Università degli studi di Modena e Reggio Emilia

Facoltà di Ingegneria – Sede di Modena

Corso di Laurea Specialistica in Ingegneria Informatica

Controllo e gestione di una *wireless sensor network (WSN)* attraverso un database distribuito *MySQL*

Relatore:

Chiar.ma Prof. Sonia Bergamaschi

Candidato:

Zhu Song

Correlatore:

Ing. Fabio Bertarelli

Anno Accademico 2008-2009

Indice

Introduzione 7
Composizione della tesi 9
Capitolo 1: Wireless Sensor Network (WSN) 11
1.1 Comunicazione con tecnologia wireless 12
1.2 Nodo sensore 13
1.3 Consumi di un nodo sensore 16
1.4 Architettura di una WSN 16
1.4.1 Tipologie di nodi 17
1.4.2 Topologie delle WSN 17
1.5 Texas Instruments eZ430-RF2500 19
1.5.1 MSP430F2274 21
1.5.2 CC2500 22
1.6 WSN usato per il progetto 25
Capitolo 2: MySQL e MySQL++ 27
2.1 MySQL 28
2.1.1 Sviluppo di MySQL 29
2.1.2 MySQL e i database distribuiti 32
2.1.2.1 Creazione di tabelle federated attraverso le stringhe di connessione 33
2.1.2.2 Creazione di tabelle federated attraverso CREATE SERVER 34
2.1.3 Stored procedure 35
2.2 MySQL++ 37
2.2.1 Funzioni principali delle librerie per database 38
2.2.2 Classi principali di MySQL++ 38
Capitolo 3: Architettura del sistema di controllo 41
3.1 Funzionalità del sistema 42
3.2 Componenti del Sistema 45
3.2.1 Interfaccia d'Utente 46
3.2.2 Query Parser 47
3.2.3 Wrapper 47
3.2.4 Event manager 48
3.2.5 Error manager 49
3.3 Database 49
3.3.1 Device 50
3.3.2 Query di Rilevamento 51
3.3.3 Dati rilevati 52
Capitolo 4: Implementazione Query Parser 53
4.1 Principio di funzionamento del Query Parser 54
4.1.1 Analisi delle SQ 54
4.1.1.1 Attributi dei sub-template 57
4.1.2 Esecuzione delle SQ 57
4.2 Classi realizzate per il Query Parser 59
4.2.1 Template 60
4.2.2 Queries 63
4.2.3 SensorQuery 65
4.2.3.1 Parse delle SensorQuery 66
4.2.3.2 Esecuzione delle SensorQuery 69
4.2.4 Value 72

4.2.5 ConnectionData....72
4.2.6 Procedure....73
4.2.7 DistributedQuery....74
4.3 Database per il modulo Query Parser....74
4.3.1 Database per il template....75
4.3.2 Database per le procedure....76
4.4 Sintassi delle SQ....77
4.4.1 Store....77
4.4.2 Alter....78
4.4.3 Drop, stop, restart e select....79
4.5 Esempio: Implementazione della SQ store....80
4.5.1 Database dei template....81
4.5.2 Definire la stored procedure da eseguire sul database del sistema centrale....82
4.5.3 Definire le stored procedure sui database distribuiti....83
4.5.4 Database delle procedure....83
Capitolo 5: Prospettive future....85
5.1 Moduli di supporto....86
5.2 Modulo di controllo automatico....86
5.3 Architettura multi-controller....88
Conclusioni....91
Appendice A: Codice Stored Procedure....93
A1 Stored Procedure sul database centrale per SQ STORE....93
A2 Stored Procedure sui database Host Wrapper per SQ STORE....97
A2.1 Inserimento configurazione sensori....97
A2.2 Inserimento condizioni eventi....97
Bibliografia....99

Indice delle figure

Figura 1 Sensori wireless 11
Figura 2 Architettura nodo sensore 14
Figura 3 Tipi di nodo destinazione 17
Figura 4 Rete a stella 18
Figura 5 Rete ad albero 18
Figura 6 Rete mesh 18
Figura 7 eZ430-RF2500T + interfaccia USB 19
Figura 8 eZ430-RF2500T + estensione per batterie 19
Figura 9 MSP430F22xx 21
Figura 10 Schema a blocchi MSP430F2274 22
Figura 11 CC2500 23
Figura 12 Schema a blocchi del CC2500 24
Figura 13 Rete di sensori wireless realizzato 25
Figura 14 Logo MySQL 28
Figura 15 Schema di funzionamento delle tabelle federated 32
Figura 16 Architettura del sistema 44
Figura 17 Schema a blocchi dei componenti 45
Figura 18 Schema E/R del database per memorizzare le informazioni sui device 50
Figura 19 Schema E/R del database per memorizzare le informazioni sulle query di rilevamento 51
Figura 20Schema E/R del database per memorizzare i valori rilevati 52
Figura 21 Struttura delle query 54
Figura 22 Diagramma di flusso per la suddivisione delle sub-query 56
Figura 23 Schema d'esecuzione delle SQ 58
Figura 24 Classi create per il QueryParser 59
Figura 25 Diagramma di flusso del caricamento dei sub-template 62
Figura 26 Diagramma di flusso della suddivisione delle SQ 64
Figura 27 Diagramma di flusso del parseSubQuery 67
Figura 28 Diagramma di flusso del parseValue 68
Figura 29 Diagramma di flusso dell'esecuzione di una SQ 71
Figura 30 Schema E/R del database per memorizzare i template 75
Figura 31 Schema a blocchi del sistema centrale con il controllo automatico 87
Figura 32 Evoluzione del sistema da un solo server di controllo centrale a più server di controllo 89

Introduzione

I sensori sono dei dispositivi in grado di rilevare determinati fenomeni fisici attraverso l'uso dei trasduttori.

Con l'evoluzione della tecnologia, il mercato dei sensori offre non solo prodotti sempre più precisi e affidabili, ma anche funzionalità sempre più sofisticate, e nuovi tipi di modalità di comunicazione, tra cui la possibilità di comunicare attraverso le tecnologie *wireless*.

Questa evoluzione tecnologica, comunicazione *wireless*, consente ai sensori di poter abbandonare i cavi, ciò rende possibile la realizzazione delle reti di sensori senza fili, *wireless sensor network* (*WSN*), che possono avere applicazioni in moltissimi settori.

Però da un lato il fatto di non avere cavi porta delle possibilità di applicazioni che prima erano irrealizzabili o poco pratici, dall'altra parte apre altre problematiche che possono limitare l'uso di tali dispositivi. Uno di questi problemi è l'alimentazione.

L'assenza del cavo di alimentazione significa che il dispositivo deve essere in grado di funzionare con un'alimentazione autonoma. Nonostante lo sviluppo delle nuove fonti di energia (es. energia solare) e riduzione progressiva dei consumi dei dispositivi elettronici, è ancora impensabile avere un dispositivo che mantenga un funzionamento attivo

permanente con un'alimentazione autonoma. Ciò significa che il dispositivo ha un limite di consumo (la capacità delle batterie, la massima energia solare sfruttabile in un giorno ecc..), perciò per rendere il dispositivo più efficiente possibile è necessario limitare il consumo dell'energia solo nei momenti necessari. Dai dati sperimentali, il maggior consumo di potenza, per i sensori *wireless*, si ha nel momento di trasmissione delle informazioni. Quindi per rendere efficiente il consumo dell'energia è necessario trasmette la minore quantità di dati possibile, ma ciò non deve in alcun modo compromettere la qualità dei dati rilevati dalla *WSN*.

Una delle soluzioni proposte è di poter regolare la frequenza di rilevamenti per raccogliere i dati solo quando i dati sono veramente interessanti, uno dei esempi di trasmissioni efficienti delle informazioni, rilevati dalle reti di sensori è presentato dalla tesi dell'ing Alessio Cavallini [2]. Per poter fare questo è necessario un sistema in grado di controllare le *WSN*, che è l'obbiettivo della tesi.

La progettazione e realizzazione di un componente software, che faccia da interfaccia tra una rete di sensori e un do centrale costituito da un computer è già stata sviluppata dalla tesi dell'ing. Daniele Caiti [3]. Quindi l'obbiettivo di questa tesi, è quello relativo alla progettazione e realizzazione di un componente software che permetta di disegnare ad alto livello un rette di sensori wireless e fornisca un'interfaccia di interrogazione di tale rete di sensori, i cui dati vengono estratti, attraverso l'interazione con il componente realizzato dal Dott. Caiti. Un obiettivo fondamentale della interfaccia di interrogazione è quella di supportare un linguaggio di interrogazione di alto livello che permetta una facile sviluppo di interrogazioni: tale obiettivo e` stato totalmente raggiunto definendo un linguaggio basato su un’estensione del linguaggio *SQL* [1]. Un altro obiettivo fondamentale era quello di una interrogazione efficiente di una rete: anche questo obiettivo e` stato raggiunto definendo una politica di esecuzione di interrogazioni distribuite.

Composizione della tesi

Questa tesi è composta da 5 capitoli, oltre alla introduzione e alle conclusioni:

- Nel primo capitolo viene fornita una presentazione generale delle Wireless Sensor Network (WSN in maniera generale), seguita da una breve descrizione dei componenti hardware utilizzati nel laboratorio di reti di sensori del dipartimento di Ingegneria dell'Informazione;
- Nel capitolo 2 viene presentato il *DBMS* utilizzato, ***MySQL***, insieme alla libreria per interfacciarsi al *database*, ***MySQL++***;
- Nel capitolo 3, viene illustrata l'architettura funzionale dell'intero sistema di controllo delle ***WSN*** verrà realizzato**;**
- Nel capitolo 4 viene descritto il componente software realizzato per la tesi, illustradone il principio di funzionamento e le classi che lo compongono;
- Infine, il capitolo 5 è dedicato ad illustrare le prospettive future del sistema, degli altri componenti necessari al sistema pe una piena funzionalità, e di una possibile evoluzione del sistema.

Capitolo 1: Wireless Sensor Network (WSN)

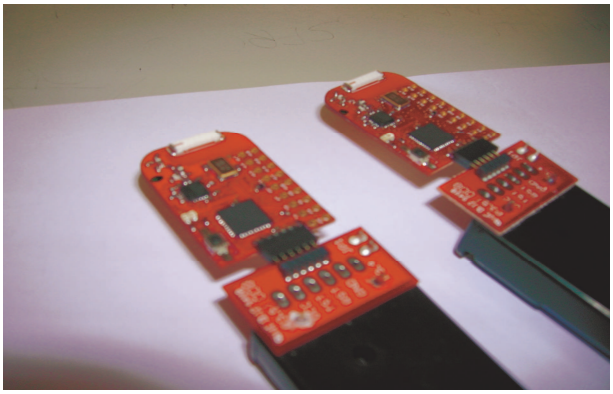


Figura 1 Sensori wireless

Con l'evoluzione tecnologica, i produttori dei sensori cercano di migliorare sempre di più la precisione e l'affidabilità dei sensori. Ma l'evoluzione dei sensori non consiste solo nel miglioramento dei rilevamenti, ma anche nelle modalità di comunicazione, nei consumi sempre più ridotti, nell'aumento delle capacità di elaborazione dei sensori, riduzione delle dimensioni ecc..

Una delle evoluzioni più importanti consiste nell'uso delle comunicazioni in tecnologie *wireless*, che consentono ai sensori di liberarsi dei cavi e di avere la possibilità di creare delle reti di sensori senza fili (*WSN*).

1.1 Comunicazione con tecnologia wireless

Esistono vari sistemi usati per le trasmissioni wireless che possono essere impiegati nella comunicazione tra i sensori wireless e sono:

Sistemi induttivi, la tecnologia che sfrutta questo sistema è la *Radio Frequency Identification* (*RFID*), e consiste nell'utilizzare dei *tag* attivi o passivi, i quali vengono letti, generando un forte campo elettromagnetico che crea una corrente indotta sull'induttanza all'interno del *tag*, ciò consente la lettura e la scrittura del *tag*. I principali svantaggi di questa tecnologia è la distanza per cui è possibile effettuare la corretta operazione di lettura/scrittura, l'elevato consumo di potenza per generare il campo elettromagnetico;

Sistemi ottici, esistono diverse tecnologie che sfruttano il sistema ottico, una di queste è l'*Infrared Data Association*, il principale svantaggio di questo sistema è che richiede la linea di vista (*line of sight*) diretta tra i dispositivi, ciò richiede un allineamento dei dispositivi non sempre possibile;

Ultrasuoni, la tecnologia di questi sistemi non ha il problema della linea di vista, ma lo svantaggio è l'elevato consumo di potenza per la generazione dei ultra suoni, e i costi e le dimensioni dei dispositivi che sfruttano questa tecnologia non sono per nulla contenuti;

Radio-frequenza, l'uso delle radio frequenza consente di superare i problemi legati alla linea di vista, con un consumo relativamente contenuto, il problema è la disponibilità delle frequenze, poiché le bande di frequenza utilizzate sono regolate da legislazioni locali. Le bande *ISM* (*Industrial, Scientific and Medical)* rappresentano una delle scelte migliori effettuabili visto che sono disponibili in molti stati. Le frequenze *ISM* sono state regolamentate dagli standard ETSI EN 301 498-1, ETSI EN 300 328-1 V1.3.1 per l'Europa, FCC CFR 47 per Usa e Canada e ARIB STD-T66 per il Giappone, altri stati hanno delle normative locali che però consentono l'utilizzo di dispositivi che rispettino gli standard ETSI e/o FCC. Si ricorda che per lo sviluppo di una rete di sensori *wireless* si vogliono realizzare dispositivi dai costi e dalle dimensioni contenute ciò porta a considerare solo alcune delle bande *ISM*, questo a causa di alcune limitazioni hardware in termini di efficienza delle antenne e limitazione dei consumi; negli ultimi anni si

stanno concentrando studi per le bande nell'intorno dei 2.4 GHz. Il vantaggio nell'utilizzo delle *ISM* deriva oltre che dalla loro disponibilità, dal fatto che risultano essere svincolate dall'utilizzo di un particolare standard di comunicazione consentendo quindi al progettista ampia libertà nell'implementazione di tecniche di *power saving* fondamentali nelle reti di sensori.

D'altro canto vi sono diverse limitazioni inerenti la potenza di uscita delle trasmissioni radio dovute a ragioni legislative per minimizzare l'inquinamento elettromagnetico ed in secondo luogo motivazioni di ordine economico. Le potenze tipiche consentite all'interno delle *ISM* coprono un *range* compreso fra gli 0 dBm ed i 20 dBm, la potenza limitata imposta alle comunicazioni radio si traduce in coperture limitate del segnale. [5]

Ognuno di questi mezzi presenta vantaggi e svantaggi, e possono essere scelti a seconda dell'applicazione e della topologia della rete che s'intende realizzare.

1.2 Nodo sensore

Per comprendere meglio le caratteristiche delle reti di sensori *wireless*, è necessario prima conoscere i componenti singoli che formano la rete, e tali componenti sono i *nodi sensori*. Un nodo sensore non è solo un trasduttore capace di rilevare un determinato fenomeno, ma essendo parte di un rete, deve avere anche le capacità di elaborazione. Un generico nodo sensore può essere schematizzato con il seguente schema (Figura 2):

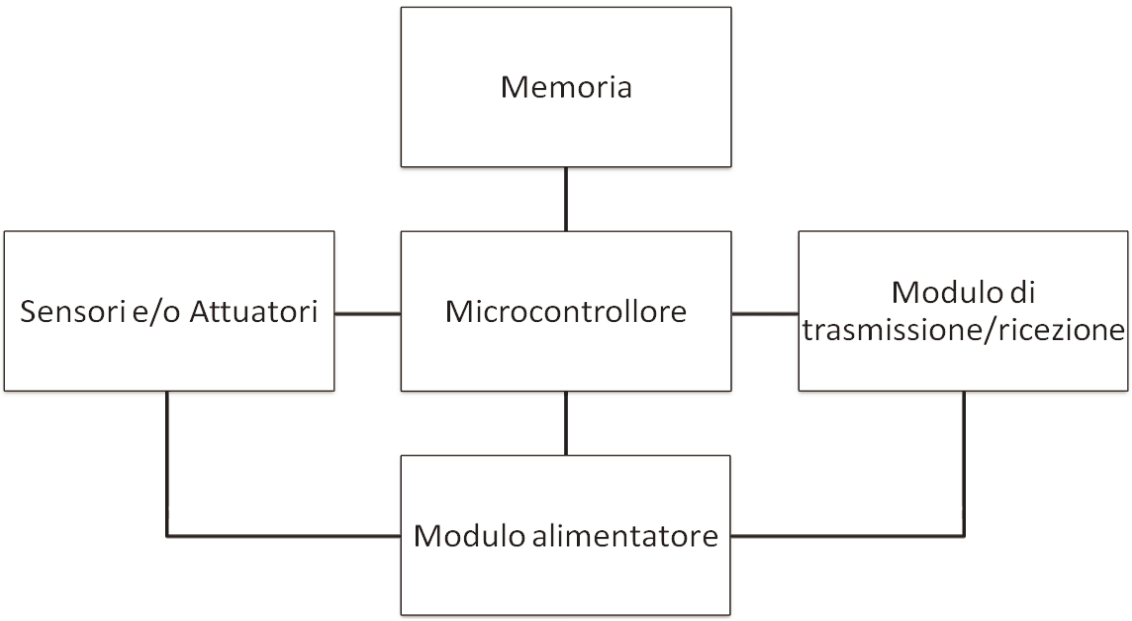


Figura 2 Architettura nodo sensore

I componenti del sistema sono:

- **Microcontrollore**: è la parte centrale del sistema, nella quale risiede le capacità di elaborazione del nodo sensore. Il compito del microcontrollore è la gestione di tutte le periferiche del nodo sensore, tra cui l'attivazione dei attuatori, la gestione della comunicazione e dei protocolli di comunicazione. Inoltre deve occuparsi anche della gestione ed elaborazione dei dati rivelati dai sensori/attuatori.

Ci sono varie soluzioni che si possono adottare per implementare il microcontrollore, una di queste è di usare un processore *general-purpose* programmabile, questa soluzione ha il vantaggio della flessibilità del nod, nella scelta dei processori è necessario ricordare che una delle caratteristiche principali dei processori usati per questo compito è il basso consumo di energia.

Un'altra delle soluzioni possibile è l'uso di un *Application-Specific Integrated Circuit* (*ASIC*), lo svantaggio di questa soluzione è che non è flessibile, in quando un *ASIC* non è programmabile. La perdita della flessibilità in questo caso si traduce in un elevato grado di efficienza del dispositivo sia in termini di velocità computazionale, sia in termini di consumo di potenza, in quanto il dispositivo è ottimizzato per le operazioni specifiche che deve svolgere.

Generalmente si usa la prima soluzione durante la progettazione, mentre per il prodotto finito si preferisce la seconda soluzione.

- **Sensori e/o attuatori**: sono l'interfaccia del nodo sensore con il mondo fisico, e sono i trasduttori usati nel nodo sensore.

  Un trasduttore è un dispositivo, generalmente elettrico o elettronico, in grado di trasformare un tipo di energia legato ad una determinata grandezza in segnali elettrici (sensori) o viceversa (attuatori). Questo dispositivo svolge un ruolo fondamentale nel nodo sensore in quanto serve per rilevare i fenomeni interessati. Il trasduttore influenza in modo significato il costo del dispositivo, le sue dimensioni e il consumo del sensore. In un nodo sensore possono essere presenti uno o più trasduttori.
- **Modulo ricezione e trasmissione**: Questo modulo si occupa della comunicazione tra i nodi sensori. La comunicazione può avvenire attraverso vari sistemi o mezzi di trasmissione, come spiegato nel paragrafo precedente. Nei sensori *wireless* il mezzo di trasmissione più usato è la radio frequenza, le portanti generalmente impiegate per le *WSN* variano da 433MHz a 2.4GHz, a seconda della scelta progettuale.
- **Modulo di alimentazione**: Uno dei vantaggi delle *WSN* è l'assenza dei cavi, il che significa che i nodi sensori non possono ricevere energia attraverso un cavo di alimentazione durante il loro normale funzionamento. Quindi, ogni nodo sensore deve essere dotato di un modulo di alimentazione, che gli fornisca energia necessaria per il funzionamento, in modo autonomo. Il modulo di alimentazione può essere semplicemente una batteria che consente al nodo di funzionare per un periodo di tempo, o per i nodi più costosi, anche un pannello solare, di dimensioni contenute, in grado di fornire energia necessaria. Ovviamente il fatto di poter avere un generatore elettrico interno è molto comodo, ma non bisogna dimenticare che ciò incide molto sui costi dei nodi sensori e sulla loro dimensione.
- **Memoria**: Per le elaborazioni complesse è necessario avere un dispositivo in grado di memorizzare i dati, sia di rilevamento sia le informazioni scambiate con gli altri sensori. Quindi è ovvia l'importanza di avere una memoria all'interno del nodo. Questa memoria può essere una *RAM*, che ha però ha lo svantaggio di perdere le informazioni se viene spento, o le memorie *flash* che mantengono le informazioni anche quando non sono alimentati, la scelta ovviamente ricade sulla memoria che consuma di meno. Ma è necessario considerare tutti gli aspetti, per quanto riguarda la *RAM*, non è possibile togliere l'alimentazione durante i periodi

in cui non la si usa, ma ha il vantaggio di essere più veloce di una *flash*. Mentre la flash, ha tempi di accesso più lunghi, questo significa un maggiore consumo di potenza durante l'uso, per non parlare del consumo delle flash sui cicli di scrittura. Quindi è necessario valutare attentamente l'uso della memoria, all'interno del nodo per scegliere la tipologia di memoria da usare.

1.3 Consumi di un nodo sensore

Uno dei principali vincoli progettuali di un nodo sensore *wireless* o dei sistemi che usano le reti di sensori *wireless, poiché non può ricevere l'alimentazione da un cavo* riguarda il consumo del dispositivo, il quale influenza il tempo di vita del nodo sensore, e quindi la stabilità della rete di sensori, in quanto la "morte" di alcuni nodi sensori possono modificare il maniera significativa il funzionamento della rete.
In un nodo sensore, il consumo energetico è determinato da 3 principali funzioni:

- Il rilevamento dei dati dall'ambiente (*sensing*);
- L'elaborazione dei dati rilevati o ricevuti;
- Comunicazione con gli altri dispositivi della rete, questa è l'operazione in cui il consumo di energia è maggiore.

1.4 Architettura di una WSN

Dopo avere analizzato come può essere un generico nodo della rete, è opportuno spiegare quali possono essere architetture che si possono realizzare con tali nodi.

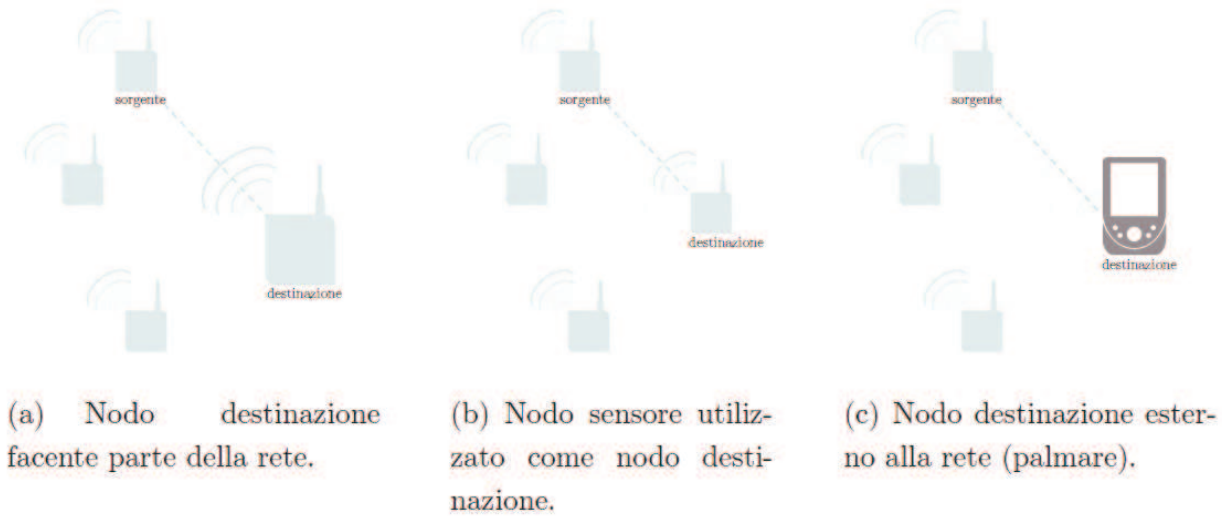
1.4.1 Tipologie di nodi

Intanto è possibile distinguere i nodi all'interno della rete in due tipologie di nodi:

- Nodi sorgenti, la sua funzione è di fornire le informazioni rilevate;
- Nodi destinazione o coordinatore, ha il compito di raccogliere le informazioni rilevati.

Il nodo di destinazione può essere:

- Un dispositivo diverso dai nodi sensori, ma che fa parte della rete;
- Un dispositivo uguale ai nodi sorgenti, ma con il compito di raccogliere le informazioni rilevate dagli altri nodi, ovviamente anche il nodo destinazione può avere la funzione di nodo sorgente;
- Un dispositivo esterno che non fa parte della rete.



(a) Nodo destinazione facente parte della rete.

(b) Nodo sensore utilizzato come nodo destinazione.

(c) Nodo destinazione esterno alla rete (palmare).

Figura 3 Tipi di nodo destinazione

1.4.2 Topologie delle WSN

La topologia di una rete è una rappresentazione schematica del modello geometrico (grafico) della rete.

Le topologie più comuni per una *WSN* sono le seguenti:

- **Rete a stella**: In questa topologia di rete c'è un nodo centrale che ha il compito di

coordinare gli altri nodi. Il vantaggio principale di questa topologia è la semplicità di progettazione, ma il svantaggio principale è rappresentato dalla robustezza di tale rete, in quanto esiste un unico punto di *failure*, se il nodo centrale si guasta compromette tutto il sistema.

Figura 4 Rete a stella

- **Rete ad albero**: Le reti di questa topologia hanno una struttura gerarchica con più livelli di controllo, questa è un'evoluzione delle reti a stella, in cui non c'è solo un coordinatore centrale, ma una gerarchia di coordinatori.

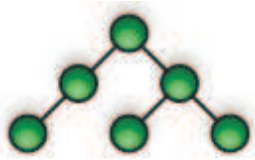


Figura 5 Rete ad albero

- **Rete *Peer to Peer***: Le reti *peer to peer* (*P2P*) sono reti in non ci sono distinzioni tra i vari nodi. Tutti i nodi possono svolgere le stesse funzioni sia come nodo sorgente sia come nodo destinazione.
- **Rete *mesh***: Le reti di tipo *mesh*, hanno la caratteristica che tutti i nodi sono connessi a tutti gli altri nodi, il vantaggio di una tale struttura è ovviamente la robustezza della rete, tutti i nodi sono interconnessi, lo svantaggio è il rappresentato dal numero dei collegamenti, e quindi dal costo dei collegamenti.

Figura 6 Rete mesh

1.5 Texas Instruments eZ430-RF2500

Dopo aver introdotto le *WSN* in maniera generica, in questo paragrafo verrà presentato il dispositivo usato in laboratorio per creare una rete di sensori *wireless*.

I dispositivi utilizzati in laboratorio per lo sviluppo della rete sono prodotti dalla **Texas Instruments**, il nome del dispositivo è **eZ430-RF2500**.

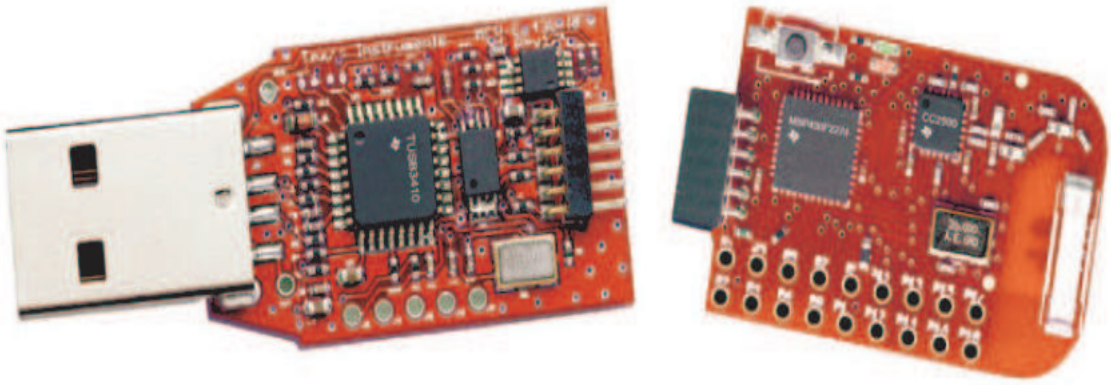


Figura 7 eZ430-RF2500T + interfaccia USB

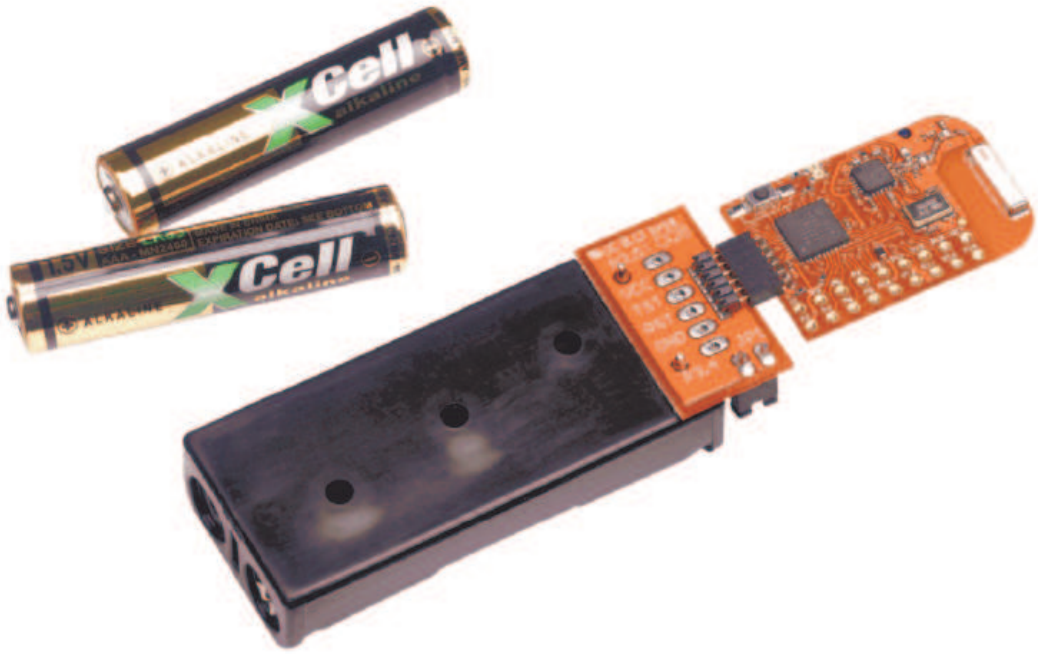


Figura 8 eZ430-RF2500T + estensione per batterie

L'**eZ430-RF2500** è uno strumento di sviluppo basato su USB per valutare le prestazioni del microcontrollore **MSP430F2274** e *transceiver wireless* **CC2500**. Il dispositivo è programmabile attraverso *IAR Embedded Workbench Integrated Development Environment* (*IDE*) o *Code Composer Essentials* (*CCE*).
L'hardware del kit completo dell'**eZ430-RF2500** è composto da:

- 2 *Target Board* (**eZ430-RF2500T**), che è il chip con il microprocessore **MSP430F2274** e *transceiver wireless* **CC2500**;
- Un'interfaccia USB attraverso la quale è possibile collegare il *Target Board* a un calcolatore e programmarlo. Attraverso questo collegamento USB è possibile anche comunicare con il microprocessore. Il collegamento tra l'interfaccia USB e il *Target Board* è reso possibile da un *controller USB to Serial*, chiamato *TUSB3410*. Inoltre l'interfaccia USB fornisce anche l'alimentazione al dispositivo;
- Un modulo per le batterie collegabile con il *Target Board*, questo modulo serve a fornire al chip *Target Board* l'alimentazione quando non è collegato attraverso l'interfaccia USB.

Altri parti presenti sul *Target Board* sono:

- 18 pin, utilizzabili per interfacciarsi al dispositivo;
- 2 led, uno rosso e uno verde utilizzabili come output visuale per l'utente;
- Un bottone, atrveso il quale l'utente può interagire con il dispositivo.

In laboratorio vengono chiamati *End Device* (*ED*) i nodi sorgente, e sono composti solitamente da un *Target Board* e un modulo per le batterie, e vengono chiamati *Access Point* (*AP*) i nodi destinazione, e solitamente sono dei *Target Board* collegati a un calcolatore attraverso l'interfaccia *USB*.

Come detto prima L'**eZ430-RF2500** è uno strumento di sviluppo usato principalmente per valutare il suo microcontrollore e il *tranceiver wireless*, che sono i componenti più importanti del dispositivo nodo sensore.

1.5.1 MSP430F2274

Figura 9 MSP430F22xx

Come detto in precedenza il vincolo del basso consumo dei dispositivi sul è nodo sensore è un vincolo importantissimo, la scelta del processore per un nodo sensore deve tener conto di questo vincolo. Infatti il microprocessore **MSP430F2274** fa parte della famiglia di processori **MSP430**, che è una famiglia di processori *ultralow-power*. Grazie ai loro 6 stati di funzionamento, uno stato attivo e 5 stati di bassi consumi, questi processori sono ottimizzati per allungare la durata delle batterie che lo alimentano.

La serie **MSP430F2274M** è formata dai microcontrollori a basso consumo che opera con i segnali misti (sia analogico che digitale), grazie ad un'interfaccia di comunicazione seriale universale, un convertitore A/D a 10bit, e due amplificatori operazionali *general-purpose*, 32 pin I/O.

Il modello **MSP430F2274** è dotato sia di una memoria *RAM* sia di un memoria *flash*:

- 32 kB + 256B *Flash Memory*;
- 1 kB *RAM*.

Il dispositivo dispone di un CPU RISC a 16 bit con registri a 16 bit, con un set di istruzioni formato da 27 istruzioni e 7 modalità d'indirizzamento. Il *digitally controlled oscillator* (*DCO*) consente il *wake-up*, il passaggio da uno stato di basso consumo alla modalità attiva, in meno di 1μs.

Lo schema seguente riporta i blocchi funzionali che compongono il dispositivo (Figura 10):

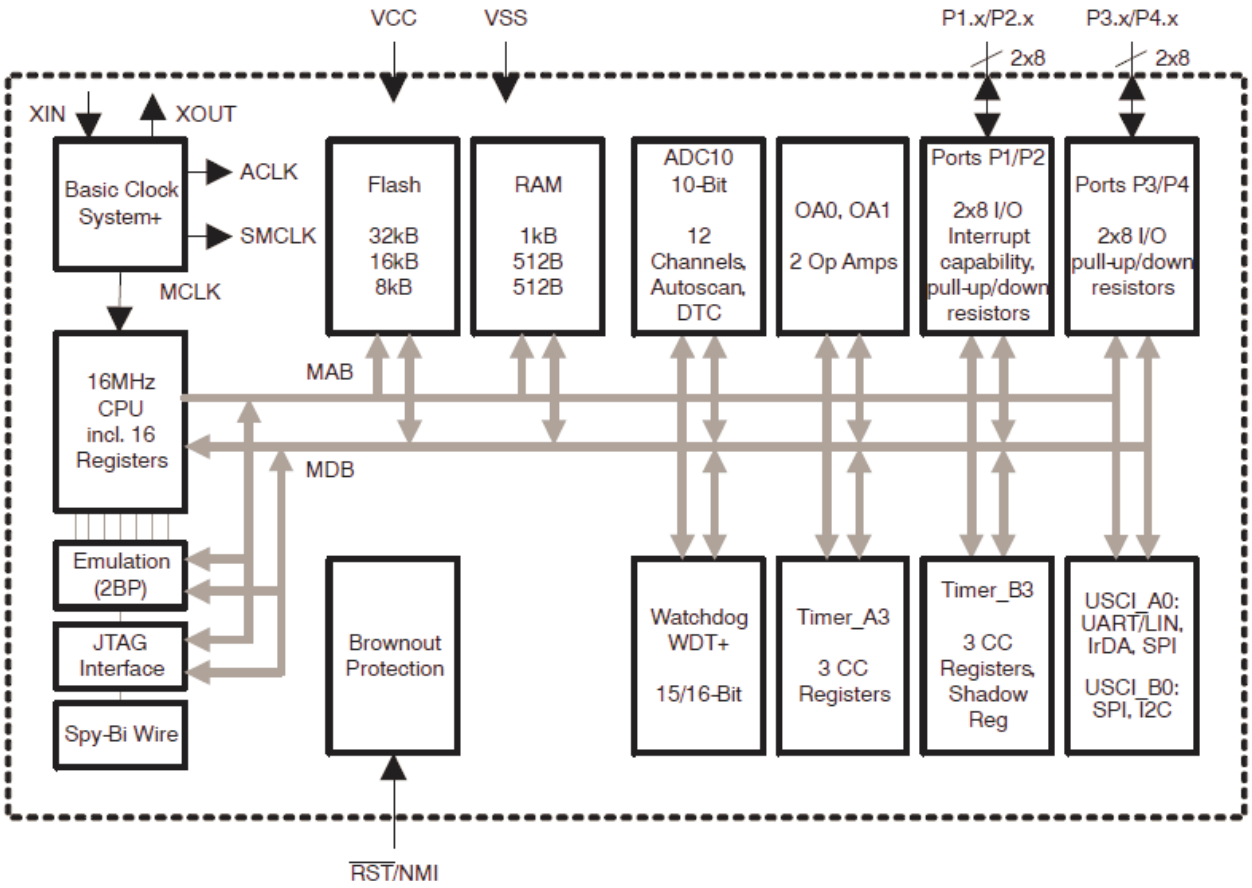


Figura 10 Schema a blocchi MSP430F2274

All'interno del **eZ430-RF2500T**, il **MSP430F2274** svolge le funzioni di 3 moduli:

- Microcontrollore dell'intero nodo sensore;
- Il modulo di memoria, avendo **MSP430F2274** sia una memoria RAM che una memoria flash;
- Sensore di temperatura e del livello di batteria del sensore.

1.5.2 CC2500

Un altro componente importante sul dispositivo nodo sensore *wireless* è il *transceiver*. Il *transceiver* usato su **eZ430-RF2500** è il **CC2500**.

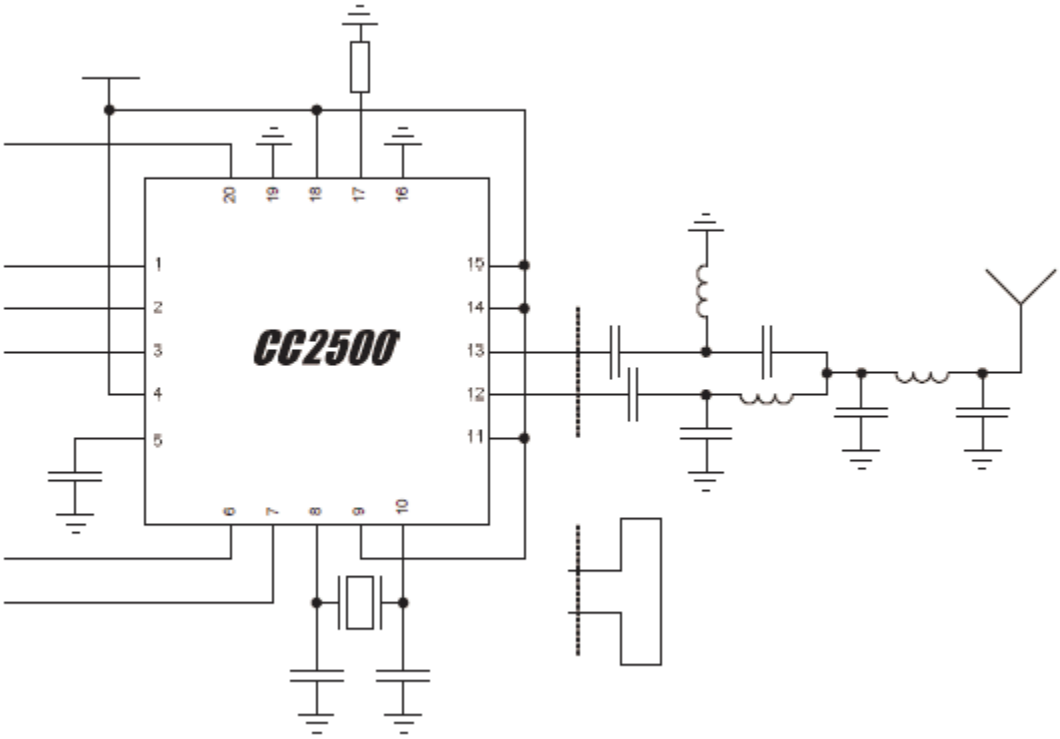


Figura 11 CC2500

Il **CC2500** è un *transceiver* a 2,4 Ghz a basso costo ed a basso consumo. Il circuito opera in una banda di frequenze da 2400 a 2483,5 Mhz, che rientra nella banda *ISM* e *SRD* (*Short Range Device*). Inoltre il dispositivo supporta diverse modulazioni (OOK, 2-FSK, GFSK, MSK) e dispone di un data rate configurabile fino a 500kBaud.

La tensione d'alimentazione del dispositivo va da 1,8 a 3,6 volt, perfetto per le applicazioni che usano le batterie.

Lo schema a blocchi del *transceiver* è la seguente (Figura 12):

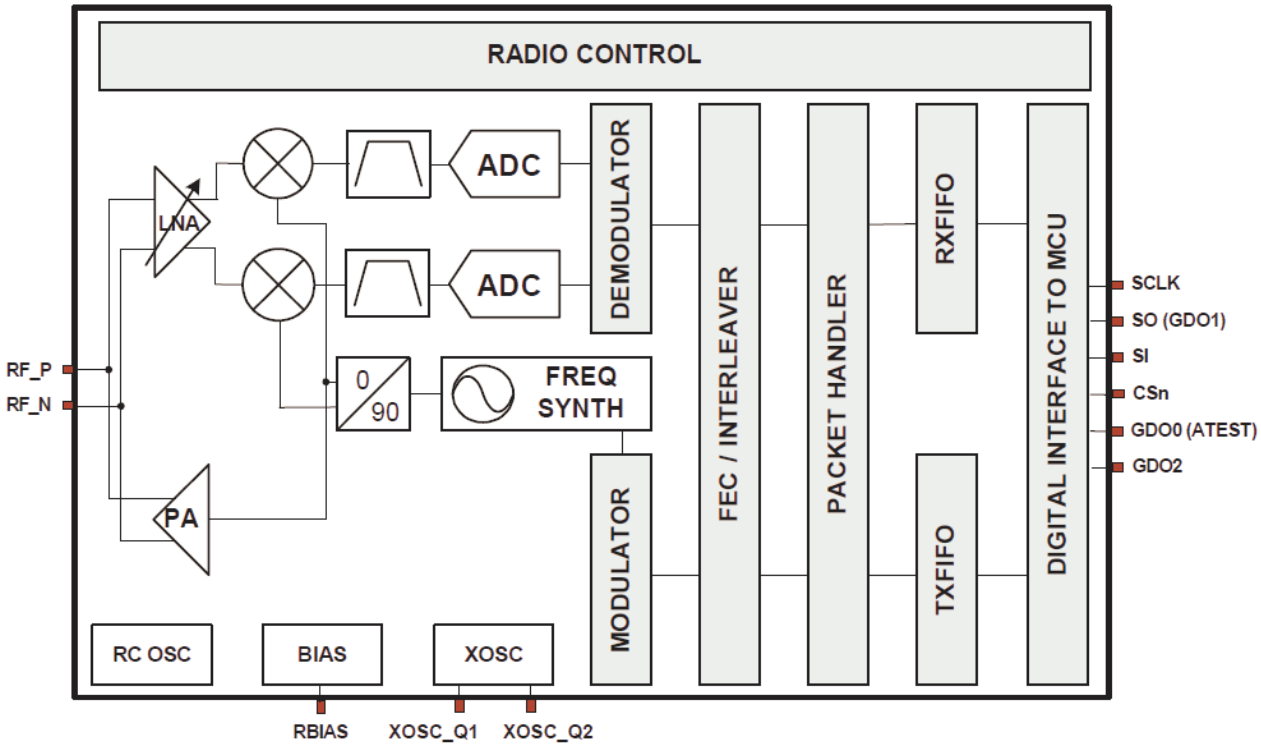


Figura 12 Schema a blocchi del CC2500

Il segnale ricevuto alla frequenza RF viene amplificata dal *Low-Noise Amplifier* (*LNA*) e poi convertito in quadratura (in segnali I e Q) alla frequenza intermedia. I segnali I e Q sono convertiti in segnali digitali dai convertitori A/D. Le operazioni di *Automatic Gain Control* (*AGC*), del filtraggio selettivo, demodulazione e sincronizzazione dei bit dei pacchetti vengono eseguite digitalmente.

La funzione di trasmettitore del **CC2500** è basato sulla sintesi diretta della frequenza RF.

Il sintetizzatore di frequenza ha un oscillatore VCO integrato e e di uno sfasatore a 90 gradi per la generazione dei segnali I e Q in modalità di ricezione.

Sulle porte *XOSC_Q1* e *XOSC_Q2* devono essere collegati al quarzo, che serve all'oscillatore per generare la frequenza di riferimento per il sintetizzatore, per i *timer* dei convertitori A/D e per la parte digitale.

Interfaccia seriale *SPI* a 4 fili consente l'interfacciamento e l'accesso per la configurazione e l'accesso buffer di dati.

1.6 WSN usato per il progetto

Dopo aver presentato l'hardware usato per i nodi sensori, la sezione successiva verrà dedicata alla descrizione della rete *WSN* realizzata per il progetto.

Come si è detto in una rete di sensori esistono 2 tipi di nodi, i nodi sorgente e i nodi destinazione, i nodi sorgente sono i nodi che raccolgono le informazioni dall'ambiente, mentre i nodi destinazione sono i nodi che richiedono le informazioni raccolte. Nella rete sviluppata in laboratorio i nodi sorgente sono chiamati con il nome di *End Device* (*ED*), e sono dei dispositivi composti dal *Target Board* e l'estensione per la batteria. Mentre i nodi destinazione sono chiamati *Access Point* (*AP*) e sono composti da un *Target Board* e collegati a un calcolatore attraverso l'interfaccia *USB*.

La topologia di rete usata è una rete a stella, con un *AP* collegato al calcolatore che coordina gli *ED*. Però è previsto che l'intero sistema possa avere più *AP*, ogni *AP* è controllato da un calcolatore, a cui è collegato attraverso il connettore *USB*, chiamato *host wrapper*. Tutti gli *host wrapper* sono controllati da un calcolatore che ha la funzione di sistema di controllo centrale. Quindi la topologia dell'intero sistema ha una struttura gerarchica, con un calcolatore che coordina gli *host wrapper*, i quali controllano una o più reti di sensori a stella formati dagli *AP* e dai *ED*.

Figura 13 Rete di sensori wireless realizzato

Capitolo 2: MySQL e MySQL++

Nel sistema di controllo, che verrà illustra nel capitolo successivo, il database è una parte fondamentale del sistema. Esso non si occupa solo di tenere in memoria i dati raccolti dai sensori, ma ha anche la struttura del sistema, in che modo sono disposti i sensori, i loro raggruppamenti, le informazioni di connessione dei database dei wrapper ecc... Inoltre i database sono usati anche come mezzo di comunicazione tra il sistema di controllo e i vari wrapper dei sensori.

Uno dei vincoli del progetto del sistema di controllo delle reti di sensori è di sfruttare il più possibile il mondo **Open Source**. Quindi la scelta del database da usare per il sistema è ovviamente **MySQL**, e di conseguenza **MySQL++** come *application program interface* (*API*) tra i programmi in C++ e il database.

2.1 MySQL

Figura 14 Logo MySQL

MySQL è il più diffuso *relational database management system* (*RDBMS*) **Open Source** basato sul linguaggio SQL. Questo prodotto viene fornito dall'azienda **MySQL AB** che sviluppa il proprio business erogando servizi basati su **MySQL** stesso [13].

Un **database** è una raccolta strutturata di dati. Per inserire, accedere ed elaborare i dati memorizzati in un database informatico, è necessario un sistema di gestione di database come **MySQL**. Dal momento che i computer possono contenere una grande quantità di dati, la gestione del database svolge un ruolo fondamentale nel settore informatico.

Un **database relazionale** memorizza i dati in tabelle separate. Le tabelle sono collegate attraverso delle relazioni, con i quali è possibile combinare i dati provenienti da diverse tabelle [14] . Questo, oltre a permettere flessibilità e velocità nell'accesso ai dati, permette una più efficace modellazione della struttura dei dati.

La rapida diffusione di **MySQL** è principalmente legata a 3 fattori:

- La distribuzione di **MySQL** è gratuita per organizzazioni non a scopo di lucro;
- **MySql** si presenta per gli utilizzatori finali un database potente ed estremamente flessibile;
- La possibilità di personalizzare **MySQL** secondo le proprio esigenze. Infatti è possibile modificare il codice sorgente di **MySQL**, da parte di un programmatore esperto, a seconda delle proprie esigenze.

Come si può notare tutti i motivi sono legate al fatto che **MySQL** è **Open Source**, il che ha favorito anche lo sviluppo di interfacce, rilasciate sotto licenza **GPL**, in quasi tutti i linguaggi maggiormente utilizzati (es. **MySQL**), soprattutto nello sviluppo di applicazioni web ma non solo: in **PHP** ad esempio viene fornita in modo nativo al linguaggio la connettività al database server, in Perl la connessione è possibile grazie ai moduli *DBI* e *DBD::MySql*, utilizzando *MySQL Connector/ODBC* (*MyODBC*) è possibile connettersi al database utilizzando applicazioni come ad esempio **Microsoft Access**,

Microsoft Excel o linguaggi di programmazione come **Delphi Borland** o **ASP** e **Visual Basic**, viene inoltre fornito supporto ai client **Java**, sia in ambiente **Windows** che **Unix**, tramite l'interfaccia *Connector/JDBC*. L'accesso ai dati è reso possibile dall'utilizzo di *Structured Query Language* (*SQL*), in particolare viene garantito il supporto alla versione corrente dello standard definito nel ANSI/ISO *SQL Standard*. [16]

Le caratteristiche principali di **MySQL** sono:

- Centouno connessioni contemporanee al database;
- Rapido accesso ai dati contenuti in database;
- Sicurezza dei dati contenuti nei confronti dgli accessi indesiderati e non autorizzati;
- Database multi-piattaforma;
- Disponibilità del codice sorgente.

2.1.1 Sviluppo di MySQL

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*, utilizzando in parte il codice di un altro *DBMS* **mSQL**, il quale supporta le connessioni TCP/IP ed un piccolo sottoinsieme di *SQL*.

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.

MySQL svolge il compito di *DBMS* nella piattaforma **LAMP**, una delle più usate e installate su Internet per lo sviluppo di siti e applicazioni web dinamiche. Nel luglio 2007

la società svedese **MySQL AB** ha 385 dipendenti in 265 paesi. I suoi principali introiti provengono dal supporto agli utilizzatori di **MySQL** tramite il pacchetto *Enterprise*, dalla vendita delle licenze commerciali e dall'utilizzo da parte di terzi del marchio **MySQL** .

Il 16 gennaio 2008 **Sun Microsystem** ha acquistato la società per un miliardo di dollari, stimando il mercato del database in 15 miliardi di dollari. E il 20 aprile di quest'anno (2009), **Sun Microsystem** e **Oracle Corporation** hanno annunciato, l'acquisizione della **Sun Microsystem** da parte di **Oracle Corporation**, per un valore di 7,4 miliardi di dollari [18], ciò significa che l'**Oracle** comprando la **Sun**, ha acquistato anche **MySQL**.

MySQL è scritto in **C** e **C++** e viene testato con un ampia gamma di diversi compilatori, inoltre lavora su diverse piattaforme come:

- **IBM AIX 4.x** e **5.x**
- **Amiga**
- **FreeBSD 5.x** e versioni superiori con *threads* nativi
- **HP-UX 11.x** e versioni superiori con *threads* nativi
- La maggior parte delle distribuzioni di **Linux**
- **Mac OS X**
- **NetBSD 1.3/1.4 Intel** e **NetBSD 1.3 Alpha**
- **Novell NetWare 6.0 e 6.5**
- **OpenBSD 2.5** con *threads* nativi e **OpenBSD** nelle versioni precedenti alla 2.5 con il *MIT-pthreads package*
- **SCO OpenServer 5.0.X** con il recente *porting* del **FSU** *Pthreads package*
- **SCO Openserver 6.0.x**
- **SCO UnixWare 7.1.x**
- **SGI Irix 6.x** con *threads* nativi
- **Solaris 2.5** e versioni superiori con i *threads* nativi su **SPARC** e **x86**
- **Tru64 Unix**

- **Windows 2000**, **Windows XP**, **Windows Vista**, **Windows Server 2003**, e **Windows Server 2008**

L'architettura di **MySQL** Server è multi-livello con moduli indipendenti, inoltre è completamente multi-*threads* con la possibilità di utilizzare più CPU qualora disponibili. Fornisce *Storage Engine* sia transazionali che non-transazionali, le modifiche fatte sugli *Storage Engine* non transazionali vengono effettuate immediatamente, mentre gli *Storage Engine* transazionali consentono di effettuare più modifiche e di convalidarle insieme prima delle modifiche effettive, e in caso di errore consente di tornare alla situazione precedente. Il motore originale di **MySQL** e' costituito da una serie di librerie *ISAM* per l'accesso ai dati. L'efficienza dell'implementazione, la stabilità delle *releases*, la possibilità di utilizzo in rete e la distribuzione **Open Source** ne hanno decretato l'enorme successo. Si tratta del piu' diffuso *RDBMS* per le applicazioni web.

MySQL permette al più centouno accessi contemporanei al database, un'estrema velocità di accesso ai dati al suo interno, un'elevata sicurezza al fine di evitare accessi indesiderati e non autorizzati e la disponibilità di personalizzare il codice sorgente.

Caratteristiche più significative introdotte nel corso degli anni nelle versioni di **MySQL** :

- La versione 4.0 ha introdotto le *UNION* e il sistema di *caching*;
- La versione 4.1 ha introdotto le SUBQUERY, il supporto per *OpenGIS*, un maggior numero di *storage engine* (prima di questa versione erano supportati solo *ISAM*, *MyISAM*, *INNODB*) e un supporto molto scalabile per la localizzazione;
- La versione 5.0 ha introdotto *STORED PROCEDURES*, VISTE (*View*), *Cursor*, TRANSAZIONI (*Transaction*), *TRIGGERS*, ulteriori *storage engine*; [17]

2.1.2 MySQL e i database distribuiti

Per il sistema di controllo dei sensori è necessario che il database sia distribuito, in modo che ogni wrapper abbia il suo database locale.

Per la gestione dei database distribuiti **MySQL** fornisce uno *Storage Engine* di tipo *federated*. È disponibile dalla versione 5.0.3. Si tratta di un *Storage Engine* che accede ai dati in tabelle di database remoti, piuttosto che nelle tabelle locali.

Per poter utilizzare *Storage Engine* di tipo *federated* è necessario attivare l'opzione *federated* nella configurazione del *server* di **MySQL**.

Le tabelle *federated* sono le tabelle con lo *Storage Engine* di tipo *federated*. E consentono di accedere da un server a una tabella su un altro in *server* remoto.

Quando si crea una tabella *federated*, viene creato solo il file "frm" sul *server* locale. Il file frm è il file che descrive il formato della tabella (ossia la definizione della tabella), e porta lo stesso nome della tabella, con una estensione .frm (test_table.frm nell'esempio). Ciò significa che sul *server* locale non viene fatto nessuna replica dei dati della tabella sul *server* remoto, e le operazione di accesso e modifica sono fatte direttamente sulla tabella del *server* remoto.

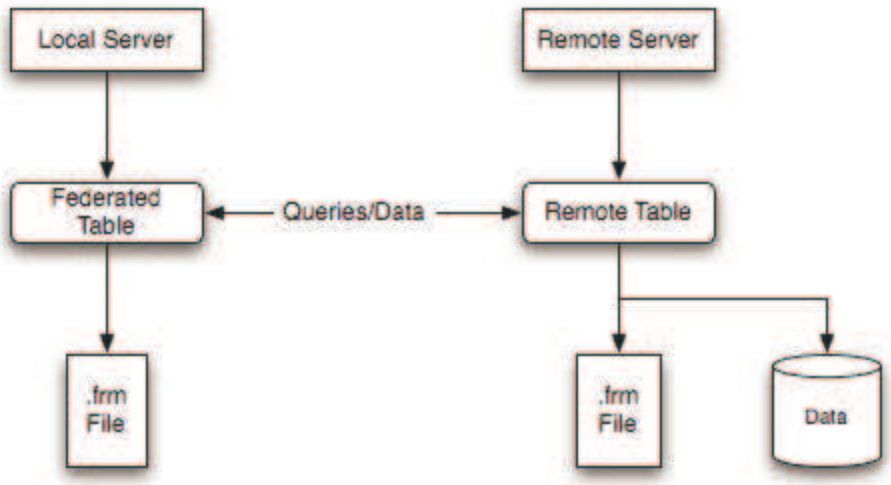


Figura 15 Schema di funzionamento delle tabelle federated

Esistono due modi di creare le tabelle *federated*:

- Attraverso le stringhe di connessione;

- Attraverso la creazione di una “definizione” del server nel database di **MySQL**.

2.1.2.1 Creazione di tabelle federated attraverso le stringhe di connessione

Per poter creare le tabelle di tipo federated è necessario, come detto in precedenza, attivare l'opzione *federated*, una volta definita la tabella sul server remoto da accedere. Es:

```
CREATE TABLE test_table (
    id     INT(20) NOT NULL AUTO_INCREMENT,
    name   VARCHAR(32) NOT NULL DEFAULT '',
    other  INT(20) NOT NULL DEFAULT '0',
    PRIMARY KEY  (id),
    INDEX name (name),
    INDEX other_key (other)
)
ENGINE=MyISAM
DEFAULT CHARSET=latin1;
```

Possiamo creare una tabella sul *server* locale, collegata alla tabella sul *server* remoto in questo modo:

```
CREATE TABLE federated_table (
    id     INT(20) NOT NULL AUTO_INCREMENT,
    name   VARCHAR(32) NOT NULL DEFAULT '',
    other  INT(20) NOT NULL DEFAULT '0',
    PRIMARY KEY  (id),
    INDEX name (name),
    INDEX other_key (other)
)
ENGINE=FEDERATED
DEFAULT CHARSET=latin1
CONNECTION='mysql://fed_user@remote_host:9306/federated/test_table';
```

La struttura della tabella sul server locale è identica a quella del server in remoto, solo che è necessario indicare la tipologia di *Storage Engine* (ENGINE=FEDERATED), e definire la stringa di connessione con il server remoto.

La composizione della stringa di connessione è la seguente:

scheme://user_name[:password]@host_name[:port_num]/db_name/tbl_name

Questo è un modo con cui si può creare un database distribuito direttamente con **MySQL**. Ma l'uso di questo metodo può essere. Sulla documentazione fornita dal sito di **MySQL** relativo alla tabella *federated* è riportato:

“L'uso della *CONNECTION* per specificare la stringa di connessione è non ottimale, ed è destinata a cambiare in futuro. Tenetelo a mente per le applicazioni che utilizzano tabelle *federated*. Tali applicazioni avranno probabilmente bisogno di modifiche.

Perché qualsiasi password inserita nella stringa di connessione viene memorizzato come testo normale, può essere vista da qualsiasi utente che può usare *SHOW CREATE TABLE* o *SHOW TABLE STATUS* sulla la tabella *federated*, o interrogare la tabella *Tables* del database INFORMATION_SCHEMA.”

2.1.2.2 Creazione di tabelle federated attraverso CREATE SERVER

É possibile creare le tabelle *federated* attraverso la “definizione” di un *server* remoto, per creare la definizione del server remoto si usa il comando *CREATE SERVER*, questo commando è disponibile solo dopo la versione **MySQL 5.1.15**, e ha la seguente sintassi:

```
CREATE SERVER server_name
    FOREIGN DATA WRAPPER wrapper_name
    OPTIONS (option [, option] ...)

option:
  { HOST character-literal
  | DATABASE character-literal
  | USER character-literal
  | PASSWORD character-literal
  | SOCKET character-literal
  | OWNER character-literal
  | PORT numeric-literal }
```

Le *option* sono le informazioni di connessione al server di cui si vuole creare la definizione.

Un esempio dell'uso di *CREATE SERVER* (per creare la stessa tabella *federated* del metodo precedente):

```
CREATE SERVER fedlink
FOREIGN DATA WRAPPER mysql
OPTIONS (USER 'fed_user', HOST 'remote_host', PORT 9306, DATABASE
'federated');
```

E la creazione della tabella sul server locale diventa:

```
CREATE TABLE test_table (
    id     INT(20) NOT NULL AUTO_INCREMENT,
    name   VARCHAR(32) NOT NULL DEFAULT '',
    other  INT(20) NOT NULL DEFAULT '0',
    PRIMARY KEY  (id),
    INDEX name (name),
    INDEX other_key (other)
)
ENGINE=FEDERATED
DEFAULT CHARSET=latin1
CONNECTION='fedlink/test_table';
```

Questo metodo può diventare utile quando si vuole creare molte tabelle *federated* con lo stesso server remoto, in questo modo è necessario definire solo una volta il *server* remoto.

Con la creazione della definizione di un server si inserisce un record nella tabella *mysql.server*. Tale record verrà usato per la connessione, quando si richiede di accedere ad una tabella remoto (*federated*) che ha come parametro di *CONNECTION* il server che è stato definito.

2.1.3 Stored procedure

Una *stored procedure* è un insieme di istruzioni in *SQL* memorizzate in database sotto un nome che le identifica. Tale nome può essere usato in seguito per eseguire le istruzioni memorizzato nella *stored procedure*.

MySQL ha introdotto le stored procedure solo dalla versione 5.0. Inoltre per eseguire le *stored procedure* con dei *set* di risultati (ottenuto ad esempio da una *select* all'interno della *stored procedure*) è necessario abilitare il *flag CLIENT_MULTI_RESULTS* durante la connessione con il *server*.

La sintassi per creare una *stored procedure* è la seguente:

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MySQL data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

routine_body:
    Valid SQL procedure statement
```

Per creare una *stored procedure* è necessario usare anche il comando *DELIMITER*, con il quale è possibile specificare un altro carattere per delimitare le istruzioni, essendo il separatore delle istruzioni (punto virgola) lo stesso usato per delimitare le istruzioni. Nel caso in cui il comando DELIMITER non venga eseguito, al primo punto e virgola (all'interno della *stored procedure*) il server interpreterebbe che il comando CREATE sia terminato. Scegliendo doppio *slash* come il nuovo carattere per delimitare, le istruzioni per la creazione dello stored procedure diventa:

```
DELIMITER //
-- definizione della stored procedure usando la sintassi di CREATE
DELIMITER ;
```

Come detto in precedenza, ogni procedura è identificata da un nome. Inoltre la procedura è associata ad un database (a partire da **MySQL 5.0.1**). Di conseguenza la procedura viene assegnata ad un database al momento della creazione, ed i nomi al suo interno sono riferiti allo stesso database, a meno che non venga specificato un altro database. Se durante la fase di creazione della procedura non viene specificato il nome di un database, la procedura viene associato di default al database su cui è stato eseguito il comando di creazione.

Per eseguire le *store procedure* si usa il comando *CALL*, il quale ha la seguente sintassi:

```
CALL sp_name([parameter[,...]])
```

Inoltre per eliminare una stored procedure viene usata il comando di *DROP*:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Infine esiste il commando *ALTER PROCEDURE*, usato per modificare una *stored procedure* già esistente:

```
ALTER PROCEDURE proc_name [characteristic ...]

characteristic:
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'
```

Molto spesso però, il commando di *ALTER PROCEDURE* non viene usato, si preferisce usare la combinazione del comando di *DROP* insieme al comando di *CREATE PROCEDURE*, per modificare una procedura.

2.2 MySQL++

MySQL++ è un'*application program interface* (*API*) **C++** per **MySQL**. Come tale si occupa di fornire una serie procedure che consentire a un programma in **C++** di interfacciarsi a un database **MySQL**, e di eseguire facilmente le query.

MySQL++ è stato creato da Kevin Atkinson nel 1998. L'obbiettivo iniziale era di creare una libreria per database indipendente dal *DBMS*, e di chiamarlo **SQL++**. Venne rilasciato la *release* pre-1.0. Successivamente nel 1999, la **MySQL AB**, proprietaria del codice di **MySQL**, ha ripreso lo sviluppo della libreria, e nel 2001 con la *release* 1.7.9, l'idea di avere una libreria multi-database morì. [19]

Attualmente la versione usata per implementare il progetto è la versione 3.0.9 rilasciato il 04/02/2009.

2.2.1 Funzioni principali delle librerie per database

Le funzioni principali per una libreria che si occupa di interfacciarsi a un database sono:

- Connessione al database;
- Esecuzione delle *query*;
- Se la *query* è stato eseguito con successo, elaborare i risultati, solitamente con delle iterazioni, ltrimenti riportare gli errori.

Tutte queste funzioni coincidono con una classe o una famiglia di classi della libreria MySQL++.

2.2.2 Classi principali di MySQL++

La connessione con il database viene gestita dagli oggetti della classe *Connection*. Per aprire una connessione con un database si usa la funzione Connection::connect() e per chiuderla si usa *Connection::disconnect*.

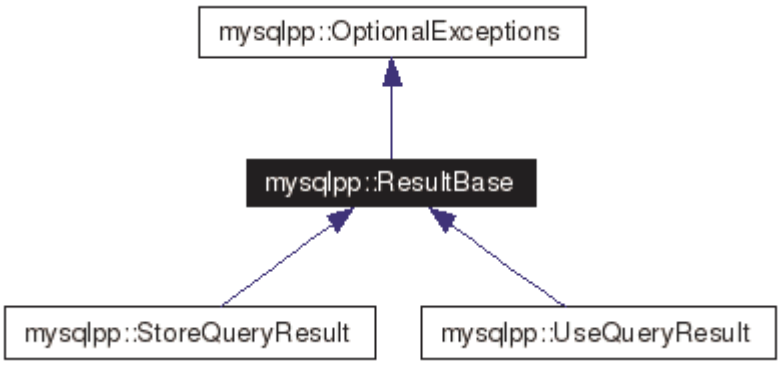
Come si può vedere dalla figura la classe *Connection* supporta 3 tipi di connessioni:

- *TCP/IP*
- *Unix Domain Socket*
- *Windows named pipes*

Dagli oggetti della classe *Connection* è possibile ottenere gli oggetti della classe *Query*, i quali si occupano della gestione delle *query*.
La gestione delle *query* comporta 2 funzioni principali, la creazione delle stringhe che rappresentano le *query* e l'esecuzione.
Per creare le stringhe delle *query*, la classe *Query* mette a disposizione diverse possibilità.

Un oggetto della classe *Query* può essere usato come un *output stream*, cioè è possibile inserire le stringhe da scrivere come all'interno di *std::cout* (es. cout << “hello word”;).
Oppure è possibile creare le stringhe con *Template Queries*, funziona in modo analogo alla funzione printf() di **C**. Si inserisce la stringa della *query* con alcuni tag che devono essere sostituiti con delle variabili che possono essere impostate successivamente attraverso la funzione *parse()* della classe Query. In questo modo se è necessario costruire molte *query* strutturalmente uguali ma che hanno dei variabili diversi questo può essere un metodo molto comodo.
Infine è possibile creare le *query* con *Specialized SQL Structures* (*SSQLS*). Questo metodo consente di creare delle strutture in C++ che hanno la stessa struttura delle tabelle in database, e di consentire all’utente di accedere alla tabella del database direttamente attraverso la struttura.

L’esecuzione delle *query* può avvenire in varie modalità a secondo della necessità dell’utente. A seconda della modalità di esecuzione produce tipi di risultati diversi, e che quindi devono essere immagazzinati in classi “*Result*” diverse.



La classe *ResultBase* è la classe di risultati che non contiene dati delle *query*, ma riporta solo i risultati dell’esecuzione (es. corretta esecuzione delle query, il numero di righe modificate, ecc...). Gli oggetti di questa classe possono essere usati nelle *query* che non producono valori di ritorno (es. *CREATE TABLE*), usando la funzione d’esecuzione *Query::execute()*.

La classe *StoreQueryResult* è una classe di risultati che ha la caratteristica di consentire l’accesso casuale ai suoi dati. Gli oggetti di questa classe sono creati attraverso la funzione d’esecuzione *Query::store()*.

La classe *UseQueryResult* è la classe di risultati che non consente l'accesso casuale sui dati. L'accesso ai dati degli oggetti *UseQueryResult* deve essere sequenziale, ciò significa che questi oggetti sono memorizzati in maniera più efficiente, e non hanno bisogno di essere memorizzati nella RAM. Gli oggetti di questa classe possono essere molto utili quando si ha una grande quantità di dati prodotti dall'esecuzione delle *query*. Gli oggetti della classe *UseQueryResult* sono creati dall'uso della funzione d'esecuzione *Query::use()*.

Infine i risultati delle *query* possono essere memorizzati nelle *Specialized SQL Structures* (*SSQLS*), come accennato in precedenza nella costruzione delle *query*. Questo metodo consente di memorizzare i dati risultati dalle *query* direttamente in strutture dati create dal programmatore.

Capitolo 3: Architettura del sistema di controllo

L'obbiettivo del sistema in analisi è quello di rendere efficiente la raccolta dati da reti di sensori. Ovviamente la rete è composta da più sensori che comunicano con più *host* su cui sono installati degli A*ccess Point* (*AP*). Ciò significa che occorre un sistema distribuito con un server centrale che gestisce la raccolta dei dati.

Il problema maggiore dei sensori *wireless* è l'alimentazione del dispositivo, sia quando vengono alimentati con le batterie sia con un dei sistemi di alimentazione autonomi, ad esempio l'energia solare. Quindi ridurre i consumi al minimo è uno degli obbiettivi principali del sistema, questo significa ridurre il più possibile la comunicazione tra i sensori, in quanto il maggior consumo di energia è dovuto alla comunicazione tra i nodi.

Per realizzare il progetto si è scelto di suddividere il sistema in vari moduli che hanno funzioni specifiche, e che insieme riescono a realizzare le funzionalità del sistema che si vuole realizzare. Questo approccio rende il sistema realizzato robusto e semplice da comprendere e da modificare.

In questo capitolo si presenterà la suddivisione in moduli del sistema di controllo dei sensori, le funzionalità dei vari moduli e le loro interazioni.

3.1 Funzionalità del sistema

Come si è detto nell'introduzione l'obbiettivo del sistema è di rendere efficiente l'uso delle reti di sensori *wireless*. Il che significa ridurre al minimo la comunicazione con i sensori, che è la causa principale del consumo di potenza, senza tuttavia perdere il rilevamento dei dati significativi.

Per realizzare l'obbiettivo si è pensato ad un sistema che a seconda delle necessità possa regolare la frequenza dei rilevamenti. Quindi si è pensato ad un metodo che sia capace di creare o modificare delle *query* di rilevamento, attraverso le quali è possibile:

- Selezionare i dispositivi della rete di sensori coinvolti;
- Dare una frequenza di rilevamenti;
- Possibilità di richiedere che venga segnalato il superamento di certe soglie di rilevamento, gli eventi (che possono essere temperatura minima e/o massima ecc..);
- Possibilità di fermare o cancellare una *query* di rilevamento;
- Selezionare i dati raccolti dalla *query* di rilevamento.

Il sistema inoltre deve essere in grado di supportare la possibilità di avere più *query* di rilevamento contemporaneamente.

Per poter interagire con il sistema è necessario creare una modalità d'interazione in grado di controllare le *query* di rilevamento, si è pensato di realizzare un set di istruzioni *pseudo-SQL*, che verrà chiamato *Sensor Query* (*SQ*) o *query* di controllo, con i quali sarà possibile interagire con il sistema.

Il set di *SQ* fondamentali per il controllo del sistema attualmente sono:

- *Store*, che consente la creazione di una nuova query di rilevamento, specificando il nome della *query* di rilevamento, i dispositivi (sensori) che devono effettuare i rilevamenti, la frequenza dei rilevamenti e i eventi degli rilevamenti;
- *Alter*, col quale è possibile modificare una *query* di rilevamento già creata, specificando il nome della *query* di rilevamento e le modifiche da apportare;

- *Drop*, attraverso il quale è possibile eliminare una *query* di rilevamento;
- *Stop*, è l'istruzione usato per fermare il rilevamento di una query di rilevamento;
- *Restart*, usato per far ripartire una query di rilevamento fermato dall'istruzione *stop*;
- *Select*, comando per selezionare i risultati dei rilevamenti.

La sintassi delle *SQ* sarà illustrata in modo più approfondito nel capitolo dedicato al *Query Parser*.

Il sistema realizzato sarà controllato inizialmente da un utente umano che a seconda dei rilevamenti attuali decide la modifica o inserimento dei *query* di rilevamento. E successivamente il sistema potrà essere controllato da un modulo di controllo automatico che a seconda delle necessità crea automaticamente le *SQ* per controllare il sistema.

Le funzionalità del sistema attualmente sono:

- Ricevere istruzioni dall'utente, attraverso delle *SQ* che saranno interpretato dal sistema;
- Configurare il sistema secondo le *SQ* ricevute dall'utente;
- Restituire i dati richiesti dall'utente con le *SQ* (nel caso in cui la *SQ* sia una *select*);
- Segnalare particolari eventi richiesti nelle *query* di rilevamento.

Per realizzare queste funzionalità si è pensato di usare un sistema di controllo con dei database distribuiti. Nel database centrale, sono contenuti le informazioni del sistema (es. la posizione dei sensori, degli host ecc..). Nei database distribuiti sui singoli *host wrapper* (*host* con *AP*), in cui salvare i dati rilevati, sono contenute le configurazioni dei sensori (es. frequenza) e gli eventi da segnalare al sistema.

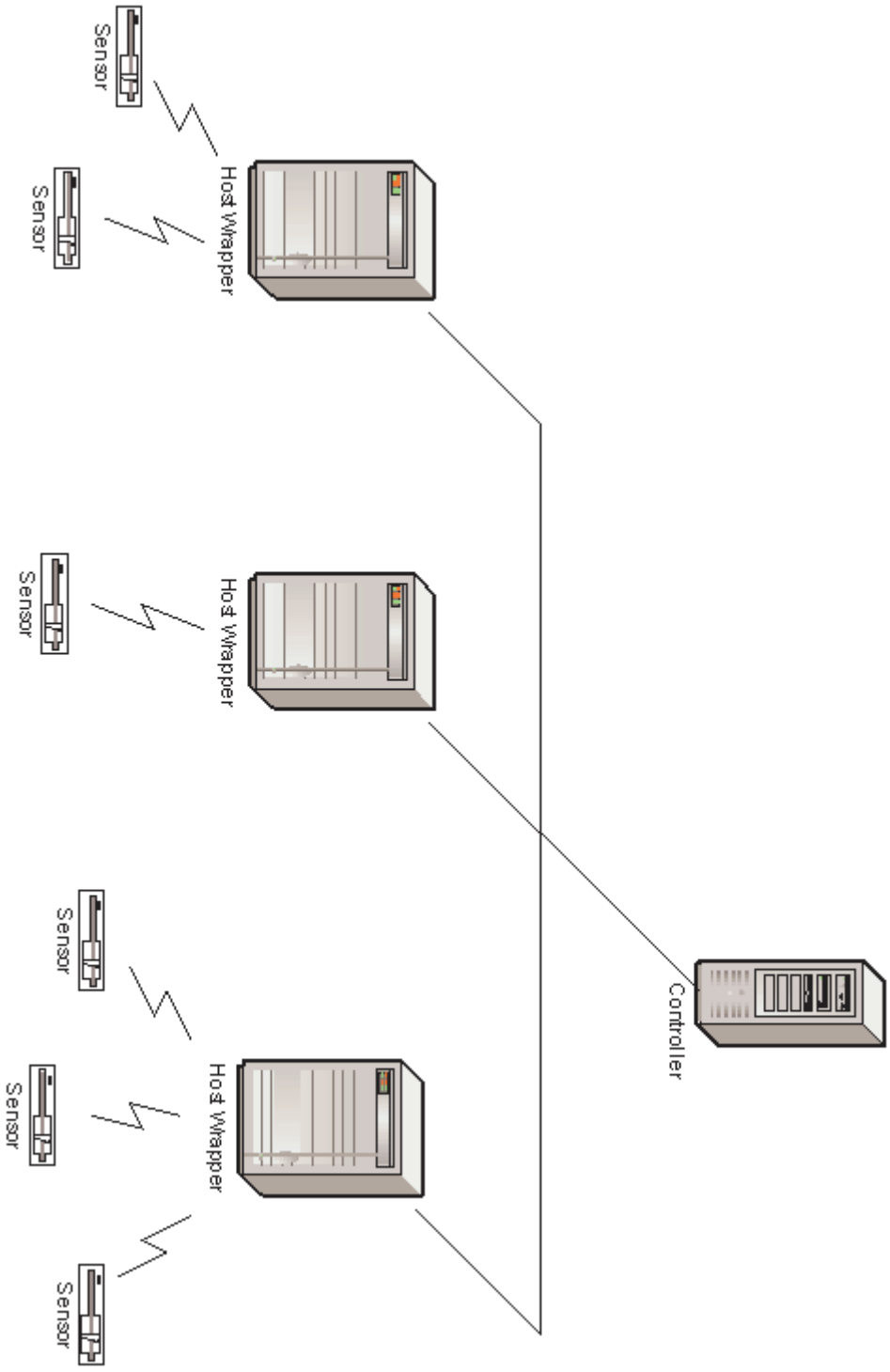


Figura 16 Architettura del sistema

3.2 Componenti del Sistema

Lo schema generale del sistema semplificato con un singolo *host wrapper* è il seguente (Figura 17):

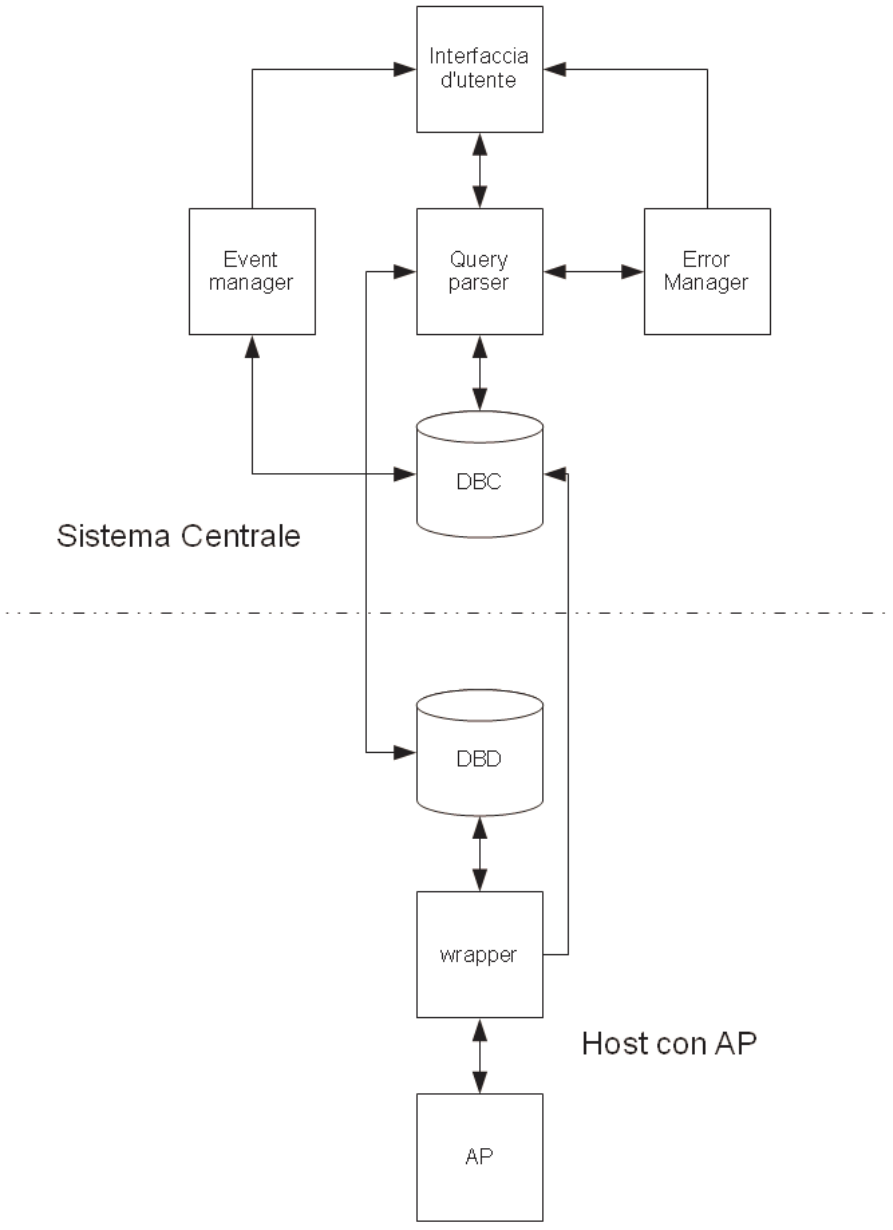


Figura 17 Schema a blocchi dei componenti

Questo schema mostra la i moduli necessari per realizzare il sistema e l'interazione degli moduli stessi con i database. La comunicazione tra il sistema centrale e l'*host wrapper* avviene direttamente nel *database*. Il modulo *Query Parser* si connette al database distribuito per inserire o modificare le configurazioni dei sensori, eventuali eventi da segnalare e richiedere i dati rilevati. Il modulo *Wrapper* invece segnala al sistema centrale gli eventi rilevati, inserendo eventi nel database distribuito.

Nei paragrafi successivi verranno illustrate le principali funzioni dei componenti del sistema.

3.2.1 Interfaccia d'Utente

Il modulo Interfaccia Utente è il componente attraverso il quale l'utente inserisce le *query* per configurare il sistema e richiedere i dati raccolti. Inoltre l'Interfaccia Utente deve visualizzare eventuali dati richiesti, gli eventi segnalati dal sistema. Nello schema generale l'Interfaccia Utente è stata collocata nel sistema centrale, ma nella realizzazione pratica non è detto che sia completamente vero, in quanto il sistema centrale è su un server, mentre l'interfaccia utente deve risiedere, o almeno una parte su dei computer *client* che dialogano con il sistema centrale. Una possibile implementazione dell'interfaccia utente potrebbe essere un'applicazione con un'interfaccia *web*, o comunque un'applicazione *client-server*.

In una fase più avanzata del progetto, l'interfaccia d'utente dovrà essere sostituita o affiancata da un componente di controllo automatico, che a seconda degli eventi o dei dati raccolti, deciderà la configurazione dei sensori.

3.2.2 Query Parser

Il modulo *Query Parser* è il componente che si occupa di:

- controllare la sintassi e fare interpretare le *query* inserite attraverso l'interfaccia utente.
- Eseguire le *query* interpretate.

Le *query* possono richiedere una modifica del database delle configurazioni dei sensori o di fornire raccolti dati nei database.

Nel primo caso il *Query Parser* dopo aver controllato la sintassi ed avere interpretato la *query*, deve raccogliere i dati sui database da modificare (le configurazioni dei sensori sono sui database distribuiti) e lanciare le procedure di modifica su ogni database.

Nel secondo caso, il *Query Parser* deve controllare la sintassi, interpretare la *query*, raccogliere le informazioni della *query* (su quali host ci sono i dati necessari), lanciare le procedure per raccogliere i dati sui i database distribuiti , fondere i risultati e mandarli all'utente.

L'implementazione del *Query Parser* è lo scopo principale di questa tesi e verrà trattato in modo più approfondito nel capitolo successivo.

3.2.3 Wrapper

Il modulo *Wrapper* è il componente installato sui *host wrapper* che si occupa di comunicare con i sensori attraverso AP e controllare il database locale. Le sue funzioni principali sono:

- Controllare lo stato dei sensori;
- Configurare i sensori ogni volta che c'è un cambiamento nella tabella della configurazione dei dispositivi all'interno del database distribuito;

Inserire i dati rilevati dagli sensori nel database locale;

Inserire nel database centrale eventuali eventi rilevati.

Un'implementazione del modulo *Wrapper* è già stato realizzato in una tesi dal ing. Daniele Caiti [3].

3.2.4 Event manager

Il modulo che si occupa di segnalare gli eventi all'interfaccia utente è il modulo *Event Manager*; come si è detto in precedenza quando si rileva un evento, l'*host wrapper* del sensore che rileva l'evento lo segnala al sistema centrale inserendo un record relativo all'evento nel *database*, perciò l'*Event Manager* non è altro che un'interfaccia fra il database e l'interfaccia utente.

Si è pensato a due modi per realizzare l' *Event Manager*, il primo è di usare un *trigger* nel database del sistema centrale, e il compito del trigger sarà notificare l'evento all'interfaccia d'utente. La seconda soluzione è di realizzare un programma che interroghi la tabella degli eventi periodicamente e segnali l'evento quando ci sono nuove righe inserite.

Entrambi modi per realizzare il modulo hanno vantaggi e svantaggi. Il primo metodo ha il vantaggio di essere rapido ed è un approccio molto simile all'*interrupt*, ma ha lo svantaggio della complessità implementativa, in quando è piuttosto complesso interfacciare un *trigger* con dei programmi esterni. Il secondo metodo è più semplice da realizzare, essendo semplicemente un programma che si interfaccia con il database e che lo interroga periodicamente, però ha lo svantaggio di non poter segnalare l'evento immediatamente quando si verifica, ma solo nel periodo di controllo, è un approccio di tipo *polling*.

3.2.5 Error manager

Il modulo *Error Manager* si occupa di gestire gli errori avvenuti durante l'esecuzione delle *SQ* da parte del *Query Parser*. Ci possono essere vari cause d'errori, ad esempio:

- Errore nelle informazioni di connessione;
- Connessione interrotta;
- *Host Wrapper* non più disponibile;
- Ecc..

L'*Error Manager*, in caso d'errori, deve cercare di correggere l'errore se è possibile, lanciando delle *SQ* speciali usate nei casi d'errori, altrimenti procedere a una procedura di *roll back*, per far in modo che la situazione ritorni allo stato precedente esecuzione della *SQ* che ha provocato l'errore, poi notificare l'errore avvenuto all'interfaccia utente.

3.3 Database

Come spiegato in precedenza i database sono una parte fondamentale del sistema. In questo paragrafo verrà illustrato, suddiviso in vari blocchi, il database del sistema per comprendere le funzioni dei vari blocchi .

Informazioni necessarie nel database sono:

- Informazioni sui sensori e come raggiungerle (*host wrapper* a cui è collegato);
- Dati rilevati rilevati dai sensori;
- Le *query* di rilevamento, con le relative informazioni di configurazione e degli eventi.

Ovviamente queste informazioni sono distribuite su vari database distribuiti e centrale e i vari moduli spiegati precedentemente servono ad agire sui vari database del sistema a seconda delle funzioni richieste.

3.3.1 Device

Per controllare il sistema di sensori è necessario avere le informazioni della collocazione dei vari sensori, ed interrogando il database del sistema centrale deve essere possibile conoscere con quali *Host Wrapper* è possibile interagire con determinati sensori.

Il seguente schema *E/R* (*entity relationship*) mostra una possibile struttura per contenere i dati relativi ai sensori.

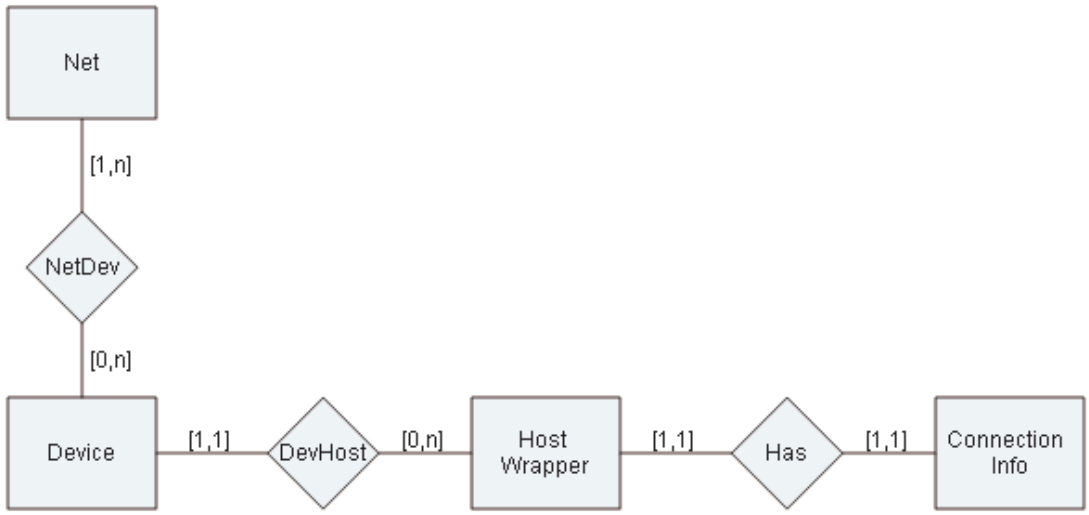


Figura 18 Schema E/R del database per memorizzare le informazioni sui device

I *device* sono i sensori e sono identificati univocamente da una numerazione univoca per ogni *host wrapper*, inoltre si è pensato di assegnar loro un nome univoco per identificarle in modo univoco in tutto il sistema. Per ogni *host* è necessario avere le informazioni di connessione al database del *host wrapper*. Per identificare univocamente gli Host Wrapper si è pensato di usare il nome del *host* stesso.
Inoltre i sensori possono essere raggruppati in gruppi di sensori (net), che sono identificati da un NetID univoco e un nome del gruppo.

Nella pratica per questa struttura si sono create 4 tabelle:

- *Device*, che contiene ID del sensore, il nome dell'Host Wrapper a cui appartiene, in questo modo si ha un identificativo univoco. Inoltre nella tabella è contenuto anche il nome univoco del sensore;
- *ConnectionInfo*, che contiene le informazioni di connessione al database del *Host Wrapper*, ha come identificativo univoco il nome del *Host Wrapper*;
- *Net*, la tabella che contiene le informazioni dei gruppi di sensori, che sono il nome del gruppo e l'ID del gruppo
- *NetDev*, è la tabella che associa i sensori ai gruppi (*Net*), ogni sensore può far parte di più gruppi, così come ogni gruppo può avere uno o più sensori.

3.3.2 Query di Rilevamento

Le query di rilevamento sono delle istruzioni di configurazione del sistema, con le quali si impostano le frequenze di rilevamenti, i sensori, e quindi gli H*ost Wrapper* coinvolti (si veda lo schema E/R che segue).

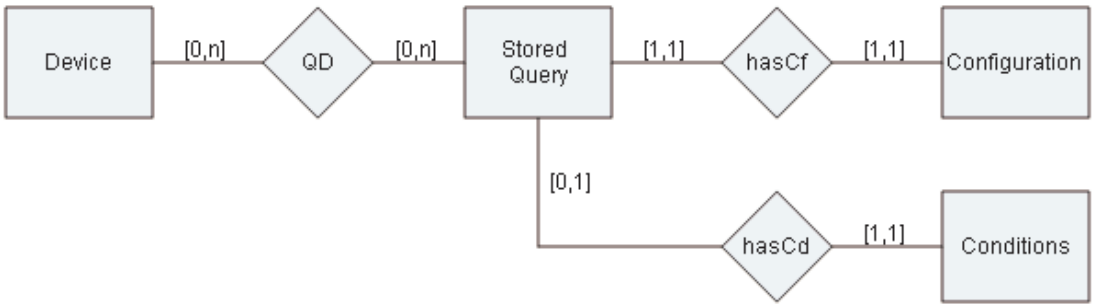


Figura 19 Schema E/R del database per memorizzare le informazioni sulle query di rilevamento

Le query di rilevamento memorizzate (Stored Query) sono identificate da un numero univoco e da un nome dato dall'utente. Le varie informazioni contenuto nella *query* di rilevamento vengono memorizzate in maniera separata nel database distribuito, in quando queste informazioni servono per gestire parti diversi del sistema.
Le informazioni memorizzate nel database centrale riguardano le informazioni generali, il

nome della *query* di rilevamento, i sensori e quindi gli *host wrapper* coinvolti, in questo modo interrogando il database centrale è sempre possibile sapere su quali database degli *Host Wrapper* è necessario accedere per operare con una determinata *query* di rilevamento. Mentre le informazioni nei database degli *host wrapper*, riguardano i parametri di configurazione dei sensori, frequenza di rilevamento richiesti, gli eventi da segnalare, oltre ovviamente quali sensori devono fare i rilevamenti. Inoltre tra gli attributi della Configuration c'è anche l'informazione sullo stato delle Query di rilevamento, cioè se al momento sono attive o no, è possibile impostare questo parametro attraverso *SQ start* e *stop*, e indica se è necessario continuare a fare dei rilevamento per la *query* di rilevamento.

3.3.3 Dati rilevati

I dati rilevati sono memorizzate nei database distribuiti, quando un utente richiede i dati, con un commando di *select* della *query* di rilevamento, il modulo di Query Parser richiede al database centrale quali *host wrapper* sono da interrogare per recuperare i dati, poi per ogni singolo *host wrapper* vengono richiesti i dati della *query* di rilevamento.
I dati rilevati quindi devono avere almeno le seguenti informazioni:

- L'identificativo univoco del sensore che ha rilevato il dato;
- L'identificativo della *query* di rilevamento che ha richiesto il rilevamento;
- Il momento in cui il dato è stato rilevato.

Figura 20Schema E/R del database per memorizzare i valori rilevati

Capitolo 4: Implementazione Query Parser

Il lavoro principale di questa tesi è l'implementazione del modulo *Query Parser*. Come già spiegato nel capitolo precedente, questo modulo ha il compito di ricevere le istruzioni attraverso le *Sensor Query* (*SQ*), controllarne la sintassi, interpretarla e poi eseguirla sul database centrale e sui database distribuiti.

Uno degli obbiettivi principali che si è posto nella realizzazione di questo modulo è di rendere il componente flessibile, in modo da poter aggiungere facilmente nuove *SQ*, senza dover modificare il codice del modulo *Query Parser* o almeno doverlo modificare il meno possibile. Questo perché il progetto è solo agli inizi ed è impossibile prevedere tutte le istruzioni che il sistema potrebbe aver bisogno in base a future applicazioni di reti di sensori. Per evitare le modifiche del codice per aggiungere nuove istruzioni può rendere il codice ingestibile, soprattutto se le modifiche vengono fatte da diverse persone e molto distanti nel tempo.

In questo capitolo di illustrerà l'approccio usato per implementare il modulo *Query Parser*, le classi e funzioni create per implementare il modulo, inoltre verrà spiegata nel dettaglio anche le struttura delle *SQ* finora ipotizzate. Infine si mostrerà con un esempio come è possibile inserire una *SQ* nel sistema.

4.1 Principio di funzionamento del Query Parser

Il modulo Query Parser come detto in precedenza ha 2 funzioni:

- Controllare la struttura sintattica delle *SQ* in ingresso ed estrarre i parametri delle *SQ*;
- Eseguire le procedure appropriate per ogni *SQ* sui vari database.

Inoltre si è posto l'obbiettivo di realizzare un componente flessibile che riesca a gestire inserimento di nuove *SQ*, senza modificare il codice del componente.

4.1.1 Analisi delle SQ

La soluzione adottato è quella di creare un analizzatore di *query* generico in grado di analizzare qualunque *query* che abbia una struttura a *sub-query* ricorsivo attraverso il suo *template* ricorsivo.

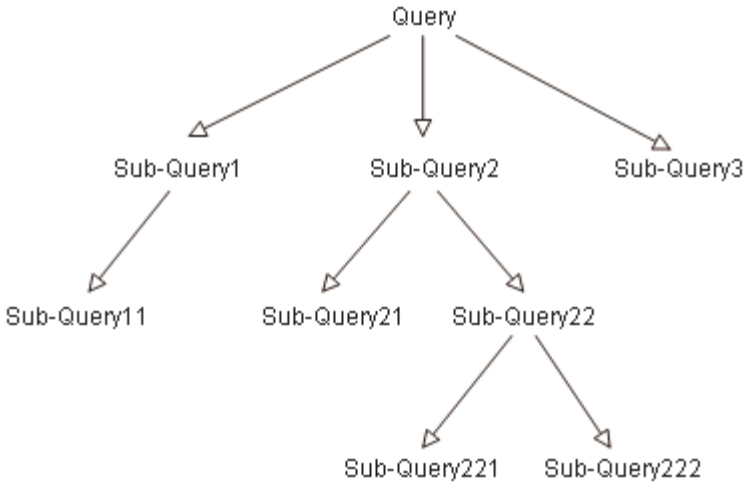


Figura 21 Struttura delle query

Una *query* con una struttura a *sub-query* ricorsiva è una *query* che può essere scomposta in vari *sub-query*, che a loro volta possono essere scomposta in *sub-query* fino ad avere una forma atomica.

La **forma atomica** è una composizione di una parola chiave (*key word*) ed un insieme di valori dei parametri (*values*).

Possiamo schematizzare le *query* in questo modo:

Nella struttura riportata in figura le S*ub-Query*11, S*ub-Query*21, S*ub-Query*221, S*ub-Query*222 e S*ub-Query*3 sono *sub-query* in forma atomica, da cui è possibile estrarre i parametri della *Query*.

Questa struttura può essere vista come struttura a livelli, *Query* è al livello 0, S*ub-Query*1, S*ub-Query*2, S*ub-Query3* sono al livello 1 ecc.. E i corrispondenti *template* avranno gli stessi livelli.

Per analizzare le *query* strutturate in questo modo abbiamo bisogno dei *template* anch'essi con una struttura ricorsiva, in modo da poter fare il confronto delle parole chiave e quindi ricavarne la struttura della *query* in ingresso. Per ogni *query* è necessario definire uno specifico *template*, che a sua volta sarà composto dai *sub-template*, fino alla forma atomica.

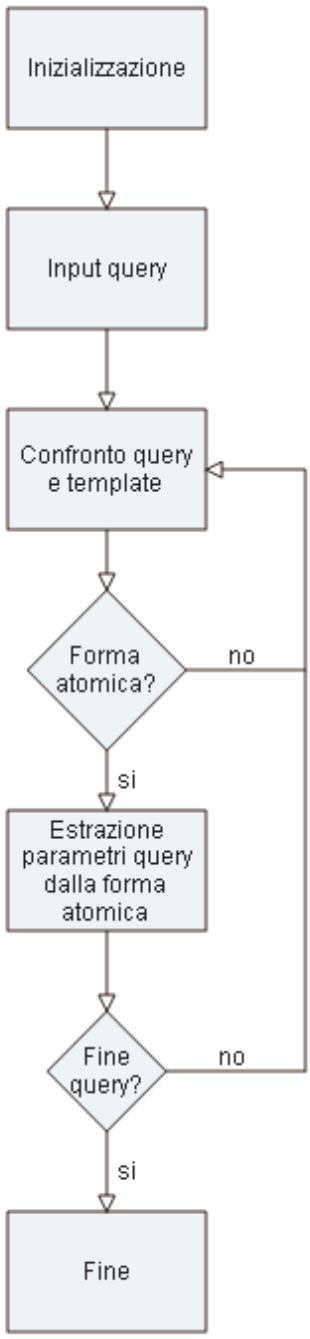


Figura 22 Diagramma di flusso per la suddivisione delle sub-query

Durante la fase di analisi viene fatto un confronto tra la *query* e il suo *template*, il confronto consiste nel cercare le parole chiave delle *sub-template* del *template* nella *query* in ingresso. In questo modo è possibile scomporre la *query* in *sub-query*, usando come separatori le parole chiave (dette anche *Begin Word*). Lo stesso procedimento viene ripetuto in modo ricorsivo fino a trovare la forma atomica. Una volta trovata la forma

atomica, si estraggono i valori (escludendo la parola chiave che li procede) e li si memorizza in una classe appropriata.

4.1.1.1 Attributi dei sub-template

Esistono vari attributi nei *sub-template* che servono per descrivere caratteristiche delle relative sub-*query* e sono:

- *Valore*, per indicare se nella *sub-query* relativa ci sono dei valori dei parametri, questo consente di creare dei *sub-query* anche senza valori nella forma atomica;
- *Atomica*, indica se la relativa *sub-query* è in forma atomica o no;
- *Opzionale*, indica se la *subquery* è opzionale od obbligatorio, nel primo caso una *query* può anche non contenere la relativa *sub-query*, mentre nel secondo caso l'assenza della *sub-query* relativo al *sub-template* è considerato un errore (es un utente che non mette la sub-query del nome della SQ store);
- *Ripetuta*, indica se la *sub-query* è ripetuta o no, in caso affermativa, significa che la *sub-query* può essere presente più volte nella stessa *query*.

4.1.2 Esecuzione delle SQ

Le *SQ* sono istruzioni usate per controllare i sensori, i quali comunicano solo con i *Host Wrapper* su cui sono installati gli *AP*, e che hanno tutti un database locale. Il che significa che la maggior parte delle *SQ*, se non tutte, sono delle istruzioni che agiscono su più database. Quindi è necessario trovare un metodo per agire su più database distribuiti sui singoli *Host Wrapper*.

L'idea iniziale è stata quella di sfruttare la struttura e le funzioni dei database distribuiti forniti direttamente dal **MySQL**, le tabelle con *engine federated*. Il problema di questo

approccio è il fatto che non è possibile creare le tabelle *federated* né con delle stringhe di connessione né con la definizione dei server remoti. In quando non consentono di definire delle connessioni in maniera dinamica. Il che significa che per ogni *database* distribuito è necessario creare in maniera permanente una serie di tabelle *federated*, Ma ciò può essere molto oneroso e complesso dal punto di vista della manutenzione.

La soluzione adottata è quella di eseguire 2 procedure dal QueryParser:

- La prima procedura viene eseguita sul database centrale, serve per recuperare dati di connessione e le procedure che verranno eseguite in seguito;
- Una volta recuperato i dati di connessione necessari, è possibile eseguire le procedure ricavate, che saranno già parametrizzate dalla prima procedura.

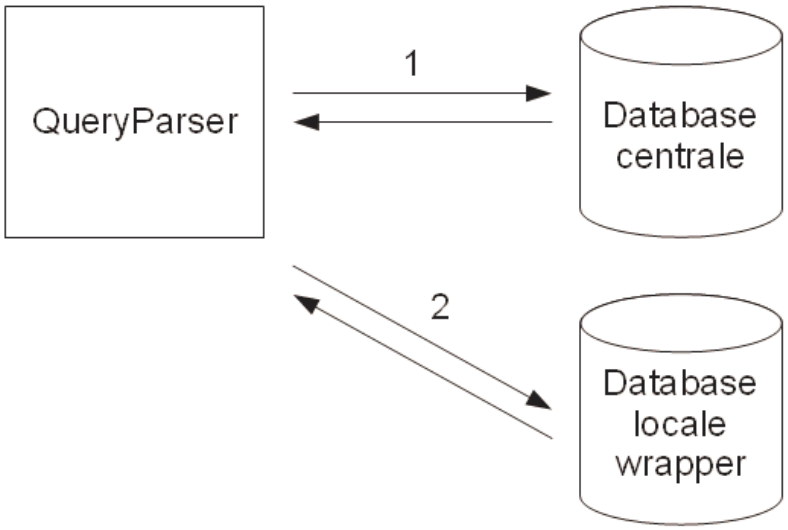


Figura 23 Schema d'esecuzione delle SQ

In questo modo quando si cambiano i parametri di connessione di un *Host Wrapper* è sufficiente aggiornare la tabella delle informazioni di connessione nel database centrale.

4.2 Classi realizzate per il Query Parser

Questo paragrafo esporrà le classi create per realizzare il modulo *Query Parser*.

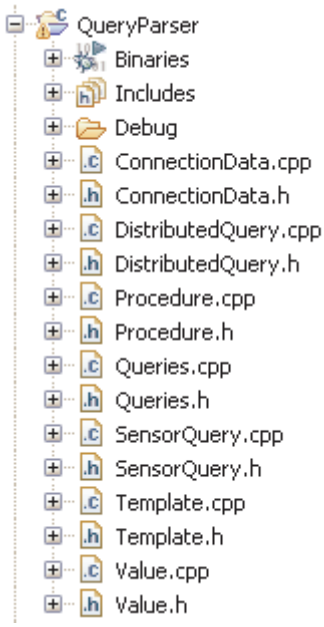


Figura 24 Classi create per il QueryParser

Per realizzare il componente *Query Parser* sono state create 7 classi di oggetti, ognuno dei quali con funzioni specifiche, e sono:

- *Template*, è la classe usata per contenere le informazioni dei *template* delle *SQ*;
- *Queries*, classe di oggetti usata per suddividere l'insieme di *SQ* inserito dall'utente;
- *SensorQuery*, usata per processare le SQ singole, questa classe ha sia le funzioni di *parse* che quelle di esecuzione delle SQ;
- *Value*, è la classe di oggetti usata per contenere i parametri estratti dalle *SQ*;
- *ConnectionData*, questa è una classe creata per contenere le informazioni di connessione usate per definire i parametri di connessione con il database;
- *Procedure*, è la classe usata per memorizzare le procedure usate per eseguire le *SQ*;
- *DistributedQuery*, una lista di questa classe viene usata per contenere tutte le *query* distribuite che devono essere eseguite per la *SQ*.

4.2.1 Template

La classe *template* è la classe di oggetti usata per contenere i *template* delle *SQ*.

Questa classe è una classe di oggetti "ricorsivi", nel senso che in un oggetto della classe *template* è presente come una variabile interna una lista dei oggetti della classe *template*. Le istanze rappresentano l'insieme di *sub-template* che compongono l'oggetto *template* "padre", e a loro volta gli oggetti che rappresentano i *sub-template* possono avere anche loro dei *sub-template*. Per facilitare la consultazione di template e dei sub-template sono state create alcune funzioni per "scorrere" i sub-template, i quali consentono la lettura sequenziale e il *reset* dei sub-template (tornare all'ordine iniziale).

Inoltre, visto che un oggetto *template* può contenere una lista di *template* si è pensato di caricare tutti i *template* delle *SQ* in un unico oggetto *template*, il quale contiene l'insieme di *template* che viene usato. Per facilitare la ricerca di un templare specifico è stata creata una funzione di ricerca dei *template* per parola chiave (solitamente viene usato per cercare *begin word* di una *SQ* per trovare il *template* appropriato da usare) all'interno della classe template.

All'interno della classe *template* esistono altre proprietà oltre alla lista dei *sub-template*, che sono utili nell'analisi delle *SQ*, sono proprietà riferiti alle *sub-query* rappresentate dai *sub-template* e sono:

- *getBegin()*, è la parola iniziale della *sub-query* ed è anche la parola chiave che rappresenta il *sub-template*:
- *isOptional*(), indica se il s*ub-query* è opzionale;
- *hasValue*(), indica se il *sub-query* contiene un valore;
- *isRepeated*(), indica se il *sub-query* è ripetuto;
- *isMinimal()*, indica se il sub-query è atomica.

Il caricamento dei *template* è un'operazione che viene eseguita nell'inizializzazione del modulo. Inizialmente l'intenzione era quella di caricare i *template* attraverso un file xml,

contenente le informazioni sulla struttura dei *template* stessi. La struttura dei file *xml* è particolarmente adatta a questa funzione in quanto ha una struttura flessibile e può contenere annidamenti. Poi ho pensato che in realtà la stessa struttura era realizzabile anche con un database relazionale e visto che il progetto usa già la libreria di **MySQL++** per interfacciarsi a un database **MySQL**, sono arrivato ad una soluzione relazionale. In conclusione, il caricamento dei *template* è stato realizzato attraverso un caricamento da una database *MySQL* la cui struttura verrà descritta nel paragrafo dedicato al database dedicato del *Query Paser*.

Essendo l'oggetto della classe *template* del tipo ricorsivo, anche il caricamento del *template* è una funzione ricorsiva. La funzione del caricamento dei *sub-template* fa le seguenti operazioni:

- Attraverso una *query* al database dei *template* ricava tutte le informazioni riguardanti ai sub-*template* dell'oggetto *template* in esame (oggetto che ha chiamato la funzione del caricamento dei *sub-template*);
- Per ogni *sub-template* viene creato un oggetto della classe *template* e viene aggiunto alla lista dei *sub-template* dell'oggetto che invoca la funzione del caricamento del *sub-template*;
- Per ogni oggetto *template* creato (i nuovi sub-template creati), viene invocata la loro funzione di caricamento dei *sub-template*.

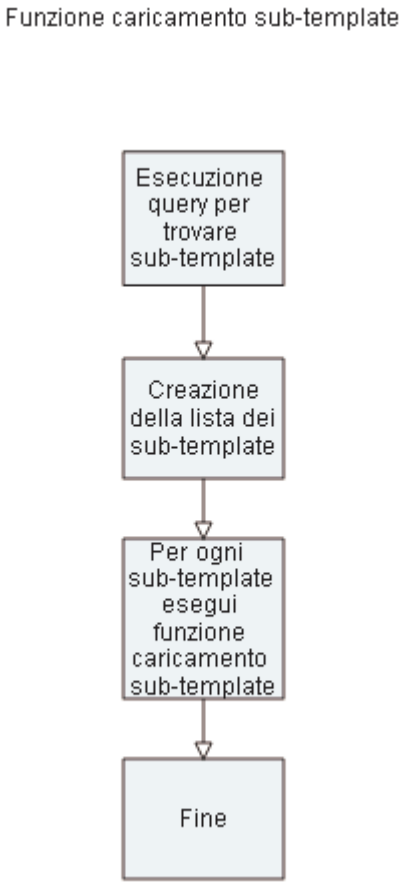


Figura 25 Diagramma di flusso del caricamento dei sub-template

In questo modo si riesce a caricare tutti i *sub-template* in modo ricorsivo. Assumendo che il *template* di partenza sia un *template* vuoto, e si richiamando la funzione di caricamento dei *sub-template*. Esiste una condizione particolare, fatta appositamente per caricare i template vuoti, invece di usare la *query* per caricare i *sub-template*, usa una *query* per caricare tutti i *template* di livello 0, che sono il punto di partenza di ogni *template*.

È necessario fare attenzione l'ordine nella lista dei *templat*, in quanto l'ordine dei *sub-template* deve essere corretto visto che l'analisi delle *query* è sequenziale. È importante dunque assegnare un numero di sequenza ad ogni *sub-template* e durante il caricamento nella *query* per ricavare le *sub-query* è necessario che il risultato venga ordinato per il numero di sequenza dei *sub-template* (ad esempio attraverso una clausola di *order by* all'interno della *query* di caricamento) per avere così una corretta analisi delle SQ.

4.2.2 Queries

Gli oggetti della classe Queries rappresentano l'insieme di *SQ* inseriti dall'utente.

Questa classe ha 4 funzioni principali:

- Suddividere l'insieme di SQ inserito dall'utente in SQ singole creando gli oggetti della classe *SensorQuery*, e inserire questi oggetti all'interno di una lista di *SensorQuery*;
- Assegnare ad ogni *SensorQuery* un *template* appropriato;
- Lanciare la funzione di parse per ogni oggetto *SensorQuery* creato;
- Eseguire la lista di *SensorQuery* creato e analizzato.

Per eseguire queste funzioni è stato necessario:

- Definire un separatore di *SQ* inserito dall'utente per poter distinguere una SQ dalla successiva, come separatore delle *SQ* si è scelto il simbolo ";" usato già in molti linguaggi come separatore delle istruzioni (es. **MySQL**, **C++** ecc..);
- Avere all'interno dell'oggetto *Queries* l'insieme dei *template* facilmente utilizzabile, ciò significa che è necessario avere come variabile interna un oggetto *template* contenete l'insieme di tutti i *template* utilizzati, come già descritto nel paragrafo precedente. Inoltre, come detto in precedenza, l'oggetto Queries dispone di una funzione di ricerca di *template* per parola chiave (*begin word*), che è molto utile per la ricerca del *template* corretto da assegnare alle *SQ*;
- Definire le funzioni di parse ed esecuzione all'interno della classe *SensorQuery*, che verranno spiegate nel paragrafo successivo relativo alla classe *SensorQuery*;

Le prime 3 funzioni della classe Queries (la suddivisione delle SQ con la creazione della lista degli oggetti *SensorQuery*, l'assegnamento dei *template* agli oggetti *SensorQuery* e l'analisi) vengono eseguite direttamente nel metodo costruttore della funzione degli oggetti *Queries*. L'esecuzione delle *SQ* viene eseguita da una funzione separata (*execute(ConnectionData ConnProcedure, ConnectionData ConnSensorDB)*).

Il metodo costruttore per eseguire le sue funzioni ha quindi bisogno dei *template* utilizzati e della stringa dell'insieme delle SQ dell'utente.

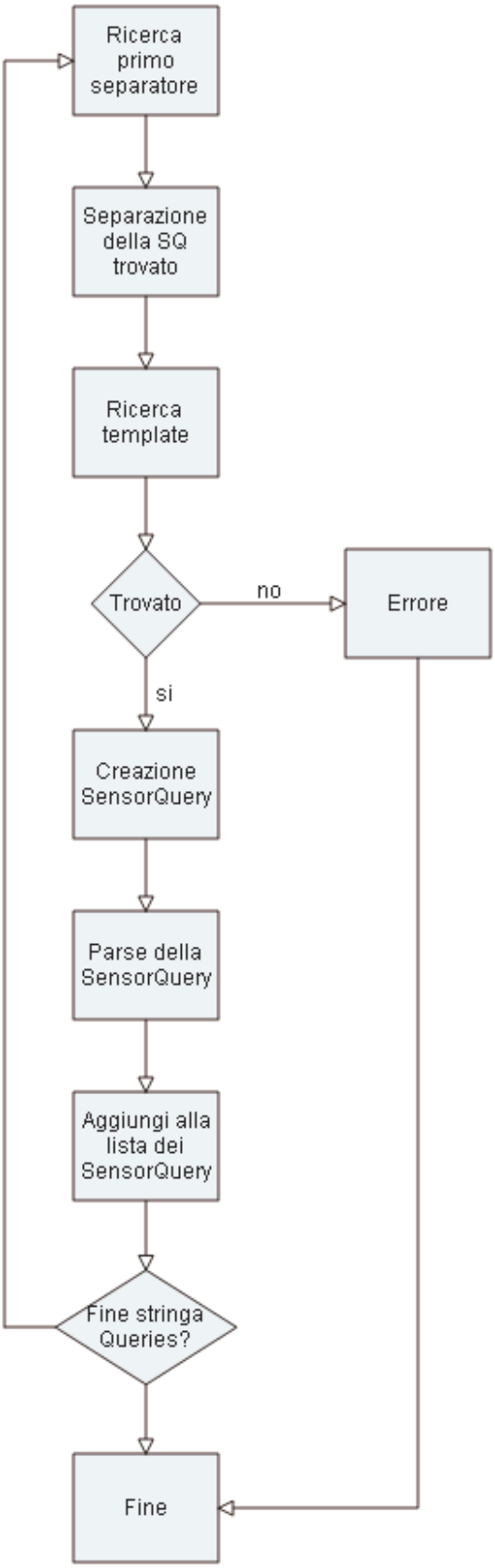


Figura 26 Diagramma di flusso della suddivisione delle SQ

Per eseguire la lista degli oggetti *SensorQuery* sono necessari 2 oggetti contenenti parametri di connessioni a due *database* che sono:

- Il *database* del server centrale , *ConnSensorDB*;
- Il *database* che contiene le procedure da eseguire, *ConnProcedure*, che verrà spiegato negli dettagli nel paragrafo dedicato al *database* del *Query Parser*.

Per facilitare l'accesso agli oggetti singoli nella lista degli oggetti *SensorQuery* sono state realizzate, come per i *template* delle funzioni di accesso sequenziale (*nextQuery()*) e di *reset* (*resetQueries()*).

Oltre alle funzioni principali durante la realizzazione della classe si è realizzata anche una funzione di *output* a video del parse delle *SQ* (*showQueries()*): tale funzione è stata ritenuta molto utile per mostrare all'utente il risultato del parse.

4.2.3 SensorQuery

La classe *SensorQuery* è la classe principale del modulo. Gli oggetti di questa classe si occupano della gestione delle singole *SQ* inserite dall'utente.

Le sue funzioni sono:

- *Parse* delle stringhe di *SQ*;
- Esecuzione delle *SQ*.

4.2.3.1 Parse delle SensorQuery

Per realizzare la funzione parse, si usano gli oggetti della classe template e si fa il confronto delle parole chiave.

Esistono 2 funzioni di parse:

- la prima, *parseSubQuery*, serve per suddividere la *SensorQuery* in un'insieme di *sub-query*. Questa funzione scorre sequenzialmente i *sub-template* dell'oggetto della classe *template* assegnato e cerca di suddividere la stringa della *query* in *sub-query* attraverso le parole chiave (*begin word*) delle *sub-query*. Appena identificata una nuova *sub-query*, si crea di nuovo un oggetto della classe *SensorQuery* usando la stringa della *sub-query* ricavata e associandogli il relativo *sub-template*; l'oggetto creato viene inserito nella lista di *sub-query*. Se la *sub-query* non è atomica, si richiama la funzione di parse delle *sub-query* in modo ricorsiva, altrimenti se la *sub-query* ha anche dei parametri, viene richiamata la seconda funzione di *parse*, la funzione *parseValue*. Una volta terminata la funzione di *parse* di una delle *sub-query* si aggiungono i parametri ricavati dalla funzione parse, con la propria lista dei valori dei parametri, in questo modo, una *SensorQuery* ha sempre tutti i valori dei parametri delle sue *sub-query*. Attraverso questa funzione di parse applicato in maniera ricorsiva si ottiene una struttura di oggetti ricorsivi simile alla struttura ricorsiva degli oggetti *template*.

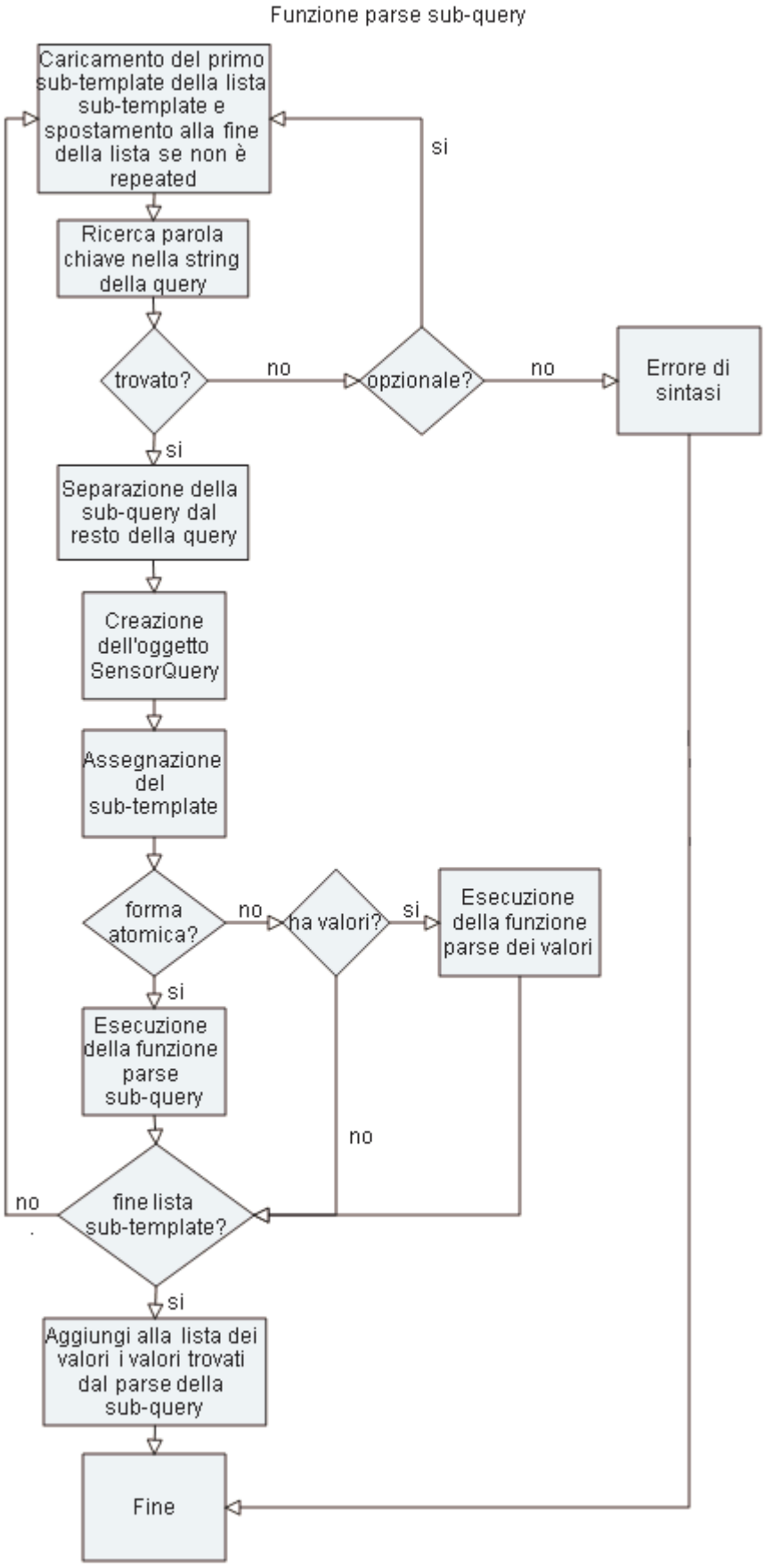


Figura 27 Diagramma di flusso del parseSubQuery

2. Questa seconda funzione di *parse*, *parseValue*, serve a ricavare i valore dei parametri delle *SQ* dalle *sub-query* in forma atomica. Il funzionamento è semplice, si toglie la parola chiave della *SQ* e si memorizzano le restanti parole in una lista di oggetti della classe *Value* ove ogni valore è considerato separato dal successivo dallo spazio. Per poter identificare meglio i valori dei parametri, gli oggetti della classe *Value* sono coppia di 2 stringhe, di cui la prima rappresenta la parola chiave del parametro e l'altra il valore vero e proprio.

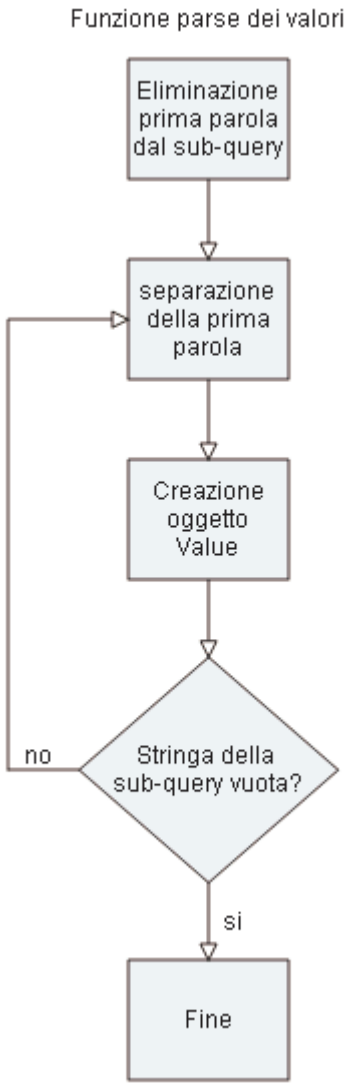


Figura 28 Diagramma di flusso del parseValue

Una volta terminata la funzione del *parse*, sia la prima sia la seconda funzione, ogni oggetto della classe *SensorQuery* ha sempre una lista di parametri ricavato dal parse. Per le *sub-query* in forma atomiche vengono ricavati direttamente dalla funzione *parse* dei valori, mentre per le *sub-query* non atomiche il valore viene ricavato unendo i valori dei

parametri delle sue *sub-query* dalla funzione di parse dei *sub-query*. Attenzione, le liste di valori degli parametri possono essere anche vuote, questo può capitare quando una *sub-query* atomica non ha valori, ma è composto solamente da una parola chiave.

4.2.3.2 Esecuzione delle SensorQuery

L'esecuzione delle SensorQuery sono delle interazioni con i database del sistema.

Come anticipato nel paragrafo relativo alla classe *Queries*, per eseguire le *SensorQuery* sono necessari 2 oggetti *ConnectionData*, contenenti le informazioni dei due database necessari:

- Il *database* del server centrale, *ConnSensorDB*;
- Il *database* che contiene le procedure da eseguire, *ConnProcedure*, che verrà spiegato negli dettagli nel paragrafo dedicato al *database* del *Query Parser*.

L'esecuzione viene suddivisa in 6 passi:

1. Ricavare la procedura da eseguire sul *database* centrale. Questo passo consiste nel connettersi al *database* delle procedure da eseguire, ricavare le informazioni della procedura da eseguire attraverso una *query* al database e creare un oggetto della classe *Procedure* con le informazioni ricavate. Le informazioni della procedura da eseguire sono la *stored procedure* da eseguire sul database centrale, non parametrizzato, e due valori booleani che rispettivamente indicano se la procedura produce dei risultati (*hasResult()*) e se è necessario eseguire delle query distribuite (*isDistributed()*);
2. Preparazione della *stored procedure* da eseguire, parametrizzando le variabili con i valori dei parametri ricavati dal parse del *SensorQuery*;
3. Connessione al database del sistema centrale ed esecuzione della *stored procedure* parametrizzato;
4. Rielaborazione dei risultati ricavati dalla *stored procedure* eseguita sul *database*

centrale, tali risultati possono essere o l'insieme di informazioni sulle *query* distribuite da eseguire nei vari *database*, se *isDistributed()* è vero, oppure se *hasResult()* è vero i risultati vengono memorizzati in una lista di risultati. Se è necessario eseguire un'insieme di *query* distribuite, *isDistributed()* è vero, con i risultati della procedura vengono creati una lista degli oggetti *DistributedQuery*, tali oggetti devono contenere le *stored procedure* da eseguire e le informazioni di connessione, gli oggetti *ConnectionData*;

5. Per ogni oggetto della classe *DistributedQuery* viene creata una connessione al database con le informazioni all'interno dell'oggetto *ConnectionData*, e viene eseguito la *stored procedure*;
6. Ogni singola stored procedure può ritornare un'insieme di risultati, che devono essere uniti per produrre le righe di risultati complessivi della *SQ*.

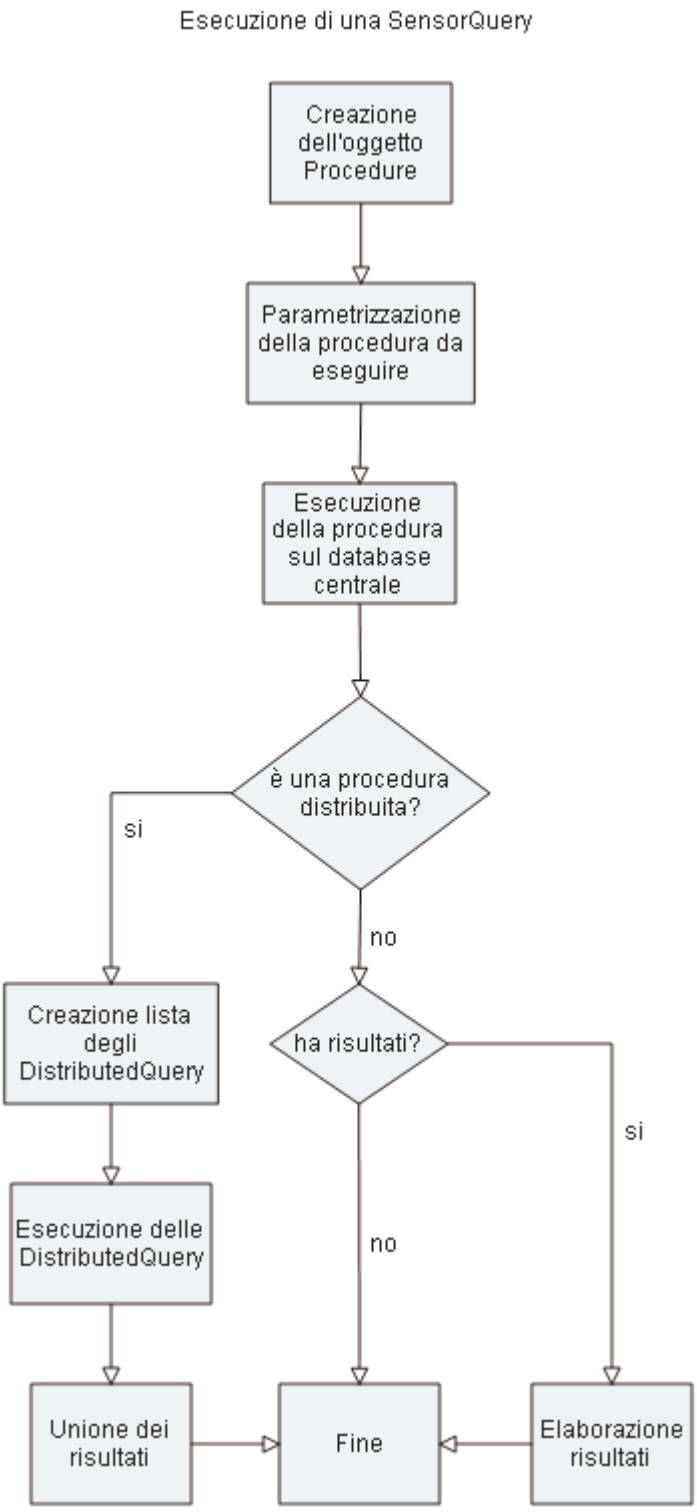


Figura 29 Diagramma di flusso dell'esecuzione di una SQ

4.2.4 Value

Gli oggetti della classe *Value* sono usati per memorizzare i valori dei parametri estratti dal *parse* delle *SQ*.

La creazione di questa classe è stata resa necessaria per facilitare l'uso dei parametri delle SQ. Le proprietà principali di questa classe sono ovviamente le due stringhe:

- La prima rappresenta la parola chiave del parametro, si è deciso di usare una concatenazione di tutte le parole chiave, non ripetuto, delle sub-query madri della sub-query atomica da cui si estrae il parametro;
- La seconda rappresenta il valore del parametro estratto.

Otre a queste due proprietà principali ci sono anche altre proprietà che possono facilitarel'uso dei parametri, e sono:

- *isRepeated()*, questa proprietà definisce se l'estratto può essere multiplo, questa proprietà è importante durante la parametrizzazione, la sostituzione dei parametri, nelle stored procedure, per concatenare i parametri che hanno valori multipli;
- *isUsed()*, questa proprietà invece è utile durante la paramtrizzazione per indicare che il parametro è già stato usato nella sostituzione.

4.2.5 ConnectionData

La classe ConnectionData è stata creata per poter passare agevolmente le informazioni di connessione tra le varie funzioni.

La struttura di questa classe descrive una serie di proprietà che rappresentano i parametri necessari alla connessione ad una *database* **MySQL**, attraverso la funzione *mysqlpp::connect(const char *db=0, const char *server=0, const char *user=0, const char *password=0, unsigned int port=0)*. Le proprietà della classe sono:

- *getSchema()*, che è il database a cui si vuole connettere;

- *getHost()*, rappresenta l'*host* su cui si trova il database;
- *getUser()*, è l'utente usato per la connessione;
- *getPassword()*, è la password usata per la connessione;
- *getPort()*, è la porta a cui connettersi per la connessione.

4.2.6 Procedure

La classe *Procedure* serve per creare gli oggetti che memorizzano i dati delle procedure da eseguire sul database del sistema centrale.

Le proprietà di questa classe sono:

- *getBegin()*, rappresenta la parola chiave (*begin word*) della *query* per cui l'oggetto *Procedure* viene creata;
- *getProcedureName()*, è la *stored procedure* non parametrizzata che verrà poi successivamente elaborata, parametrizzata, ed eseguita.
- *HasResult()*, è la proprietà che indica se la procedura eseguita ha o no un risultato da mostrare all'utente (escludendo le informazioni delle *query* distribuite);
- *isDistributed()*, se il valore è vero indica se è necessario eseguire delle *query* distribuite per la *SQ*, ed è necessario creare un lista di oggetti della classe *DistributedQuery* per eseguirle;

4.2.7 DistributedQuery

Le istanze della classe *DistribuitedQuery*, sono utilizzate per eseguire le procedure sui database distribuiti nel sistema. Quindi questa classe ha il compito di creare gli oggetti in grado di eseguire le *stored procedure*.

Per utilizzare questa classe è quindi necessario fornire alla classe costruttore una stringa che corrisponde all'invocazione della *stored procedure* da eseguire, inoltre è necessario un oggetto della classe *ConnectionData*, che fornisca le informazioni della connessione al database.

Infine per questa classe occorre anche una lista di stringa necessaria per immagazzinare i risultati ottenuti dall'esecuzione della *stored procedure*.

4.3 Database per il modulo Query Parser

Come anticipato nei paragrafi precedenti il modulo *Query Parser* ha bisogno di un *database*, che può essere separato dal resto dei *database* del sistema, per diversi scopi:

- Caricamento dei *template* delle *SQ*;
- Caricamento delle procedure da eseguire sul database del sistema centrale.

4.3.1 Database per il template

Il *database* per memorizzare i *template* delle *SQ* deve poter contenere una struttura ricorsiva.

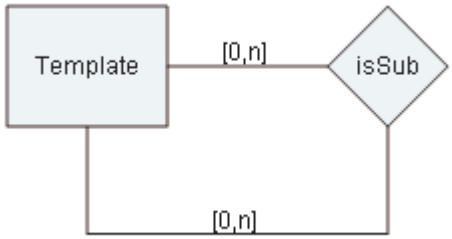


Figura 30 Schema E/R del database per memorizzare i template

Per implementare questa struttura sono stati create 2 tabelle:

- La tabella *template*, in cui vengono immagazzinate le informazioni dei *template* e dei *sub-template*, tra cui:
  - *BeginWord* che è la parola chiave del *template* o *sub-template*;
  - *subQueryLevel*, è il livello del *template* o *sub-template*, il livello del *template* è sempre 0. La coppia *BeginWord* e *subQueryLevel* identifica univocamente un record del *template* (chiave primaria);
  - *hasValue*, indica se il *sub-template* ha valore dei parametri;
  - *isMinimal*, indica se il *sub-template* è atomica.
- La tabella *subTemplates*, ha il compito di tenere traccia della relazione di *sub-template* tra i record all'interno del *template*. Perciò rappresenta la relazione *isSub*. Inoltre, in questa tabella contiene anche le informazioni relative alla posizione del *sub-template* figlio, e se il *sub-template* figlio è opzionale e/o ripetuto. I suoi campi sono:
  - *BeginWordTmpl*, la parola chiave del padre;
  - *QueryLevel*, il livello *template* del padre;
  - *BeginWordSubTmpl*, la parola chiave del figlio;

- *SubLevel*, il livello *template* del figlio;
- *Sequence*, la posizione del figlio all'interno dei *sub-template* del padre;
- *isOptional*, indica se il figlio è un *template* opzionale all'interno del padre;
- *isReapeated*, indica se il figlio è un *sub-template* ripetuto all'interno del padre;

4.3.2 Database per le procedure

La parte del *database* usata per memorizzare le procedure delle *SQ*, è in realtà una sola tabella (ed è stata chiamata *execprocedure*) con le informazioni necessarie per l'esecuzione delle procedure.

I campi della tabella sono:

- *BeginWord*, è la parola chiave della *SQ* a cui è associata la procedura;
- *ProcedureName*, è l'istruzione della *stored procedure* da eseguire, non parametrizzata;
- *HasResult*, indica se la *stored procedure* eseguita sul *database* del sistema centrale ha o meno un *set* di risultati, che non siano le informazioni per costruire gli oggetti *DistributedQuery*;
- *distributed*, indica se per completare l'esecuzione della *SQ* è necessario eseguire delle *query* distribuite.

4.4 Sintassi delle SQ

In questo paragrafo si vuole descrivere la struttura delle SQ già in parte anticipata nel capitolo precedente.

Il set di *SQ* fondamentali per il controllo del sistema attualmente sono:

- *Store*, che consente la creazione di una nuova query di rilevamento, specificando il nome della *query* di rilevamento, i dispositivi (sensori) che devono effettuare i rilevamenti, la frequenza dei rilevamenti e i eventi dei rilevamenti;
- *Alter*, col quale è possibile modificare una *query* di rilevamento già creata, specificando il nome della *query* di rilevamento e le modifiche da apportare;
- *Drop*, attraverso il quale è possibile eliminare una *query* di rilevamento;
- *Stop*, è l'istruzione usata per fermare l'esecuzione di una query di rilevamento;
- *Restart*, usato per far ripartire una query di rilevamento fermata dall'istruzione *stop*;
- *Select*, comando per selezionare i risultati dei rilevamenti.

Per scrivere la sintassi delle *SQ* si è scelto di usare una sintassi *pseudo-SQL*, in cui le *sub-query* indicate tra i parentesi quadre [] sono opzionali.

4.4.1 Store

La sintassi della *store* è la seguente:

STORE

QUERY query_name
DEVICES devices
[FREQUENCY frequency_value]

[EVENTS [MAX maxvalue] [MIN minvalue] [MAXDELTA maxdeltavalue] [MINDELTA mindeltavalue]]

La *SQ* è composta da 5 *sub-query* (*STORE*, *QUERY*, *FROM*, *FREQUENCY*, *EVENTS*) di cui le prime 3 sono obbligatorie, mentre quelle successive sono facoltativi.

La *sub-query STORE* è una *sub-query* senza parametri, e serve per identificare la *SQ*.

La *sub-query QUERY* è una *sub-query* usata per dare il nome alla *query* di rilevamento che si crea; il nome della *query* di rilevamento lo si mette al posto del *query_name*.

La *sub-query DEVICES* è una *sub-query* usata per indicare i sensori o i gruppi di sensori con cui si vogliono fare i rilevamenti.

La *sub-query FREQUENCY* è una *sub-query* usata per dare la frequenza di rilevamento, il valore può essere composto da un valore di tempo con un'unità di misura che può essere minuto (m), ora (h) o secondo (s), es. *FREQUENCY* 5 s, un rilevamento ogni 5 secondi. Se l'unità di misura non viene indicata si suppone che sia in secondi (default). Se la *SQ* non contiene la *sub-query FREQUENCY*, significa che la frequenza di rilevamento è **uguale a 0** (default), e di conseguenza gli *host wrapper* chiedono ai sensori di fare **un solo rilevamento**.

La sub-*query EVENTS* è una *sub-query* usata per segnalare gli eventi da rilevare dagli *host wrapper*. È composta a sua volta da 5 sub-query (*EVENTS*, *MAX*, *MIN*, *MAXDELTA*, *MINDELTA*), ognuna delle quali, tranne la prima, richiede un tipo di evento specifico e sono tutti opzionali.

4.4.2 Alter

La sintassi della *alter* è la seguente:

ALTER

QUERY query_name

[ADD [DEVICES devices_name] [EVENT [MAX maxvalue] [MIN minvalue] [MAXDELTA maxdeltavalue] [MINDELTA mindeltavalue]]

[REMOVE [DEVICES devices_name] [EVENT [MAX maxvalue] [MIN minvalue] [MAXDELTA maxdeltavalue] [MINDELTA mindeltavalue]]

[UPDATE [FREQUENCY frequency_value] [EVENT [MAX maxvalue] [MIN minvalue] [MAXDELTA maxdeltavalue] [MINDELTA mindeltavalue]]]

Questa *SQ* è leggermente più complessa della precedente, è composta da 5 *sub-query* (*ALTER*, *QUERY*, *ADD*, *REMOVE*, *UPDATE*). La parola chiave della *SQ* è *ALTER* e come la *sub-query STORE* non ha parametri.

La *sub-query QUERY* è la stessa della *SQ* precedente, ma in questo caso indica il nome della *query* di rilevamento da modificare.

La *sub-query ADD* serve a specificare le cose da aggiungere alla *query* di rilevamento, e ha al suo interno 3 sub-query (*ADD*, *DEVICES*, *EVENTS*). *ADD* è una *sub-query* senza parametri, serve a differenziare *DEVICES* da aggiungere rispetto a *REMOVE* che le rimuove. La *sub-query DEVICES* è una *sub-query* che serve a indicare i sensori, molto simile al *FROM* della *SQ store*, mentre *EVENTS* è la *sub-query* per indicare gli eventi, come quella della *SQ store*.

La *sub-query REMOVE* è simile alla *sub-query ADD*, con la differenza che specifica i sensori e gli eventi che non servono più.

La *sub-query UPDATE* serve ad aggiornare la frequenza di rilevamento e gli eventi da segnalare, usa le stesse *sub-query* già presenti nella *store*.

4.4.3 Drop, stop, restart e select

Le sintassi della *drop*, *stop*, *restart* e select sono molto simili e semplici sono le seguenti:

DROP

QUERY query_name

STOP

QUERY query_name

RESTART

QUERY query_name

SELECT

QUERY query_name

Per tutte 4 le *SQ* è sufficiente indicare il nome della *query* di rilevamento su cui agire, quindi le strutture delle *SQ* sono praticamente uguali tranne per la parola chiave della *SQ* che le identifica.

4.5 Esempio: Implementazione della SQ store

In questo paragrafo si vuole mostrare un esempio di come implementare una *SQ*. La *SQ* che verrà considerata è la *SQ store*, che è una delle *SQ* più complesse ed è la più importante.

Per implementare la *SQ* ci sono varie operazioni da fare:

1. Aggiungere la *SQ* al database dei *template*;
2. Definire la procedura da eseguire sul database del sistema centrale per la *SQ* sul database centrale;
3. Definire le procedure distribuite da fare sui database locali dei *host wrapper*;
4. Aggiungere la procedura da eseguire sul database del sistema centrale;
5. Testare la *SQ*.

4.5.1 Database dei template

Aggiornare il database dei *template* serve per poter fare correttamente il parse della *SQ*, è necessario inserire tutti i *sub-template* utilizzati, e le loro relazioni nel database. Per i *sub-template* della *SQ store* sono state inserite le seguenti valori:

```
-- Query
insert into template value( 'query', 1, true, true);
-- Devices
insert into template value( 'devices', 1, true, true);
-- Frequency
insert into template value( 'frequency', 1, true, true);
-- Event
insert into template values( 'events', 1, false, false);
insert into template values( 'events', 2, false, true);
insert into subtemplates values( 'events', 1, 'events', 2, 0, false, false);
insert into template values( 'min', 1, true, true);
insert into subtemplates values( 'events', 1, 'min', 1, 1, true, false);
insert into template values( 'max', 1, true, true);
insert into subtemplates values( 'events', 1, 'max', 1, 2, true, false);
insert into template values( 'deltamin', 1, true, true);
insert into subtemplates values( 'events', 1, 'deltamin', 1, 3, true, false);
insert into template values( 'deltamax', 1, true, true);
insert into subtemplates values( 'events', 1, 'deltamax', 1, 4, true, false);
```

Mentre per il *template store* sono stati inseriti:

```
-- Store
insert into template value( 'store', 0, false, false);
insert into template value( 'store', 1, false, true);
insert into subtemplates value( 'store', 0, 'store', 1, 0, false, false);
insert into subtemplates value( 'store', 0, 'query', 1, 1, false, false);
insert into subtemplates value( 'store', 0, 'devices', 1, 2, false, false);
insert into subtemplates value( 'store', 0, 'frequency', 1, 3, true, false);
insert into subtemplates values 'store', 0, 'events', 1, 4, true, false);
```

Dopo l'inserimento di questi valori nel database del template, il componente Query Parser è già in grado di fare di fare il parse di una SQ come la seguente:

"store query provafinale14 devices provaNet prova2 frequency 20 m events min 27.6 max 37.5 ;"

Il risultato del parse è:

Query 1: store query provafinale14 devices provaNet prova2 frequency 20 m events min 27.6 max 37.5
store.query: provafinale14
store.devices: provaNet
store.devices: prova2
store.frequency: 20
store.frequency: m
store.events.min: 27.6
store.events.max: 37.5

4.5.2 Definire la stored procedure da eseguire sul database del sistema centrale

Una volta fatto il parse della *query* è necessario definire la *stored procedure* da eseguire sul database del sistema centrale, tale procedura ha due scopi:

- Aggiornare i dati del *database* centrale;
- Acquisire informazioni delle procedure da eseguire per i *database* distribuiti sui *host wrapper*.

Il codice della stored procedure è in appendice A1.

4.5.3 Definire le stored procedure sui database distribuiti

Oltre alla *stored procedure* sul *database* del sistema centrale è necessario definire le *stored procedure* da eseguire sugli *database* distribuiti sugli *host wrapper*.

Per la *SQ store* sono state definite 2 *stored procedure* sugli *host wrapper*:

- La prima usata per inserire i dati sulla tabella delle configurazioni dei sensori;
- L'altra usata per inserire le condizioni da rilevare per le *query* di rilevamento.

Il codice delle stored procedure è in appendice A2.

4.5.4 Database delle procedure

Infine occorre inserire nella tabella delle procedure da seguire la relazione tra la *SQ* e la *stored procedure* da eseguire sul *database* del sistema centrale.

Per la *SQ store* sono stati inseriti i seguenti valori:

insert into execprocedure value('store', 'call store("store.query","store.devices","store.frequency","store.events.min","store.events.max","store.events.deltamin","store.events.deltamax");', false, true)

Una volta fatto quest'ultimo passo è possibile testare la *SQ*.

L'output del programma dell'esecuzione della SQ che è stato mostrato precedentemente ("store query provafinale14 devices provaNet prova2 frequency 20 m events min 27.6 max 37.5 ;") è:

select * from execProcedure where BeginWord = 'store';
exec Procedure: call store('provafinale14','provaNet prova2','20 m','27.6','37.5',",");

Execute procedure: call insertconfiguration('48','1','1200', true);

Execute procedure: call insertconfiguration('48','9','1200', true);

Execute procedure: call insertconfiguration('48','2','1200', true);

Execute procedure: call insertconditions('48','27.6','37.5','','');

Query Executed

Alla *query* di rilevamento creato è stato assegnato *IDQ* 48.

I sensori usati per fare i rilevamenti hanno *IDD* 1, 9, 2. I sensori 1 e 9 fanno parte del gruppo sensori *provaNet*, mentre il sensore 2 ha il nome di *prova2*.

La frequenza di rilevamento è di 1 rilevamento ogni 1200 secondi (20 minuti).

Infine sono necessari il rilevamento delle condizioni di temperatura minima 27,6 e temperatura massima 37,5.

Capitolo 5: Prospettive future

Il sistema finora descritto è ancora incompleto, a parte la mancanza dei moduli Event Manager, l'Interfaccia Utente e l'Error Manager, al sistema mancano ancora degli altri moduli di supporto per essere completo.

Inoltre come si è detto fin dall'inizio, l'Interfaccia Utente dovrà essere sostituita o affiancata da un componente di controllo automatico, che a seconda degli eventi o dei rilevamenti dovrà calcolare automaticamente le *SQ* da far eseguire al sistema.

Infine c'è da dire che l'architettura del sistema finora descritto è un'architettura *client-server* con un solo server centrale. Questo tipo di architettura è inefficiente e poco sicura.

In questo capitolo si cercherà di mostrare quali potrebbe essere lo sviluppo futuro del sistema finora mostrato, affrontando le problematiche che sono state appena descritte.

5.1 Moduli di supporto

Il sistema di controllo dei sensori ha bisogno di altri moduli di supporto per poter funzionare correttamente.

I moduli aggiuntivi devono svolgere i seguenti compiti:

Aggiornare lo stato dei sensori del sistema, lo spostamento dei sensori da un host wrapper un'altro, le condizioni dei sensori (attivo, spento), ecc..;

Aggiornare lo stato degli host wrapper;

Aggiornamento delle informazioni di connessione agli host wrapper ed al database centrale.

Questi moduli per quanto siano opzionali in una fase di sviluppo del sistema, in cui tutti i parametri possono essere immessi manualmente, sono molto importanti per un uso reale del sistema, in quanto è impensabile che ci sia un operatore umano che modifichi manualmente tutti gli aggiornamenti dei cambiamenti del sistema, per ovvie ragioni di praticità ed errore umano nell'inserimento di dati. Quindi in un'applicazione reale del sistema di controllo dei sensori, è necessario che ci siano dei moduli che preservino continuamente l'aggiornamento delle condizioni del sistema. L'operatore umano deve intervenire solo in casi di malfunzionamenti del sistema o in casi particolari che non si riesce ad automatizzare.

5.2 Modulo di controllo automatico

Il modulo di controllo automatico ha la funzione di calcolare in maniera automatica la frequenza di rilevamento da effettuare per osservare il fenomeno a cui si è interessati. Il modulo dovrebbe cercare di effettuare il minimo numero di rilevamenti necessari senza però perdere i fenomeni interessanti o critici.

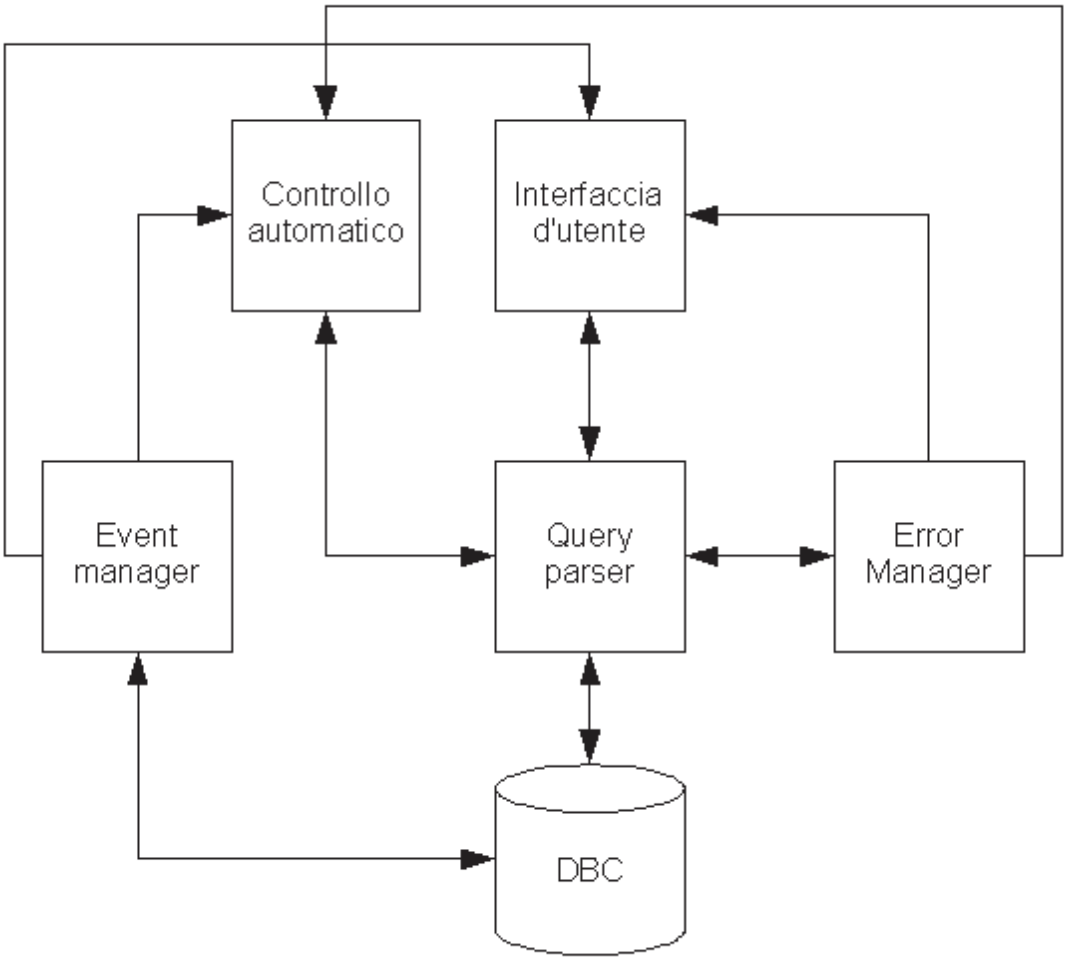


Figura 31 Schema a blocchi del sistema centrale con il controllo automatico

Durante il funzionamento normale del sistema deve essere sempre il controllo automatico a decidere la frequenza di rilevamento. Anche se questo non significa escludere completamente l'operatore umano, anzi il controllo umano deve essere prioritario rispetto al controllo automatico, ma durante il funzionamento normale, il sistema deve essere in grado di gestirsi in maniera automatica.

Il più semplice dei moduli di controllo automatico potrebbe aumentare la frequenza di rilevamento quando si hanno rapide variazioni tra un rilevamento e quello successivo, utilizzando l'evento delta minimo o delta massimo, modificando la frequenza di

rilevamento quando si supera un certo valore di soglia di delta massima o quando è inferiore a una delta minima.

Un modulo con un comportamento più complesso può implementare il modello di predizione dei rilevamenti descritto nella tesi del dott. Alessio Cavallini, *OPTIMIZATION OF CONTINUOUS-MONITORING QUERIES IN SENSOR NETWORKS* [2] (http://www.dbgroup.unimo.it/tesi/indexVod.html), in cui si cerca di predire i rilevamenti futuri attraverso il modello di Holt-Winter, che ha la caratteristica di considerare oltre al valore precedente, anche il trend e la stagionalità . Con questo modello di previsione, è possibile chiedere al sistema di aumentare la frequenza di rilevamento a seconda della differenza tra il dato rilevato e il dato predetto.

5.3 Architettura multi-controller

L'architettura del modello descritto finora ha solo un sistema centrale che coordina tutti gli *host wrapper*, i quali a loro volta segnalano eventi al sistema centrale scrivendo sul *database* centrale.

Un'architettura del genere ha l'unico vantaggio di essere semplice, ma presenta numerosi svantaggi tra cui:

- Il sistema centrale può diventare il collo di bottiglia del sistema;
- Un unico punto di *failure*, rappresentato dal sistema centrale, se il sistema centrale smette di funzionare l'intero sistema smette di funzionare;
- Tutte le applicazioni dovranno per forza rivolgersi ad un unico server per controllare i sensori ed ottenere i risultati dei rilevamenti.

Quindi per avere un sistema di controllo più robusto ed efficiente è necessario avere più *server* di controllo che possono agire sui sensori.

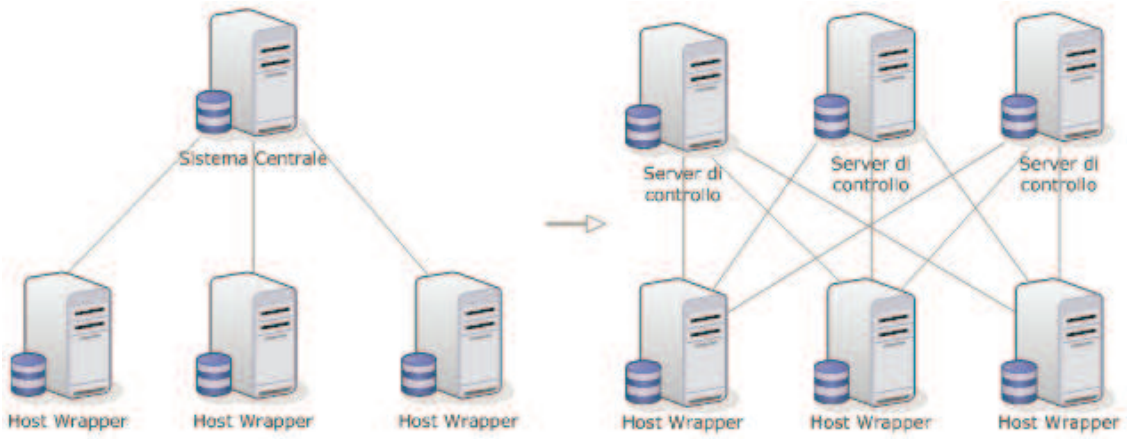


Figura 32 Evoluzione del sistema da un solo server di controllo centrale a più server di controllo

Ciò significa che ogni *host wrapper* non fa più riferimento ad un unico sistema di controllo centrale, ma deve poter interagire con più server di controllo, quindi anche ogni *host wrapper* dovrà avere una tabella con le informazioni di connessione di diversi server di controllo, che dovrà esser aggiornato a seconda dello stato dei vari server di controllo. Questo cambiamento nell'architettura comporterà che la numerazione delle *query* di rilevamento non sarà più univoca in senso assoluto, ma sono solo univoca all'interno di un determinato *server* di controllo. Ciò comporta una modifica negli host wrapper che dovranno considerare come chiave primaria delle *query* di rilevamento la coppia (IDQ, nome del *server* di controllo). Inoltre per segnalare un evento, l'*host wrapper* dovrà prima trovare le informazioni di connessione relative al server di controllo della *query* di rilevamento, e una volta trovatolo inserire le informazioni all'interno del *database* del *server* di controllo.

Conclusioni

Le WSN sono una tecnologia innovativa che ha permesso la realizzazione di reti di sensori che prima erano irrealizzabili con i sensori cablati, apprendo nuove nuovi prospettive e campi applicativi. Ma questa tecnologia è ancora in fase di sviluppo, ci sono ancora delle problematiche che ne limitano l'uso.

Una di queste è il consumo energetico, con questa tesi è stato realizzato un componete di un sistema in grado limitare il consumo, attraverso la regolazione della frequenza di rilevamenti dei dispositivi della rete e quindi anche la frequenza di trasmissione dei dati. Attraverso il sistema realizzato è possibile rendere più efficiente il rilevamento e la raccolta dei dati senza peggiorare la qualità dei dati raccolti.

Il componente software realizzato ha l'obbiettivo di rendere possibile il controllo delle reti di sensore attraverso semplici interrogazioni sviluppate in un'estensione del linguaggio SQL interagwndo con *host wrapper* (realizzato nella tesi dell'ing.. Daniele Caiti [3]), che ha il compito di interfacciarsi con la rete di sensori attraverso il nodo destinazione, AP a cui è collegato.

Le linee guida seguite per la progettazione e realizzazione del componente sono state quelle di sviluppare un componente estremamente flessibile ed estensibile, a cui è possibile aggiungere nuove istruzioni di controllo semplicemente aggiornando i database del sistema. Questo può risultare comodo in quanto essendo il progetto dell'intero sistema ancora in una fase iniziale, non è possibile conoscere tutte le istruzioni di controllo di cui si avrà bisogno mano a mano si sperimenterà il sistema su domini applicativi diversi. In questa tesi sono state descritte sei istruzioni fondamentali ma in futuro si prevede che molte altre potranno essere aggiunte. Il dotarsi di un analizzatore di rete realizzato in modo statico avrebbe comportato un vantaggio immediato in termini di tempi di prototipazioni e complessità ma uno svantaggio enorme in termini di estendibilità e sviluppo di un sistema generale di controllo. Lo stesso discorso vale per le funzioni d'esecuzione delle istruzioni, che a loro volta non possono essere un'insieme statico (definito a priori), ma un sistema dinamico che deve consentire il facile inserimento delle procedure per eseguire nuove istruzioni. Per questo si è pensato di inserire le nuove procedure di esecuzione sotto forma di *stored procedure* all'interno dei database. Infine le funzioni d'esecuzione devono consentire di eseguire le istruzioni che agiscono sui diversi database del sistema, distribuito tra quello centrale di controllo quelli locali sui nodi della rete.

Il risultato finale è di questa tesi è quindi il progetto e la realizzazione di un componente estremamente flessibile e versatile capace di analizzare qualsiasi istruzione con una struttura di *sub-query* ricorsive di cui è stato creato il *template*. La soluzione tecnologica di analisi di query ricorsive può trovare applicazione non solo nell'ambito delle reti di sensori, ma essere usato in un qualsiasi sistema che necessita di un analizzatore capace di estrarre i parametri da un'istruzione con un *template* definito. Inoltre lo sviluppo di questo componente fornisce un buon esempio di come gestire un *database* distribuito.

Infine c'è da dire che tutto il software sviluppato per questa tesi è stato realizzato usando degli strumenti *Open Source*. Tutte le librerie usatr per sviluppare il progetto, il database usato, **MySQL**, perfino l'*editor* di C++, **Eclipse**, il compilatore e il *debugger* sono tutti *OpenSource*. Questo aspetto ha reso un po' più complesso il processo di sviluppo, dando luogo ad un progetto stimolante e il cui risultato è pienamente soddisfacente. L'attivitò è stata svolta presso il laboratorio di reti di sensori istituito presso il dipartimento di Ingegneria dell’Informazione

Appendice A: Codice Stored Procedure

In questa appendice si mostra esempi di stored procedure per implementare le *Sensor Query*.

A1 Stored Procedure sul database centrale per SQ STORE

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `store` $$
CREATE PROCEDURE `store`(queryname varchar(50),
                devices varchar(1000),
                frequency varchar(50),
                minT varchar(10),
                maxT varchar(10),
                dmin varchar(10),
                dmax varchar(10))
BEGIN
proc: begin
  declare newIDQ int;
  declare dev varchar(50);
  declare devs varchar(1000);
  declare vcfreq varchar(50);
  declare unit varchar(3);
```

```
declare freq int;
declare dID int;
declare nID int;
declare dHost varchar(50);
declare done int default 0;

set devs = devices;

if frequency <> '' then

  set vcfreq = substring_index(frequency, ' ', 1);
  set unit = substring_index(frequency, ' ', -1);

else

  set vcfreq = '0';

end if;

-- Controll if query name is exists
if exists (select * from storedquery where name = queryname) then
  leave proc;
end if;

-- Insert new Query
insert into storedquery value(null,queryname);
set newIDQ = @@identity;

-- Calculate frequency
case unit
  when 'm' then set freq = convert(vcfreq, signed) * 60;
  when 'h' then set freq = convert(vcfreq, signed) * 3600;
  else set freq = convert(vcfreq, signed);
end case;

create temporary table if not exists tmp_procedures(
`name` varchar(50) DEFAULT NULL,
`username` varchar(50) DEFAULT NULL,
`psw` varchar(50) DEFAULT NULL,
`db` varchar(50) DEFAULT NULL,
`port` int(10) unsigned DEFAULT NULL,
`proc` varchar(1000) DEFAULT NULL);
```

```
create temporary table if not exists tmp_dev(
`IDD` varchar(50) DEFAULT NULL,
`host` varchar(50) DEFAULT NULL);


-- Search devices and insert in configuration(remote)
sdev: repeat
  set dev = substring_index(devs, ' ', 1);

  if exists (select netID from net where name = dev) then
    -- insert into temporaty table
    insert into tmp_dev
    select nd.IDD, nd.Host
    from netdev as nd
    where netID in (select netID
                    from net
                    where name = dev)
    /*and not exists (select *
                    from tmp_dev as td
                    where td.IDD = nd.IDD
                    and td.Host = nd.host)*/;

  else
    -- insert into temporaty table
    insert into tmp_dev
    select d.IDD as IDD,
           d.Host as host
    from   device as d
    where  d.name = dev
    /*and not exists (select *
                    from tmp_dev as td
                    where td.IDD = d.IDD
                    and td.Host = d.host)*/;
  end if;

  if (instr(devs,' ')) = 0 then
    set devs = '';
  else
    set devs = substring(devs from (instr(devs,' ') + 1));
  end if;

until (devs = '') end repeat;
```

```
 -- insert configuration procedure
 insert into tmp_procedures
 select distinct c.name,
                 c.username,
                 c.psw,
                 c.db,
                 c.port,
                 concat('call insertconfiguration(' '', newIDQ, '','', d.IDD, '','', freq, '', true);') as proc
 from connectionInfo as c
 join tmp_dev as d on (c.name = d.host);
 -- insert conditions procedure
 insert into tmp_procedures
 select distinct c.name,
                 c.username,
                 c.psw,
                 c.db,
                 c.port,
                 concat('call insertconditions(' '', newIDQ, '','', minT, '','', maxT, '','', dmin, '','', dmax, '');') as proc
 from   connectionInfo as c
 join   tmp_dev as d on (c.name = d.host);
 -- insert device-query in QueryDev
 insert into querydev
 select distinct d.IDD as IDD,
                 d.host as Host,
                 newIDQ as IDQ
 from tmp_dev as d;

 -- Return distibuted queries
 select * from tmp_procedures;

end proc;
END $$

DELIMITER ;
```

Questa *stored procedure* ritorna una tabella temporanea (*tmp_procedures*) creata appositamente per contenere le informazioni che verrà usato per creare le *DistributedQuery*.

A2 Stored Procedure sui database Host Wrapper per SQ STORE

A2.1 Inserimento configurazione sensori

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `insertconfiguration` $$
CREATE DEFINER=`root`@`%` PROCEDURE `insertconfiguration`(in IDQ int,
                                    in IDDEV int,
                                    in freq int,
                                    in active bool)
BEGIN
  insert into configuration value(IDQ, IDDEV, freq, active);
END $$

DELIMITER ;
```

A2.2 Inserimento condizioni eventi

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `insertconditions` $$
CREATE PROCEDURE `insertconditions`(in IDQ int,
                                in minT varchar(10),
                                in maxT varchar(10),
                                in minD varchar(10),
                                in maxD varchar(10))
BEGIN
```

```
 declare maxTemp float;
 declare minTemp float;

 declare pMaxD float;
 declare pMinD float;
 declare nMaxD float;
 declare nMinD float;

 if maxT <> '' then
   set maxTemp = cast(maxT as decimal(4,1));
 end if;

 if minT <> '' then
   set minTemp = cast(minT as decimal(4,1));
 end if;

 if maxD <> '' then
   if maxD >= 0 then
     set pMaxD = cast(maxD as decimal(4,1));
   else
     set nMaxD = cast(maxD as decimal(4,1));
   end if;
 end if;

 if minD <> '' then
   if minD >= 0 then
     set pMinD = cast(minD as decimal(4,1));
   else
     set nMinD = cast(minD as decimal(4,1));
   end if;
 end if;

 insert into conditions value(IDQ, maxTemp, minTemp, pMaxD, pMinD, nMaxD, nMinD);

END $$

DELIMITER ;
```

Bibliografia

[1] Domenico Beneventano, Sonia Bergamaschi, Francesco Guerra e Maurizio Vincini, “Progetto di Basi di Dati Relazionali”, 2007

[2] http://sparc20.ing.unimo.it/tesi/cavallini.pdf

[3] http://sparc20.ing.unimo.it/tesi/CAITI_DANIELE_TESI.pdf

[4] Milo Barozzi, “Progetto e sviluppo di una rete di sensori wireless”, 2008

[5] http://www.ing.unibs.it/~wsnlab/download/WSNlab_tutorial04_gen2.pdf

[6] http://it.wikipedia.org/wiki/Wireless_sensor_network

[7] http://it.wikipedia.org/wiki/Topologia_di_rete

[8] http://www.csupomona.edu/~rfsmith/ECE499%20MSP430/slaa378a.pdf

[9] http://focus.ti.com/lit/ds/symlink/msp430f2274.pdf

[10] http://www.silica.com/fileadmin/02_Products/07_Designers-Choice-Online/DC1_07/DesignersChoice0107_31.pdf

[11] http://focus.ti.com/lit/ds/symlink/cc2500.pdf

[12] http://netlab-mn.unipv.it/thesis/MSc/Beltrame_Thesis.pdf

[13] http://www.erweb.it/pub/tecnologie_mysql.php

[14] http://www.bo.infn.it/calcolo/manuali/mysql.pdf

[15] http://openskill.info/infobox.php?IDbox=1115&boxtype=description

[16] http://www.mysqlitalia.it/wiki/Cos%27%C3%A8_MySQL

[17] http://www.sun.com/aboutsun/pr/2009-04/sunflash.20090420.1.xml

[18] Kevin Atkinson, Sinisa Milivojevic, Monty Widenius e Warren Young, “MySQL++ v3.0.9 User Manual”, 2009

[19] http://tangentsoft.net/mysql++/doc/html/refman/

[20] http://it.wikipedia.org/wiki/MySQL