlC)) pdl = d - pdlC;
	cout << "pdh " <<pdh <<endl;
	cout << "pdl " <<pdl <<endl;

	cout << "ndhC " <<ndhC <<endl;
	cout << "ndlC " <<ndlC <<endl;
	if (d < 0) {
		if ((ndhC !=0) && (d < ndhC)) ndh = d - ndhC;
	}
	if ((ndlC !=0) && (d > ndlC)) ndl = d - ndlC;
	cout << "ndh " <<ndh <<endl;
	cout << "ndl " <<ndl <<endl;

	//si controlla ora se qualche parametro non rispetta le condizioni
(!=0), nel caso si inserisce i valori nella tabella remota "difference"
	if ((max != 0) || (min != 0) || (pdh != 0) || (pdl !=0) || (ndh
!=0) || (ndl !=0)) insToDBDiff(idq, max, min, pdh, pdl, ndh, ndl);

	//se le condizioni sono tutte rispettate non c'è alcuna azione
}
void insToDBDiff(int idq, float max, float min, float pdh, float pdl,
float ndh, float ndl){
	try {
		//creazione query
		mysqlpp::Query query = connRemote.query();
		// inserimento nella tabella "difference", gestione dei null
		difference
row(idq,mysqlpp::DateTime(),mysqlpp::null,mysqlpp::null,mysqlpp::null,mys
qlpp::null,mysqlpp::null,mysqlpp::null);
		if (max != 0) row.max = max;
		if (min != 0) row.min = min;
		if (pdh != 0) row.pdh = pdh;
		if (pdl != 0) row.pdl = pdl;
		if (ndh != 0) row.ndh = ndh;
		if (ndl != 0) row.ndl= ndl;
```

```
            query.insert(row);
            query.exec(); //uso exec() in quanto non serve alcun parametro
di ritorno
      }
      catch (const mysqlpp::Exception& er) {
            // Catch-all for any other MySQL++ exceptions
            cerr << "Error DB REMOTE: " << er.what() << endl;
      }
}
//funzioni main
void insIdq (int idq) {
      //aggiorno il flag active a 1 per la rispettiva idq
      try {
            string s = "UPDATE configuration SET active=1 WHERE idq='";
            char buff[32];
            itoa(idq,buff,10);
            s.append(buff);
            s.append("'");
            mysqlpp::Query query = connMain.query(s);
            cout << query.exec() << endl;
      }
      catch (const mysqlpp::Exception& er) {
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }

      //creo il result set con i device associati alla stessa idq
      //per effettuare il controllo e l'eventuale invio della nuova
frequenza
      mysqlpp::StoreQueryResult resDEVFREQ;
      try {
            string s = "SELECT c.iddev, c.freq, d.freq_setted FROM
configuration c, device d WHERE c.iddev = d.iddev AND c.idq = '";
            char buff[32];
            itoa(idq,buff,10);
            s.append(buff);
            s.append("'");
            mysqlpp::Query query = connMain.query(s);
            resDEVFREQ = query.store();
      }
      catch (const mysqlpp::Exception& er) {
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }

      //controllo frequenza settata se NULL (non c'è alcuna rilevazione
periodica attiva)
      //oppure se la nuova frequenza da settare è inferiore all'attuale
per ogni device associato

      for (unsigned int i=0;i<resDEVFREQ.num_rows();i++) {
            byte iddev = resDEVFREQ[i][0];
            unsigned short int freq = resDEVFREQ[i][1];
            unsigned short int freq_setted;
            if (resDEVFREQ[i][2] == mysqlpp::null) setFREQ(iddev, freq);
            else freq_setted = resDEVFREQ[i][2];
```

```
            if ((freq <= freq_setted)) setFREQ(iddev, freq);
      }

      //in caso contrario se le frequenza di rilevazione è già settata
inferiore, non si esegue nulla
}
void delIdq (int idq) {
      //si recupera la lista dei device associati all' idq da eliminare,
ma che NON hanno altre idq attive!
      mysqlpp::StoreQueryResult resDEVDEL;
      try {
            string s = "SELECT iddev FROM configuration WHERE idq ='";
            char buff[32];
            itoa(idq,buff,10);
            s.append(buff);
            s.append("' AND iddev NOT IN (SELECT iddev FROM configuration
WHERE idq !='");
            s.append(buff);
            s.append("')");
            mysqlpp::Query query = connMain.query(s);
            resDEVDEL = query.store();
      }
      catch (const mysqlpp::Exception& er) {
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }

      //vado a settare sui device la frequenza a 0 per terminare il
rilevamento del dato
      for (unsigned int i=0; i<resDEVDEL.num_rows();i++) {
            setFREQ(resDEVDEL[i][0],0);
      }

      //cancello le righe associate alla idq dalla tabella conditions e
configuration
      try {
            string s1 = "DELETE FROM conditions WHERE idq='";
            char buff1[32];
            itoa(idq,buff1,10);
            s1.append(buff1);
            s1.append("'");
            mysqlpp::Query query1 = connMain.query(s1);
            query1.exec();
            string s2 = "DELETE FROM configuration WHERE idq='";
            s2.append(buff1);
            s2.append("'");
            mysqlpp::Query query2 = connMain.query(s2);
            query2.exec();
      }
      catch (const mysqlpp::Exception& er) {
            // Catch for any MySQL++ exceptions
            cerr << "Errore MySQL++: " << er.what() << endl;
      }
}
void setFREQ (byte iddev, unsigned short int freq) {
      //scomposizione dei 16bit dell'intero frequenze in 2 byte
```

```
	byte freq_LB = freq & 255;
	byte freq_HB = freq >> 8;
	//invio i 3B di config
	for (int i=0;i<3;i++) {
		if (i==0) charWrite[0] = iddev;
		if (i==1) charWrite[0] = freq_LB;
		if (i==2) charWrite[0] = freq_HB;
		WriteFile(comPort, charWrite, 1, &byteswrite, NULL );
		cout << "inviato " << (int)charWrite[0] << endl;
	}
	cout << "Inviata configurazione: " << (int)iddev << " " <<
(int)freq_LB << " " <<(int)freq_HB << endl;

	//aggiorno la freq_setted per il device
	try {
		string s = "UPDATE device SET freq_setted=";
		if (freq==0) s.append(" null ");
		else {
			s.append(" '");
			char buff[32];
			itoa(freq,buff,10);
			s.append(buff);
			s.append("' ");
		}
		s.append("where iddev='");
		char buff2[32];
		itoa(iddev,buff2,10);
		s.append(buff2);
		s.append("'");
		cout << s << endl;
		mysqlpp::Query query = connMain.query(s);
		cout << query.exec() << endl;
	}
	catch (const mysqlpp::Exception& er) {
		// Catch for any MySQL++ exceptions
		cerr << "Errore MySQL++: " << er.what() << endl;
	}

}
void actIdq (int idq) {
	//aggiorno il flag active a 1 per la rispettiva idq
	try {
		string s = "UPDATE configuration SET active=1 WHERE idq='";
		char buff[32];
		itoa(idq,buff,10);
		s.append(buff);
		s.append("'");
		mysqlpp::Query query = connMain.query(s);
		cout << query.exec() << endl;
	}
	catch (const mysqlpp::Exception& er) {
		// Catch for any MySQL++ exceptions
		cerr << "Errore MySQL++: " << er.what() << endl;
	}
}
void deactIdq (int idq) {
```

```
	//aggiorno il flag active a 0 per la rispettiva idq
	try {
		string s = "UPDATE configuration SET active=0 WHERE idq='";
		char buff[32];
		itoa(idq,buff,10);
		s.append(buff);
		s.append("'");
		mysqlpp::Query query = connMain.query(s);
		cout << query.exec() << endl;
	}
	catch (const mysqlpp::Exception& er) {
		// Catch for any MySQL++ exceptions
		cerr << "Errore MySQL++: " << er.what() << endl;
	}
}
```

A.13 Le strutture dati SSQLS: db_table.h

```
#include <mysql++.h>
#include <ssqls.h>

sql_create_4 (dataReleaved, 1, 4,
                  mysqlpp::sql_int_unsigned, address,
                  mysqlpp::sql_float, temp,
                  mysqlpp::sql_float, battery,
                  mysqlpp::sql_int_unsigned, signal)

sql_create_4 (dataPublic, 1, 4,
                  mysqlpp::sql_int_unsigned, idq,
                  mysqlpp::sql_int_unsigned, iddev,
                  mysqlpp::sql_datetime, date,
                  mysqlpp::sql_float, temp)

sql_create_3 (device, 1, 3,
                  mysqlpp::sql_int_unsigned, iddev,
                  mysqlpp::sql_int_unsigned, freq_setted, //frequenze
effettivamente settata sul device
                  mysqlpp::Null<mysqlpp::sql_float>, last_temp)
//accetta null

sql_create_4 (configuration, 1, 4,
                  mysqlpp::sql_int_unsigned, idq,
                  mysqlpp::sql_int_unsigned, iddev,
                  mysqlpp::sql_int_unsigned, freq,
                  mysqlpp::sql_boolean, active)

sql_create_7 (condition, 1, 7,
                  mysqlpp::sql_int, idq,
                  mysqlpp::sql_float, max,
                  mysqlpp::sql_float, min,
                  mysqlpp::sql_float, pdh,
                  mysqlpp::sql_float, pdl,
                  mysqlpp::sql_float, ndh,
```

```
                    mysqlpp::sql_float, ndl)

sql_create_8 (difference, 1, 8, //tabella db remote, invio differenze
sulle condizioni
                    mysqlpp::sql_int, idq,
                    mysqlpp::sql_datetime, date,
                    mysqlpp::Null<mysqlpp::sql_float>, max,
                    mysqlpp::Null<mysqlpp::sql_float>, min,
                    mysqlpp::Null<mysqlpp::sql_float>, pdh,
                    mysqlpp::Null<mysqlpp::sql_float>, pdl,
                    mysqlpp::Null<mysqlpp::sql_float>, ndh,
                    mysqlpp::Null<mysqlpp::sql_float>, ndl)

sql_create_2 (change_config, 1, 2,
                    mysqlpp::sql_int, idq,
                    mysqlpp::sql_boolean, ins,
                    mysqlpp::sql_boolean, del,
                    mysqlpp::sql_boolean, act,
                    mysqlpp::sql_boolean, disact)
```

A.2 Access Point

```
#include "bsp.h"
#include "mrfi.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "nwk_frame.h"
#include "nwk.h"

#include "msp430x22x4.h"
#include "vlo_rand.h"

#define MESSAGE_LENGTH 3
//lunghezza totale stringa via com, aggiungo 2 byte di controllo per far
riconoscere i dati
#define STRING_LENGTH 1+3+3+MESSAGE_LENGTH+1
void TXString( char* string, int length );
void MCU_Init(void);
void transmitData(int addr, signed char rssi, char msg[MESSAGE_LENGTH] );

//ridotto a 1Byte (3 char) il campo address
void transmitDataString(char addr[3],char rssi[3], char
msg[MESSAGE_LENGTH]);
void createRandomAddress();

//nuove funzioni
void numLed (uint8_t num);
void trasmitToED (uint8_t addr);

//data for terminal output
const char splash[] = {"\r\n-------------------------------------------
-----\r\n     ****\r\n     ****           eZ430-RF2500\r\n
******o****    Temperature Sensor Network\r\n*******_///_****
Copyright 2007\r\n ******/_//_/*****  Texas Instruments Incorporated\r\n
** ***(__/*****   All rights reserved.\r\n      *********     Version
1.02\r\n       *****\r\n        ***\r\n-----------------------------------
----------------\r\n"};

__no_init volatile int tempOffset @ 0x10F4; // Temperature offset set at
production
__no_init volatile char Flash_Addr[4] @ 0x10F0; // Flash address set
randomly

// reserve space for the maximum possible peer Link IDs
static linkID_t sLID[NUM_CONNECTIONS];
static uint8_t  sNumCurrentPeers;

// callback handler
static uint8_t sCB(linkID_t);

// work loop semaphores
static uint8_t sPeerFrameSem =0;
```

```
static uint8_t sJoinSem = 0;
static uint8_t sSelfMeasureSem = 0;
static uint8_t sReceiveMsg = 0;

//dati ricevuti
static uint8_t addrNode[1];
static uint8_t recMsg[3];

// E' stata rimossa la funzionalità, delegata al wrapper la
visualizzazione in console
// mode data verbose = default, deg F = default
//char verboseMode = 1;
//char degCMode = 0;

//Variabili contatori per il timer di rilevazione
uint16_t timer = 0; //Fino a 65535 sec circa 18 ore
uint16_t count = 0;

void main (void)
{
  addr_t lAddr;
  bspIState_t intState;

  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  {
  // delay loop to ensure proper startup before SimpliciTI increases DCO
  // This is typically tailored to the power supply used, and in this
case
  // is overkill for safety due to wide distribution.
    volatile int i;
    for(i = 0; i < 0xFFFF; i++){}
  }
  if( CALBC1_8MHZ == 0xFF )                 // Do not run if cal values
are erased
  {
    volatile int i;
    P1DIR |= 0x03;
    BSP_TURN_ON_LED1();
    BSP_TURN_OFF_LED2();
    while(1)
    {
      for(i = 0; i < 0x5FFF; i++){}
      BSP_TOGGLE_LED2();
      BSP_TOGGLE_LED1();
    }
  }

  BSP_Init();

  if( Flash_Addr[0] == 0xFF &&
      Flash_Addr[1] == 0xFF &&
      Flash_Addr[2] == 0xFF &&
      Flash_Addr[3] == 0xFF )
  {
    createRandomAddress();                  // set Random device address
at initial startup
```

```
  }
  lAddr.addr[0]=Flash_Addr[0];
  lAddr.addr[1]=Flash_Addr[1];
  lAddr.addr[2]=Flash_Addr[2];
  lAddr.addr[3]=Flash_Addr[3];
  SMPL_Ioctl(IOCTL_OBJ_ADDR, IOCTL_ACT_SET, &lAddr);

  MCU_Init();
  //Transmit splash screen and network init notification
  TXString( (char*)splash, sizeof splash);
  TXString( "\r\nInitializing Network....", 26 );

  SMPL_Init(sCB);

  // network initialized
  TXString( "Done\r\n", 6);

  // main work loop
  while (1)
  {
    // Wait for the Join semaphore to be set by the receipt of a Join
frame from a
    // device that supports and End Device.

    if (sJoinSem && (sNumCurrentPeers < NUM_CONNECTIONS))
    {
      // listen for a new connection
      SMPL_LinkListen(&sLID[sNumCurrentPeers]);
      sNumCurrentPeers++;
      BSP_ENTER_CRITICAL_SECTION(intState);
      if (sJoinSem)
      {
        sJoinSem--;
      }
      BSP_EXIT_CRITICAL_SECTION(intState);
    }

    // if it is time to measure our own temperature...
    if(sSelfMeasureSem)
    {
      char msg [6];
     //Sostituito per uniformare l'address da HUB0 a 000
      char addr[] = {"000"};
      char rssi[] = {"000"};
      int degC, volt;
      volatile long temp;
      int results[2];

      ADC10CTL1 = INCH_10 + ADC10DIV_4;     // Temp Sensor ADC10CLK/5
      ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE +
ADC10SR;
      for( degC = 240; degC > 0; degC-- );  // delay to allow reference
to settle
      ADC10CTL0 |= ENC + ADC10SC;           // Sampling and conversion
start
```

```
      __bis_SR_register(CPUOFF + GIE);      // LPM0 with interrupts
enabled
      results[0] = ADC10MEM;

      ADC10CTL0 &= ~ENC;

      ADC10CTL1 = INCH_11;                  // AVcc/2
      ADC10CTL0 = SREF_1 + ADC10SHT_2 + REFON + ADC10ON + ADC10IE +
REF2_5V;
      for( degC = 240; degC > 0; degC-- );  // delay to allow reference
to settle
      ADC10CTL0 |= ENC + ADC10SC;           // Sampling and conversion
start
      __bis_SR_register(CPUOFF + GIE);      // LPM0 with interrupts
enabled
      results[1] = ADC10MEM;
      ADC10CTL0 &= ~ENC;
      ADC10CTL0 &= ~(REFON + ADC10ON);      // turn off A/D to save power

      // oC = ((A10/1024)*1500mV)-986mV)*1/3.55mV = A10*423/1024 - 278
      // the temperature is transmitted as an integer where 32.1 = 321
      // hence 4230 instead of 423
      temp = results[0];
      degC = (((temp - 673) * 4230) / 1024);
      if( tempOffset != 0xFFFF )
      {
        degC += tempOffset;
      }

      temp = results[1];
      volt = (temp*25)/512;

      msg[0] = degC&0xFF;
      msg[1] = (degC>>8)&0xFF;
      msg[2] = volt;
      transmitDataString(addr, rssi, msg );
      BSP_TOGGLE_LED1();
      sSelfMeasureSem = 0;
    }

    // Have we received a frame on one of the ED connections?
    // No critical section -- it doesn't really matter much if we miss a
poll
    if (sPeerFrameSem)
    {
      uint8_t     msg[MAX_APP_PAYLOAD], len, i;

      // process all frames waiting
      for (i=0; i<sNumCurrentPeers; ++i)
      {
        if (SMPL_Receive(sLID[i], msg, &len) == SMPL_SUCCESS)
        {
          ioctlRadioSiginfo_t sigInfo;
          sigInfo.lid = sLID[i];
          SMPL_Ioctl(IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SIGINFO, (void
*)&sigInfo);
```

```
            transmitData( i, (signed char)sigInfo.sigInfo[0], (char*)msg );
            BSP_TOGGLE_LED2();
            BSP_ENTER_CRITICAL_SECTION(intState);
            sPeerFrameSem--;
            BSP_EXIT_CRITICAL_SECTION(intState);
          }
        }
      }
    }
}

/*-------------------------------------------------------------------------
-------
*
-------------------------------------------------------------------------
-----*/
void createRandomAddress()
{
  unsigned int rand, rand2;
  do
  {
    rand = TI_getRandomIntegerFromVLO();     // first byte can not be 0x00
of 0xFF
  }
  while( (rand & 0xFF00)==0xFF00 || (rand & 0xFF00)==0x0000 );
  rand2 = TI_getRandomIntegerFromVLO();

  BCSCTL1 = CALBC1_1MHZ;                     // Set DCO to 1MHz
  DCOCTL = CALDCO_1MHZ;
  FCTL2 = FWKEY + FSSEL0 + FN1;              // MCLK/3 for Flash Timing
Generator
  FCTL3 = FWKEY + LOCKA;                     // Clear LOCK & LOCKA bits
  FCTL1 = FWKEY + WRT;                       // Set WRT bit for write
operation

  Flash_Addr[0]=(rand>>8) & 0xFF;
  Flash_Addr[1]=rand & 0xFF;
  Flash_Addr[2]=(rand2>>8) & 0xFF;
  Flash_Addr[3]=rand2 & 0xFF;

  FCTL1 = FWKEY;                             // Clear WRT bit
  FCTL3 = FWKEY + LOCKA + LOCK;              // Set LOCK & LOCKA bit
}

/*-------------------------------------------------------------------------
-------
*
-------------------------------------------------------------------------
-----*/
void transmitData(int addr, signed char rssi,  char msg[MESSAGE_LENGTH] )
{
  //Modificato l'indirizzo, ridotto a 3 char: uint8_t 0-255
  char addrString[3];
  char rssiString[3];
  volatile signed int rssi_int;
```

```
  addrString[0] = '0';
  addrString[1] = '0'+(((addr+1)/10)%10);
  addrString[2] = '0'+((addr+1)%10);
  rssi_int = (signed int) rssi;
  rssi_int = rssi_int+128;
  rssi_int = (rssi_int*100)/256;
  rssiString[0] = '0'+(rssi_int%10);
  rssiString[1] = '0'+((rssi_int/10)%10);
  rssiString[2] = '0'+((rssi_int/100)%10);

  transmitDataString( addrString, rssiString, msg );
}

/*-------------------------------------------------------------------------
-------
*
-------------------------------------------------------------------------
-----*/

// Funzione modificata, delega al wrapper l'output dei dati

void transmitDataString(char addr[3],char rssi[3], char
msg[MESSAGE_LENGTH] ) {
        char string_to_wrapper[STRING_LENGTH];
      //costruisco la stringa da inviare
      string_to_wrapper[0] = 0x02; //STX carattere inizio dati
      string_to_wrapper[1] = addr[0];
      string_to_wrapper[2] = addr[1];
      string_to_wrapper[3] = addr[2];
      string_to_wrapper[4] = rssi[0];
      string_to_wrapper[5] = rssi[1];
      string_to_wrapper[6] = rssi[2];
      string_to_wrapper[7] = msg[0];
      string_to_wrapper[8] = msg[1];
      string_to_wrapper[9] = msg[2];
         string_to_wrapper[10] = 0x03; //fine stringa dati
         //passo la stringa alla funzione per l'invio
      TXString(string_to_wrapper, STRING_LENGTH);
}

/*-------------------------------------------------------------------------
-------
*
-------------------------------------------------------------------------
-----*/
void TXString( char* string, int length )
{
  int pointer;
  for( pointer = 0; pointer < length; pointer++)
  {
    volatile int i;
    UCA0TXBUF = string[pointer];
    while (!(IFG2&UCA0TXIFG));              // USCI_A0 TX buffer ready?
  }
}
```

```
//Funzione per l'invio della FREQ al rispettivo ED
void trasmitToED (uint8_t addr) {
    addr --; //Sostituito l'invio al singolo dispositivo con un invio
broadcast, indirizzo settato in programmazione su ogni ED
    if (SMPL_SUCCESS == SMPL_Send(sLID[addr], recMsg, 3)) {
      BSP_TURN_ON_LED2();
      __delay_cycles (20000000);
      BSP_TURN_OFF_LED2();
    }
    else { //in caso di invio fallito
      BSP_TURN_ON_LED1();
      __delay_cycles (20000000);
      BSP_TURN_OFF_LED1();
    }
}

//Funzione numLed accende e spegne i led il numero di volte passato
void numLed (uint8_t num) {
     for (volatile int counter=0; counter < num; counter++) {
          BSP_TURN_ON_LED1();
               BSP_TURN_ON_LED2();
               __delay_cycles (2000000);
               BSP_TURN_OFF_LED1();
               BSP_TURN_OFF_LED2();
               __delay_cycles (1000000);

        }
}

/*-------------------------------------------------------------------------
-------
*
-------------------------------------------------------------------------
-----*/
void MCU_Init()
{
  BCSCTL1 = CALBC1_8MHZ;                    // Set DCO
  DCOCTL = CALDCO_8MHZ;

  BCSCTL3 |= LFXT1S_2;                      // LFXT1 = VLO
  TACCTL0 = CCIE;                           // TACCR0 interrupt enabled
  TACCR0 = 12000;                           // 12000 ~1 second
  TACTL = TASSEL_1 + MC_1;                  // ACLK, upmode

  P3SEL |= 0x30;                            // P3.4,5 = USCI_A0 TXD/RXD
  UCA0CTL1 = UCSSEL_2;                      // SMCLK
  UCA0BR0 = 0x41;                           // 9600 from 8Mhz
  UCA0BR1 = 0x3;
  UCA0MCTL = UCBRS_2;
  UCA0CTL1 &= ~UCSWRST;                     // **Initialize USCI state
machine**
  IE2 |= UCA0RXIE;                          // Enable USCI_A0 RX
interrupt
  __enable_interrupt();
}
```

```
/*-------------------------------------------------------------------------
-------
* Runs in ISR context. Reading the frame should be done in the
* application thread not in the ISR thread.
-------------------------------------------------------------------------
-----*/
static uint8_t sCB(linkID_t lid)
{
  if (lid)
  {
    sPeerFrameSem++;
  }
  else
  {
    sJoinSem++;
  }
  // leave frame to be read by application.
  return 0;
}

/*-------------------------------------------------------------------------
-------
* ADC10 interrupt service routine
-------------------------------------------------------------------------
-----*/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
  __bic_SR_register_on_exit(CPUOFF);        // Clear CPUOFF bit from
0(SR)
}

/*-------------------------------------------------------------------------
-------
* Timer A0 interrupt service routine
-------------------------------------------------------------------------
-----*/
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  //caiti: funzionalità timer
  if (timer!=0) {
    if (count == timer ) {
      sSelfMeasureSem = 1;
      count = 0;
    }
    else count ++;
  }
}


/*-------------------------------------------------------------------------
-------
* USCIA interrupt service routine
-------------------------------------------------------------------------
-----*/
```

```
// Interrupt, per ricevere i dati inviati dal wrapper
// Ricezione di un totale di 3 BYTE

#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
  if (sReceiveMsg == 0) { //ricezione primo byte
    recMsg[2] = UCA0RXBUF;
    addrNode[0] = UCA0RXBUF;
    sReceiveMsg++;
    numLed (1);
  } else  if (sReceiveMsg == 1) { //ricezione secondo byte
            recMsg[0] = UCA0RXBUF;
            sReceiveMsg++;
            numLed (1);
          } else if (sReceiveMsg == 2) { //ricezione terzo e ultimo byte
                   recMsg[1] = UCA0RXBUF;
                   numLed (2);
                   //se l'indirizzo è AP, si setta il proprio timer
                   if (addrNode[0] == 0) {
                     //ricostruzione intero a 16 bit
                     uint16_t freq = 0;
                     freq = recMsg[1];
                     freq = freq << 8;
                     uint16_t temp = 65535;
                     temp = temp & recMsg[0];
                     freq = freq | temp;
                     //settaggio del frequenza di rilevazione
                     timer = freq;
                     //azzeramento del contatore
                     count = 0;
                   }
                   else {
                     //se l'indirizzo è un ED della rete, lancio la
funzione per l'invio della frequenza
                     trasmitToED(addrNode[0]);
                   }
                 sReceiveMsg = 0;
                 }
}
```

A.3 End Device

```
#include "bsp.h"
#include "mrfi.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"
#include "vlo_rand.h"

void linkTo(void);
void MCU_Init(void);

__no_init volatile int tempOffset @ 0x10F4; // Temperature offset set at
production
__no_init volatile char Flash_Addr[4] @ 0x10F0; // Flash address set
randomly

//Variabili contatori per il timer di rilevazione
uint16_t timer = 0; //Fino a 65535 sec circa 18 ore
uint16_t count = 0;
uint8_t vai = 0;

void createRandomAddress();

//Aggiunto il Callback handler per la ricezione di frame
static uint8_t  sCB(linkID_t);
static uint8_t  sPeerFrameSem = 0;
static linkID_t linkID1;
static uint8_t  sSelfMeasureSem = 0;

//Nuove funzioni
void numLed (uint8_t);

//Indirizzo esplicito sensore, da settare diverso per ogni ED della rete
uint8_t myAddr = 2;

void main (void)
{
  addr_t lAddr;
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  {
  // delay loop to ensure proper startup before SimpliciTI increases DCO
  // This is typically tailored to the power supply used, and in this
case
  // is overkill for safety due to wide distribution.
    volatile int i;
    for(i = 0; i < 0xFFFF; i++){}
  }
  if( CALBC1_8MHZ == 0xFF )                 // Do not run if cal values
are erased
  {
    volatile int i;
    P1DIR |= 0x03;
    BSP_TURN_ON_LED1();
```

```
    BSP_TURN_OFF_LED2();
    while(1)
    {
      for(i = 0; i < 0x5FFF; i++){}
      BSP_TOGGLE_LED2();
      BSP_TOGGLE_LED1();
    }
  }

  // SimpliciTI will change port pin settings as well
  P1DIR = 0xFF;
  P1OUT = 0x00;
  P2DIR = 0x27;
  P2OUT = 0x00;
  P3DIR = 0xC0;
  P3OUT = 0x00;
  P4DIR = 0xFF;
  P4OUT = 0x00;

  BSP_Init();

  if( Flash_Addr[0] == 0xFF &&
      Flash_Addr[1] == 0xFF &&
      Flash_Addr[2] == 0xFF &&
      Flash_Addr[3] == 0xFF )
  {
    createRandomAddress();                    // set Random device address
at initial startup
  }
  lAddr.addr[0]= Flash_Addr[0];
  lAddr.addr[1]= Flash_Addr[1];
  lAddr.addr[2]= Flash_Addr[2];
  lAddr.addr[3]= Flash_Addr[3];
  SMPL_Ioctl(IOCTL_OBJ_ADDR, IOCTL_ACT_SET, &lAddr);
  BCSCTL1 = CALBC1_8MHZ;                      // Set DCO after random
function
  DCOCTL = CALDCO_8MHZ;

  BCSCTL3 |= LFXT1S_2;                        // LFXT1 = VLO
  TACCTL0 = CCIE;                             // TACCR0 interrupt enabled
  TACCR0 = 12000;                             // ~ 1 sec
  TACTL = TASSEL_1 + MC_1;                    // ACLK, upmode



  // keep trying to join until successful. toggle LEDS to indicate that
  // joining has not occurred. LED3 is red but labeled LED 4 on the EXP
  // board silkscreen. LED1 is green.
  while (SMPL_NO_JOIN == SMPL_Init(sCB))//caiti:SMPL_Init((uint8_t
(*)(linkID_t))0))
  {
    BSP_TOGGLE_LED1();
    BSP_TOGGLE_LED2();;
  }
  // unconditional link to AP which is listening due to successful join.
  linkTo();
```

```
}

void createRandomAddress()
{
  unsigned int rand, rand2;
  do
  {
    rand = TI_getRandomIntegerFromVLO();     // first byte can not be 0x00
of 0xFF
  }
  while( (rand & 0xFF00)==0xFF00 || (rand & 0xFF00)==0x0000 );
  rand2 = TI_getRandomIntegerFromVLO();

  BCSCTL1 = CALBC1_1MHZ;                     // Set DCO to 1MHz
  DCOCTL = CALDCO_1MHZ;
  FCTL2 = FWKEY + FSSEL0 + FN1;              // MCLK/3 for Flash Timing
Generator
  FCTL3 = FWKEY + LOCKA;                     // Clear LOCK & LOCKA bits
  FCTL1 = FWKEY + WRT;                       // Set WRT bit for write
operation

  Flash_Addr[0]=(rand>>8) & 0xFF;
  Flash_Addr[1]=rand & 0xFF;
  Flash_Addr[2]=(rand2>>8) & 0xFF;
  Flash_Addr[3]=rand2 & 0xFF;

  FCTL1 = FWKEY;                             // Clear WRT bit
  FCTL3 = FWKEY + LOCKA + LOCK;              // Set LOCK & LOCKA bit
}

void linkTo()
{
  //linkID_t linkID1; caiti: ridefinito statico globale
  uint8_t  msg[3];

  // keep trying to link...
  while (SMPL_SUCCESS != SMPL_Link(&linkID1))
  {
    //debug:__bis_SR_register(LPM3_bits + GIE);     // LPM3 with
interrupts enabled
    BSP_TOGGLE_LED1();
    BSP_TOGGLE_LED2();
  }

  // Turn off all LEDs
  if (BSP_LED1_IS_ON())
  {
    BSP_TOGGLE_LED1();
  }
  if (BSP_LED2_IS_ON())
  {
    BSP_TOGGLE_LED2();
  }
  while (1)
  {
    volatile long temp;
```

```
    int degC, volt;
    int results[2];
    if (sSelfMeasureSem) {
    //RIMOSSA LA LPM3 PER PERMETTERE RICEZIONE FRAME
    //SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SLEEP, "" );
    //__bis_SR_register(LPM0_bits+GIE);       // LPM3 with interrupts
enabled
    //SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_AWAKE, "" );


       BSP_TOGGLE_LED2();
        ADC10CTL1 = INCH_10 + ADC10DIV_4;       // Temp Sensor ADC10CLK/5
        ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE +
ADC10SR;
        for( degC = 240; degC > 0; degC-- );    // delay to allow
reference to settle
        ADC10CTL0 |= ENC + ADC10SC;             // Sampling and
conversion start
        __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts
enabled
        results[0] = ADC10MEM;

        ADC10CTL0 &= ~ENC;

       ADC10CTL1 = INCH_11;                      // AVcc/2
       ADC10CTL0 = SREF_1 + ADC10SHT_2 + REFON + ADC10ON + ADC10IE +
REF2_5V;
       for( degC = 240; degC > 0; degC-- );    // delay to allow
reference to settle
       ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion
start
       __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts
enabled
       results[1] = ADC10MEM;
       ADC10CTL0 &= ~ENC;
       ADC10CTL0 &= ~(REFON + ADC10ON);        // turn off A/D to save
power

    // oC = ((A10/1024)*1500mV)-986mV)*1/3.55mV = A10*423/1024 - 278
    // the temperature is transmitted as an integer where 32.1 = 321
    // hence 4230 instead of 423
        temp = results[0];
        degC = ((temp - 673) * 4230) / 1024;
        if( tempOffset != 0xFFFF )
       {
         degC += tempOffset;
       }
    /*message format,  UB = upper Byte, LB = lower Byte
    -------------------------------
    |degC LB | degC UB |  volt LB |
    -------------------------------
       0         1          2
    */

       temp = results[1];
      volt = (temp*25)/512;
```

```
        msg[0] = degC&0xFF;
       msg[1] = (degC>>8)&0xFF;
       msg[2] = volt;

      if (SMPL_SUCCESS == SMPL_Send(linkID1, msg, sizeof(msg)))
      {
        BSP_TOGGLE_LED2();
      }
      else
      {
        BSP_TOGGLE_LED2();
        BSP_TOGGLE_LED1();
      }
       sSelfMeasureSem = 0;
    }
    //A fine ciclo c'è il controllo di messaggi in arrivo dall'ap
    if (sPeerFrameSem) {
      uint8_t msg_rcv[3], len;

      //riporto a zero il semaforo per la coda in ingresso
      bspIState_t intState;
      BSP_ENTER_CRITICAL_SECTION(intState);
      sPeerFrameSem--;
      BSP_EXIT_CRITICAL_SECTION(intState);

      //se il messaggio è ricevuto correttamente lancio la funzione
numLed
      if (SMPL_Receive(linkID1, msg_rcv, &len) == SMPL_SUCCESS) {
        //controllo se sono il giusto destinatario
        if (msg_rcv[2] == myAddr) {
          numLed(1);
          //ricostruzione intero a 16 bit
          uint16_t freq = 0;
          freq = msg_rcv[1];
          freq = freq << 8;
          uint16_t temp = 65535;
          temp = temp & msg_rcv[0];
          freq = freq | temp;
          //settaggio del frequenza di rilevazione
          timer = freq;
          //azzeramento del contatore
          count = 0;
        } else //in caso contrario non faccio nulla
          {numLed(2);}
      }

    }
  }
}

/*-------------------------------------------------------------------------
-------
* ADC10 interrupt service routine
-------------------------------------------------------------------------
-----*/
#pragma vector=ADC10_VECTOR
```

```
__interrupt void ADC10_ISR(void)
{
  __bic_SR_register_on_exit(CPUOFF);        // Clear CPUOFF bit from
0(SR)
}

/*-------------------------------------------------------------------------
-------
* Timer A0 interrupt service routine
-------------------------------------------------------------------------
-----*/
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  //__bic_SR_register_on_exit(LPM0_bits);        // Clear LPM3 bit from
0(SR)

  //Funzionalità timer
  if (timer!=0) {
    if (count == timer ) {
      count = 0;
      sSelfMeasureSem = 1;
    }
    else {
      numLed(1);
      count ++;
    }
  }

}

//Nuovo Callback handler
static uint8_t sCB(linkID_t lid)
{
  if (lid == linkID1)
  {
    sPeerFrameSem++;
    return 0;
  }

  return 1;
}

//Funzione numLed accende e spegne i led il numero di volte passato
void numLed (uint8_t num) {
     for (volatile int counter=0; counter < num; counter++) {
          BSP_TURN_ON_LED1();
               BSP_TURN_ON_LED2();
               __delay_cycles (2000000);
               BSP_TURN_OFF_LED1();
               BSP_TURN_OFF_LED2();
               __delay_cycles (1000000);

        }
}
```

A.4 Tabelle MySQL

A.41 Database locale

```
SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";

--
-- Database: `wsn`
--

-- --------------------------------------------------------

--
-- Struttura della tabella `change_config`
--

CREATE TABLE IF NOT EXISTS `change_config` (
  `IDQ` int(10) unsigned NOT NULL,
  `INS` tinyint(1) unsigned NOT NULL,
  `DEL` tinyint(1) unsigned NOT NULL,
  `ACT` tinyint(1) unsigned NOT NULL,
  `DISACT` tinyint(1) unsigned NOT NULL,
  PRIMARY KEY (`IDQ`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dump dei dati per la tabella `change_config`
--


-- --------------------------------------------------------

--
-- Struttura della tabella `conditions`
--

CREATE TABLE IF NOT EXISTS `conditions` (
  `IDQ` int(10) unsigned NOT NULL,
  `MAX` float DEFAULT NULL,
  `MIN` float DEFAULT NULL,
  `PDH` float DEFAULT NULL,
  `PDL` float DEFAULT NULL,
  `NDH` float DEFAULT NULL,
  `NDL` float DEFAULT NULL,
  PRIMARY KEY (`IDQ`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dump dei dati per la tabella `conditions`
--


-- --------------------------------------------------------
```

```
--
-- Struttura della tabella `configuration`
--

CREATE TABLE IF NOT EXISTS `configuration` (
  `IDQ` int(10) unsigned NOT NULL,
  `IDDEV` int(10) unsigned NOT NULL,
  `FREQ` int(10) unsigned NOT NULL,
  `ACTIVE` tinyint(1) NOT NULL,
  PRIMARY KEY (`IDQ`,`IDDEV`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dump dei dati per la tabella `configuration`
--


-- --------------------------------------------------------

--
-- Struttura della tabella `datapublic`
--

CREATE TABLE IF NOT EXISTS `datapublic` (
  `IDQ` int(10) unsigned NOT NULL,
  `IDDEV` int(10) unsigned NOT NULL,
  `DATE` datetime NOT NULL,
  `TEMP` float NOT NULL,
  PRIMARY KEY (`IDQ`,`IDDEV`,`DATE`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dump dei dati per la tabella `datapublic`
--


-- --------------------------------------------------------

--
-- Struttura della tabella `datareleaved`
--

CREATE TABLE IF NOT EXISTS `datareleaved` (
  `ADDRESS` int(10) unsigned NOT NULL,
  `TEMP` float NOT NULL,
  `BATTERY` float NOT NULL,
  `SIGNAL` int(10) unsigned NOT NULL,
  `COUNT` int(10) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`COUNT`) USING BTREE
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=8444 ;

--
-- Dump dei dati per la tabella `datareleaved`
--
```

```
-- --------------------------------------------------------

--
-- Struttura della tabella `device`
--

CREATE TABLE IF NOT EXISTS `device` (
  `IDDEV` int(10) unsigned NOT NULL,
  `FREQ_SETTED` int(10) unsigned DEFAULT NULL,
  `LAST_TEMP` float DEFAULT NULL,
  PRIMARY KEY (`IDDEV`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dump dei dati per la tabella `device`--
```

A.42 Database remoto

```
SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";

--
-- Database: `wsn_remote`
--

-- --------------------------------------------------------

--
-- Struttura della tabella `difference`
--

CREATE TABLE IF NOT EXISTS `difference` (
  `idq` int(10) unsigned NOT NULL,
  `date` datetime NOT NULL,
  `max` float DEFAULT NULL,
  `min` float DEFAULT NULL,
  `pdh` float DEFAULT NULL,
  `pdl` float DEFAULT NULL,
  `ndh` float DEFAULT NULL,
  `ndl` float DEFAULT NULL,
  PRIMARY KEY (`idq`,`date`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dump dei dati per la tabella `difference`
--
```

Bibliografia

[1] J.A. Stankovic. Wireless Sensor Networks. 2006.

[2] IEEE 802.3-2008 - IEEE Standard for Information technology--Telecommunications and information exchange between systems--Local and metropolitan area networks--Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. 2008.

[3] IEEE 802.16.2-2004 - IEEE recommended practice for local and metropolitan area networks. Coexistence of fixed broadband wireless access systems. 2004.

[4] V. Hsu, J. M. Kahn, and K. S. J. Pister, "Wireless Communications for Smart Dust", Electronics Research Laboratory Technical Memorandum Number M98/2. 1998.

[5] K. S. J. Pister, J. M. Kahn and B. E. Boser, "Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes", Highlight Article in Electronics Research Laboratory Research Summary. 1999.

[6] Using RSSI value for distance estimation in wireless sensor networks based on ZigBee Benkic, K.; Malajner, M.; Planinsic, P.; Cucej, Z.; Systems, Signals and Image Processing, 2008. IWSSIP 2008. 15th International Conference. 2008

[7] H. Park, J. Friedman, M.B. Srivastava, and J. Burke. A new light sensing module for mica motes. Sensors, 2005 IEEE. 2005.

[8] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling Ultra-Low Power Wireless Research. Computer Science Department University of California, Berkeley. 2005.

[9] J. Beutel. Fast-prototyping using the btnode platform. Design, Automation and Test in Europe Conference and Exhibition. 2006.

[10] Riccardo Crepaldi. Algoritmi di localizzazione per reti di sensori: progettazione e realizzazione di una piattaforma sperimentale. PhD thesis, Università degli Studi di Padova Dipartimento di Ingegneria dell'Informazione, 2006.

[11] Libelium, SquidBee – website: http://www.libelium.com/squidbee/index.php?title=Main_Page

[12] Firefly Sensor Network. Real-Time & Multimedia Systems Lab Carnegie Mellon University Dept. of Electrical and Computer Engineering – website: http://www.ece.cmu.edu/firefly/

[13] Texas Instruments - Wireless Sensor Monitor Using the eZ430-RF2500 - SLAA378A. 2007

[14] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer and David Culler. TinyOS: An Operating System for Sensor Networks. 2007.

[15] Adam Dunkels, Bj Granvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. Local Computer Networks, Annual IEEE Conference on. 2004.

[16] D. Gay, P. Levis, R. von Behren, M.Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In Proceedings of Programming Language Design and Implementation (PLDI). 2003.

[17] Philip Levis. TinyOS Programming. 2006.

[18] IEEE 802.15 WPAN Task Group 4 (TG4) website, http://www.ieee802.org/15/pub/TG4.html

[19] IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002) - Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs). 2005.

[20] M. Galeev. Home Networking with ZigBee. Embedded System Programming ING. 2004.

[21] S. Madden, W. Hong, J. Hellerstein, and M. Franklin. TinyDB web page. http://telegraph.cs.berkeley.edu/tinydb.

[22] Texas Instruments - MSP430x2xx Family 2008 Mixed Signal Products User's - SLAU144E. 2008.

[23] Chipcon - CC2500 Single Chip Low Cost Low Power RF Transceiver - SWRS040A. 2006.

[24] Texas Instruments - SimpliciTI: Simple Modular RF Network Specification. 2007.

[25] Texas Instruments - SimpliciTI: Application Programming Interface - SWRA221. 2007.

[26] Paul DuBois – MySQL. 2004.

[27] Tangentsoft, MySQL++ - website: http://tangentsoft.net/mysql++/