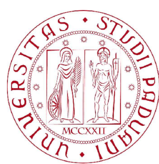


UNIVERSITA' DEGLI STUDI DI PADOVA

Laurea specialistica in
INGEGNERIA DELLE TELECOMUNICAZIONI

SVILUPPO DI UNO STRUMENTO PER L'ANALISI DI PROTOCOLLI IN RETI DI SENSORI RADIO

Laureando: **Bazzaco Leonardo**
Correlatore: **Castellani Angelo Paolo**
Relatore: **Rossi Michele**

ANNO ACCADEMICO 2009-2010

Alla mia famiglia
che ringrazio per
il costante sostegno

Sommario

In questo documento è stato sviluppato uno strumento in grado di analizzare il traffico in una rete di sensori wireless.
É stata sviluppata un'interfaccia grafica per permettere all'utente di monitorare in tempo reale l'andamento del traffico nella rete, permettendo di evidenziare eventuali problemi o malfunzionamenti.
Attraverso questo strumento è stata svolta un'analisi su un'algoritmo di routing svilppato all'interno del gruppo di ricerca SIGNET.

Indice

Introduzione **1**

1 Wireless Sensors Network **3**
1.1 Applicazioni delle WSN 4
1.2 Descrizione del testbed montato nel dipartimento 5
1.3 Obiettivo della tesi 8

2 Strumenti utilizzati **11**
2.1 Fandango 11
2.2 Google Web Toolkit 13
2.3 MySQL 14
2.3.1 Tipi di tabelle (*storage engine*) 14
2.3.2 Transazioni 15
2.4 Java DataBase Connectivity 19

3 Protocolli di routing per Wireless Sensors Networks **23**
3.1 Mobile Ad Hoc Networks 23
3.2 Wireless Sensors Networks 25
3.3 Valutazione delle prestazioni di un algoritmo di routing 27
3.4 Descrizione dell'algoritmo di Routing testato e dell'applicazione usata negli esperimenti 28
3.4.1 Formato dei pacchetti 28
3.4.2 Algoritmo di routing testato 30
3.4.3 Applicazione usata negli esperimenti 35

4 Routing Layer **37**
4.1 Struttura della Base di Dati 37
4.1.1 Tabelle 37
4.1.2 Modello Relazionale o *Entity-Relationhship* 40
4.2 Popolazione della Base di Dati 42
4.2.1 Raccolta del traffico 42

4.2.2 Processing 46
4.2.3 Populate 50
4.2.4 Copertura 57
4.3 Guida all'uso del Routing Layer 58
4.3.1 Presentazione grafica 59
4.3.2 Visualizzazione del traffico in tempo reale 61
4.3.3 Visualizzazione del traffico su un singolo link 62
4.3.4 Visualizzazione del traffico su una singola rotta 63
4.3.5 Costruzione del percorso hop by hop 64
4.3.6 Visualizzazione dei ritardi sui link di una singola rotta 64
4.3.7 Copertura 66
4.3.8 Quantità di traffico che transita per un nodo 66
4.3.9 Esportare i dati 67

5 Problemi riscontrati e risultati 69
5.1 Testbed utilizzato negli esperimenti 69
5.2 Traffico sopportato dalla rete 70
5.3 Presenza di nodi di relay 71
5.4 Ritardo End-To-End 72
5.5 Accodamento nei nodi 79
5.6 Link poco affidabili 82
5.7 Normalized Routing Load 88

Conclusioni 91

A Appendice 93

B Appendice 101

Introduzione

Il lavoro svolto in questo elaborato è parte integrante di un progetto più ampio chiamato WISE-WAI (Wireless Sensor networks for city-Wide Ambient Intelligence) finanziato dalla Cassa di Risparmio di Padova e Rovigo. Lo scopo principale del progetto è dimostrale la fattibilità dello sviluppo di Wireless Sensors Networks (WSN) su larga scala. Il Dipartimento di Ingegneria dell'Informazione dell'Università degli Studi di Padova, ha progettato e realizzato uno dei maggiori, per estensione e numero di nodi, testbed della comunità europea. Tale testbed è costituito da una rete di sensori composta da centinaia di TMoteSky, ovvero dei piccoli oggetti che integrano vari sensori ambientali, (umidità, temperatura, luminosità) un microcontrollore e un ricetrasmettitore radio. La dimostrazione dei concetti studiati per il progetto viene realizzata tramite il controllo remoto dei nodi sensore. Per visualizzare i risultati ottenuti e ottenibili sul testbed è stata realizzata, tramite l'utilizzo di Google Web Toolkit (GWT), un'applicazione di interfacciamento con gli strumenti forniti da Google Map. In particolare il lavoro svolto è stato quello di integrare nell'applicazione un livello per il controllo del traffico nella rete e in particolare per determinare le prestazioni dell'algoritmo di routing utilizzato nelle applicazioni caricate sui nodi sensori.
Lo strumento sviluppato si inserisce in modo intrusivo in una rete di sensori e non comporta perdite di prestazioni nella rete. Lo strumento ha lo scopo di evidenziare eventuali problemi o malfunzionamenti che si possono creare in una rete di sensori. Nel seguito di questo documento verrà descritto in dettaglio il testbed del dipartimento e gli strumenti utilizzati: MySQL, Java DataBase Connectivity (JDBC). Infine verranno presentati i risultati dei test effettuati.

Capitolo 1

Wireless Sensors Network

Una Wireless Sensors Network o rete di sensori senza fili è una sottoclasse delle reti ad hoc, composta da un insieme di nodi distribuiti in una certa area, capace di raccogliere informazioni dall'ambiente circostante, processare queste informazioni e trasmetterle via radio nella rete. I nodi sono dotati di un insieme di sensori ambientali quali luminosità, temperatura, umidità, una memoria RAM, una memoria ROM, un chip radio e una connessione seriale USB. Questa viene utilizzata come fonte di alimentazione nelle fasi di test. Il consumo energetico in una rete di sensori è di fondamentale importanza infatti un sensore è dotato di una limitata sorgente di energia e il tempo di vita di un nodo dipende molto dal tempo di vita della batteria. Essendo questa una fonte limitata di energia è necessario che i protocolli di comunicazione e le applicazioni tengano conto delle capacità limitate di processamento e di memorizzazione dei sensori. È per queste ragioni che molte ricerche si stanno concentrando sulla creazione di protocolli e algoritmi *power-aware*, ovvero protocolli che ottimizzano il consumo energetico.

La capacità di elaborazione interna del singolo sensore può aiutare i programmatori a diminuire le trasmissioni, aumentando quindi le prestazioni della rete, con un conseguente risparmio di energia. Questo può avvenire ad esempio aggregando le informazioni ricevute in più messaggi, e ritrasmettendole con in un unico messaggio.

In genere, in una WSN, esistono dei nodi speciali che si incaricano di raccogliere le informazioni di tutta la rete. Questi nodi vengono chiamati *sink* e rappresentano il ponte tra la rete di sensori e la rete cablata che accede a ad essi.

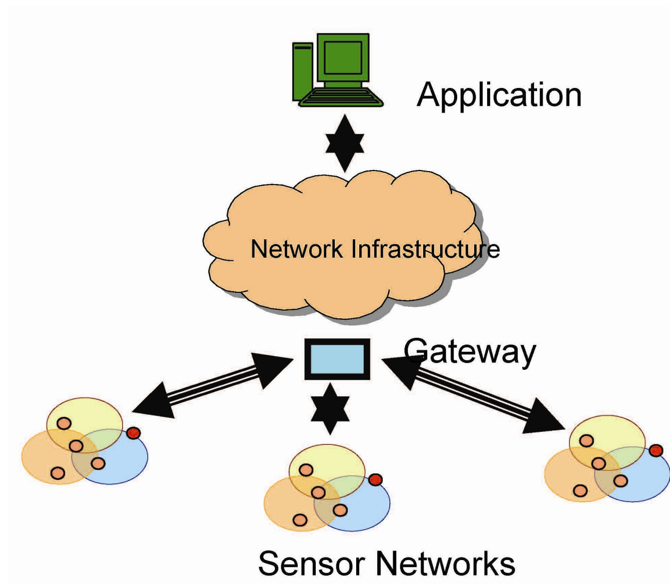


Figura 1.1: Struttura generale di connettività delle WSN

1.1 Applicazioni delle WSN

Le WSN ritrovano applicazioni in una grande varietà di ambienti, quali domestico, industriale, civile, etc. In ambiente domestico possono venire utilizzate ad esempio per il monitoraggio delle temperature in varie zone dell'edificio e quindi per il controllo dei consumi. In ambiente industriale, ad esempio per il controllo ambientale (temperatura, umidità, luminosità, vapori chimici, radiazioni) nei posti di lavoro potenzialmente pericolosi per i lavoratori (es. laboratori chimici e meccanici, fonderie, raffinerie), oppure in ambito di videosorveglianza e controllo delle intrusioni. In un ambiente cittadino intelligente, possono essere utilizzate nel controllo del traffico urbano. Si pensi ad esempio di montare un sensore in ogni automobile e grazie anche alle informazioni ottenute dal GPS riuscire a tracciare gli spostamenti dei veicoli. Uno strumento di questo tipo potrà permettere ad esempio di evitare accodamenti dando informazioni al conducente sulla rotta da seguire in tempo reale. Ancora, possono essere utilizzate nelle situazioni di emergenza per dare un supporto istantaneo alle squadre di salvataggio. Possono essere utili per gestire la sorveglianza dei luoghi pubblici, per fornire aiuto nel settore dell'agricoltura ad esempio per determinare i trattamenti da fare in funzione delle condizioni meteorologiche.

1.2 Descrizione del testbed montato nel dipartimento

Il testbed montato nel Dipartimento di Ingegneria dell'Informazione (DEI) è costituito da circa 300 sensori prodotti dalla CrossBow e appartenenti alla famiglia *telos* [1]. L'obbiettivo è quello di creare una rete sufficientemente grande da riprodurre in scala un ambiente cittadino, nella quale si possano effettuare test e simulazioni. I sensori sono distribuiti nel dipartimento in modo che ognuno di essi abbia nel proprio raggio di copertura un numero di nodi che va dai 10 ai 20. Il raggio di copertura è un parametro che è possibile variare agendo sulla potenza trasmissiva dei nodi. Per velocizzare le operazioni di riprogrammazione dei nodi è stata creata un'infrastruttura cablata che collega tutti i sensori, ad un server centrale come si vede in Figura 1.2.

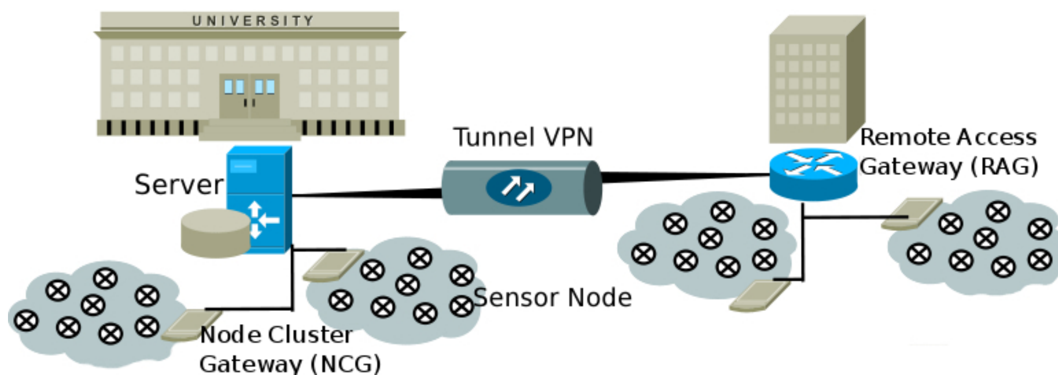


Figura 1.2: Architettura del dipartimento

Ogni nodo è connesso alla rete mediante una connessione Universal Serial Bus (USB) che fornisce l'alimentazione. In questo modo si è evita l'uso di batterie e si riducono i tempi di manutenzione della rete. Le comunicazioni tra i nodi, avvengono esclusivamente via radio. La rete cablata USB inoltre fornisce un'utile strumento per il salvataggio di dati nella fase di debug, in quanto il codice delle applicazioni che vengono caricate nei sensori non è eseguibile passo passo come nei classici linguaggi di programmazione ad alto livello. Ogni sensore infatti viene riprogrammato utilizzando un file eseguibile creato dalla compilazione del codice sorgente dell' applicazione che si vuole installare. Dopo l'avvio dell'applicazione, il nodo diventa indipendente e non è possibile ad esempio esplorare le variabili di ambiente del codice sospendendo l'esecuzione del programma. L'unico strumento che fornisce aiuto in questa fase è una libreria, chiamata *printf*, che invia alla porta seriale il valore delle variabili interessate ed è così possibile avere un minimo

controllo durante l'esecuzione di un applicazione.

Come si vede in Figura 1.2, la struttura della rete è gerarchica. Ogni nodo è connesso ad un hub USB che a sua volta è connesso ad un piccolo computer integrato Alix [2], che nella figura è chiamato *Node Cluster Gateway* (NCG). I NCG interagiscono con i nodi sia in up-stream (dati provenienti dal nodo verso il NCG), per esempio per l'invio di messaggi di debug o per la segnalazione dei dati raccolti, che in down-stream (dati provenienti dal NCG verso il nodo), per esempio per l'invio di comandi ai sensori (riprogrammazione, accensione, spegnimento). I NCG sono connessi via Ethernet al server centrale (WISE-WAI Server); a seconda della loro posizione geografica rispetto al server, possono essere connessi direttamente oppure tramite un *Remote Access Gateway* locale (RAG) che è connesso al server centrale tramite tunnel VPN.

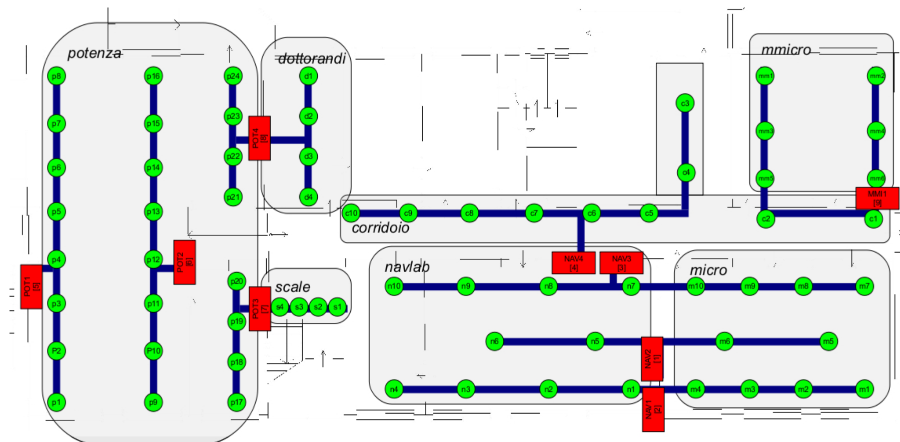


Figura 1.3: Mappa di un'area del dipartimento

I nodi sono raggruppati in insiemi a seconda della zona (laboratori) in cui sono posizionati. In AppendiceA sono mostrate in dettaglio tutte le mappe relative ai sensori montati nel dipartimento. Si prenda ad esempio la Figura 1.3 che mostra i gruppi: *navlab, mmicro, micro, corridoio, dottorandi, scale, potenza*. I nodi sono rappresentati dai cerchi di colore verde, in colore blu sono rappresentati i collegamenti USB, infine le scatole rosse rappresentano i NCGs. I nodi che appartengono ad un NCG sono connessi a gruppi di 4 ad un hub USB a 4 ingressi (es. Se un NCG connette 12 nodi, verranno utilizzati 3 hub USB). Ogni gruppo, a seconda della dimensione può contenere più NCG e si può notare che un NCG può connettere nodi di diversi gruppi. In particolare, i NCG sono una componente essenziale della gerarchia di rete.

Si tratta di piccoli computer di dimensione 15 cm x 15 cm, alimentati con *Power-over-Ethernet* (POE) dotati di interfaccia USB, Ethernet, processore, memoria RAM, memoria flash, PCI Slot. Si può vedere un esempio in Figura 1.4. Come descritto in precedenza alcuni gruppi sono connessi al server centrale mediante tunnel VPN; si può vedere dalle mappe riportate in Appendice A che questi sono i gruppi: *signet, die, torretta.*
L'architettura appena descritta è scalabile, facile da replicare nel caso debba essere estesa, e inoltre i componenti sono facilmente raggiungibili nel caso fosse necessaria una manutenzione. Va notato inoltre che gli hub USB permettono un hard-reset dei sensori, evitando che questi debbano essere scollegati e ricollegati manualmente nel caso di blocco del software. Per tutte queste ragioni, dividendo la rete in piccoli sottoinsiemi gestiti dai NCGs che permettono il controllo da remoto, una struttura di questo tipo, è un ottima soluzione.

Figura 1.4: Un esempio di collegamento tra un Alix (NCG) e gli hub USB ai quali sono connessi i sensori

Connettività IP Attualmente la rete di controllo del testbed identifica i nodi con indirizzi IPv4 [7] ed è direttamente connessa alla rete del DEI. La Local Area Network (LAN) del DEI si estende fino al server centrale a ai *Remote Access Gateways* locali (Figura 1.2), invece i NCG fanno parte di un'altra LAN non visibile dalla rete del DEI. Ad esempio un messaggio di *ping* proveniente dalla rete del DEI potrà raggiungere al massimo un *Remote Access Gateway* locale. La comunicazione tra un sensore ed il server centrale avviene quindi attraverso il seguente percorso: un messaggio inviato da un nodo attraverso la porta seriale arriva al NCG di zona, il quale lo inoltra al

Remote Access Gateways e attraverso il tunnel VPN può giungere al server centrale. L'indirizzamento di un sensore nella rete avviene mediante la creazione di un socket, composto dall'indirizzo IP del NCG a cui è connesso e da una porta che viene creata in modo casuale dal NCG ogni qualvolta si desideri iniziare una comunicazione. Quando avviene la fine di una comunicazione infatti, il socket usato scade e non è più utilizzabile per una successiva comunicazione. É necessaria una nuova richiesta al NCG che restituirà una nuovo socket per avviare una nuova comunicazione.
Un qualsiasi processo presente nel server centrale può ora mettersi in comunicazione con un sensore attraverso il socket creato.

La prospettiva futura è quella invece di passare dall'indirizzamento IPv4 all'indirizzamento IPv6 [8]. Una delle differenze più importanti tra i due protocolli è l'ampio spazio dedicato all'indirizzamento di IPv6 rispetto al predecessore. IPv6 riserva infatti 128 bit rispetto ai 32 bit riservati da IPv4. Più in particolare si andrà ad utilizzare il protocollo 6LowPan [3], acronimo di IPv6 over Low power Wireless Personal Area Networks. 6LowPan è il nome del gruppo di lavoro che si occupa degli aspetti internet del Internet Enginering Task Force (IETF). Il gruppo ha definito un meccanismo di compressione e incapsulamento che permette ai pacchetti IPv6 di essere inviati e ricevuti in reti basate sullo standard IEEE 802.15.4. Questo standard è stato concepito per regolamentare il livello fisico ed il livello MAC di reti per uso personale che lavorano con basse velocità di trasferimento dati, ad esempio una WSN.
Con IPv6 non sarà necessario ricorrere all'uso di un socket per identificare un nodo. La prima parte dell'indirizzo IP verrà formata dall'indirizzo IP del NCG, e gli ultimi 16 bit saranno formati dal valore identificativo del singolo nodo. Ogni sensore infatti è univocamente identificato da un indirizzo interno, chiamato *TOS_NODE_ID*.

1.3 Obiettivo della tesi

L'obbiettivo della tesi è quello di creare uno strumento per il monitoraggio del traffico in una WSN, più precisamente si è interessati a verificare le prestazioni di un qualsiasi algoritmo di routing utilizzato nelle applicazioni montate sui sensori. Lo strumento dovrà infatti essere indipendente da quest'ultime e anche dalla scelta dell'algoritmo di routing. Dovrà essere in grado di evidenziare eventuali perdite di pacchetti, colli di bottiglia, ritardi, copertura dei nodi.
Il tutto verrà inserito in uno specifico livello all'interno di un'applicazione

web che verrà descritta in seguito.
Questo strumento ha lo scopo di evidenziare eventuali problemi o malfunzionamenti che si possono creare in una rete di sensori. Ad esempio problemi relativi all'instradamento dei pacchetti (routing), oppure malfunzionamenti relativi ai nodi (nodi danneggiati), interferenza con altri sistemi radio (WiFi), possibile cattiva dislocazione spaziale dei sensori, tempi di latenza elevati.

Capitolo 2

Strumenti utilizzati

In questo capitolo verranno presentati tutti gli strumenti utilizzati per portare a termine il lavoro. Innanzitutto verrà presentata l'applicazione web chiamata Fandango e in seguito il framework GWT per la gestione grafica, MySQL per la gestione del database, JDBC per la connessione tra Java e MySQL.

2.1 Fandango

Come anticipato in precedenza è possibile il monitoraggio e il controllo remoto del testbed mediante un'applicazione web grafica che utilizza gli strumenti forniti da GWT [4] dal lato client e Apache Tomcat [5] dal lato server.

Come si vede in Figura 2.1, è possibile visualizzare tramite mappe, la posizione esatta di ogni sensore nel dipartimento ed inoltre è possibile inviare ad essi comandi direttamente tramite l'interfaccia grafica. Si noti che il componente che gestisce le mappe è lo stesso usato da Google Maps.
Nella colonna di sinistra è presente una lista di *plugin* dove ognuno svolge un compito diverso all'interno dell'applicazione; è una struttura espandibile ed è possibile creare un nuovo *plugin* quando si voglia aggiungere delle nuove funzionalità.
Attualmente sono presenti i seguenti *plugin*:

- Maps
- Sensors
- Preferences
- Firmware

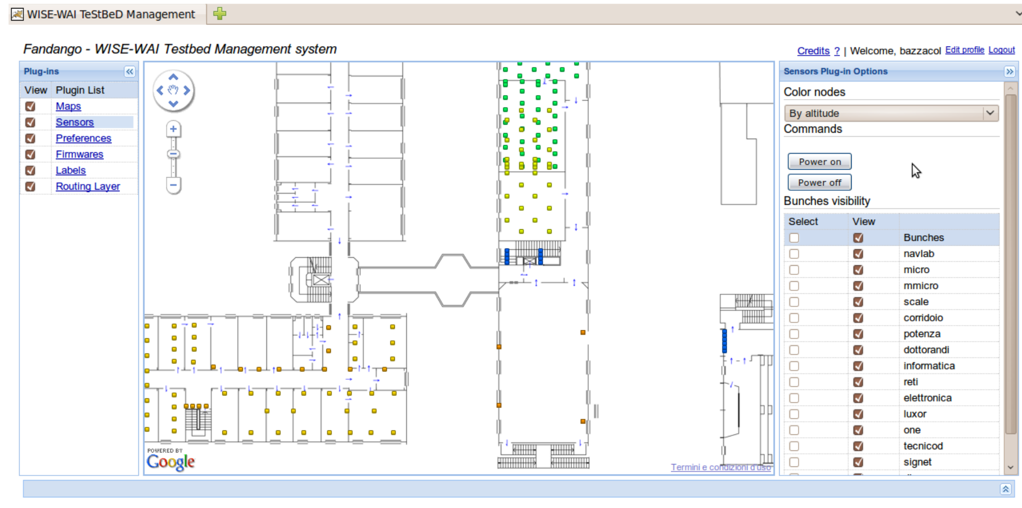


Figura 2.1: Applicazione Web

- Labels
- Reservations
- Permission
- Routing

Cliccando sul *plugin* interessato verranno visualizzate le relative opzioni nella colonna di destra. *Maps* si occupa di caricare la mappa desiderata, *Sensors* permette di visualizzare e/o selezionare i sensori desiderati, in modo manuale o a seconda del gruppi di appartenenza, *Preferences* permette di salvare delle visualizzazione preferite, *Firmware* si occupa del caricamento delle applicazioni nei sensori, *Reservation* permette di riservare i nodi per un periodo di tempo (è necessario riservare un nodo prima di interagire con esso), *Permission* permette di differenziare i permessi a seconda dell'utente, *Routing Layer* è il lavoro svolto in questo documento che permette la visualizzazione dei percorsi dei pacchetti, i ritardi, le perdite, gli accodamenti.
Gli oggetti che rappresentano i sensori non sono solo delle immagini incollate sulla mappa, ma sono degli oggetti attivi, chiamati Widget, con i quali si può interagire, cliccandoli o selezionandoli. Ad esempio è possibile selezionare un'insieme di sensori, riservarli, accenderli o spegnerli, e caricare il

software su di essi.
Tutte le informazioni relative ai sensori, quali posizionamenti sulla mappa, raggruppamenti, NCG di appartenenza, e le informazioni relative agli utenti, quali privilegi, software utilizzati, prenotazioni effettuate sono racchiuse in un database relazionale chiamato *wsn*, mentre tutte le informazioni relative al traffico che saranno necessarie per il *plugin* sviluppato sono racchiuse in un altro database relazionale che verrà illustrato in dettaglio in seguito.
Il lato client dell'applicazione comunica continuamente con il lato server, interrogando il database attraverso le librerie fornite da JDBC.

2.2 Google Web Toolkit

In questa sezione verranno elencate in modo conciso le caratteristiche principali di GWT. GWT è un'insieme di strumenti open source di sviluppo per creare e ottimizzare applicazioni web. Il suo obbiettivo è quello di permettere anche agli sviluppatori meno esperti di creare velocemente applicazioni robuste e con alte prestazioni.
Le applicazioni GWT possono essere eseguite in 2 modalità:

- *Development mode*: l'applicazione viene eseguita come bytecode Java all'interno della Java Virtual Machine (JVM). Questa modalità è usata nella fase di sviluppo e debug.
- *Web mode*: l'applicazione viene eseguita come puro JavaScript e HTML, compilato dalla sorgente Java. Questa modalità è usata nella fase di distribuzione.

GWT permette quindi di effettuare il debug delle applicazioni sviluppate in linguaggio Java, fornendo un plugin per Eclipse o per altri *Integrated Development Enviroment* (IDE). É possibile infatti effettuare il debug impostando dei *breakpoint* all'interno del codice durante l'utilizzo dell'applicazione web.

Caratteristiche

- Utilizzo di componenti GUI (graphical user interface): buttons, textbox, listbox, combobox, visual tree structures, drag-and-drop, windows.
- Semplice meccanismo di RPC (Remote procedure call).
- Supporto per tutte le funzionalità di debug con linguaggio Java.
- Open-Source.

- Sviluppo delle applicazioni in modo puramente object-oriented.
- Il codice JavaScript che il compilatore GWT genera può essere adattato per diventare di dimensioni ridotte per facilitarne il download.
- Le librerie per GWT sono disponibili da Google.

2.3 MySQL

MySQL è un software che fornisce un sistema di gestione di database relazionali, ossia un *RDBMS*, acronimo che deriva da *Relational DataBase Management System*. È composto da un client con interfaccia a caratteri e un server, entrambi disponibili sia per sistemi Unix come GNU/Linux che per Windows, anche se prevale un suo utilizzo in ambito Unix. Supporta la maggior parte della sintassi SQL e si prevede in futuro il pieno rispetto dello standard ANSI. Possiede delle interfacce per diversi linguaggi, compreso un driver ODBC, due driver Java e un driver per Mono e .NET. Esistono diversi tipi di MySQL Manager, ovvero di strumenti per l'amministrazione di MySQL. Uno dei programmi più popolari per amministrare i database MySQL è phpMyAdmin (richiede un server web come Apache HTTP Server ed il supporto del linguaggio PHP). Si può utilizzare facilmente tramite un qualsiasi browser. In alternativa la stessa MySQL AB, ovvero l'azienda sviluppatrice del codice di MySQL, offre programmi quali MySQL Administrator (amministrazione del database, degli utenti, operazioni pianificate, carico del server, ...) e MySQL Query Browser (esecuzione di svariati tipi di query), MySQL Migration Toolkit per importare da altri DBMS. Per il disegno e la modellazione di database MySQL esiste MySQL Workbench: integra il disegno, la modellazione, la creazione e l'aggiornamento di database in un unico ambiente di lavoro.

2.3.1 Tipi di tabelle (*storage engine*)

In MySQL una tabella può essere di diversi tipi (o storage engine). Ogni tipo di tabella presenta proprietà e caratteristiche differenti (transazionale o meno, migliori prestazioni, diverse strategie di blocco dei dati (*locking*), funzioni particolari, ecc). Esiste poi un'API che si può utilizzare per creare in modo relativamente facile un nuovo tipo di tabella, che poi si può installare senza dover ricompilare o riavviare il server.

Tipi di tabella più utilizzati

- MyISAM
- InnoDB
- Merge

MyISAM è un motore di immagazzinamento dei dati estremamente veloce e richiede poche risorse, sia in termini di memoria RAM, sia in termini di spazio su disco. Il suo limite principale rispetto a InnoDB, consiste nel mancato supporto delle transazioni che verranno esaustivamente descritte nella Sezione seguente. Questo supporto è una caratteristica indispensabile per il lavoro svolto e per questo tutte le tabelle che costituiscono il database sono di tipo InnoDB.

2.3.2 Transazioni

Le transazioni, ovvero le unità elementari (logiche) di lavoro svolte da un'applicazione, sono una delle caratteristiche fondamentali di un database relazionale. Sono quel meccanismo che consente di mantenere, durante la vita della base dati, tutte le informazioni consistenti. Durante le operazioni di modifica (scrittura, aggiornamento, cancellazione) delle tabelle le transazioni fanno sì che nessuna di queste avrà effetto fino a quando i nuovi valori non verranno effettivamente scritti sulla base dati stessa. È indispensabile avere un meccanismo di questo tipo quando siamo nel caso in cui più processi vanno a interagire con la stessa tabella e si vedrà in seguito dove questa caratteristica sarà molto importante ai fini del lavoro svolto.
Ogni transazione viene specificata da un'istruzione di inizio e una di fine che racchiudono al loro interno una sequenza di operazioni, che fanno quindi parte della stessa unità di lavoro.
Siamo in grado di definire una transazione tramite i comandi:

- BEGIN TRANSACTION, che specifica l'inizio della transazione,
- COMMIT TRANSACTION per richiedere che tutti gli aggiornamenti effettuati vengano materializzati nella base dati,
- ROLLBACK TRANSACTION per richiedere, al contrario, che tutti gli aggiornamenti vengano annullati (come se non fosse mai esistita quella o quelle modifiche che si stavano facendo).

Per il sistema quindi, che sia SQL Server o qualsiasi altro database relazionale, una transazione è una singola unità di esecuzione caratterizzata da

proprietà "acide" (dall'acronimo inglese ACID), ovvero **Atomicity, Consistency, Isolation, e Durability** (**A**tomicità, **C**oerenza, **I**solamento e **D**urabilità).

Perché le transazioni operino in modo corretto sui dati è necessario che i meccanismi che le implementano soddisfino queste quattro proprietà:

- **atomicità**: la transazione è indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla, non sono ammesse esecuzioni parziali. Grazie a questa caratteristica in presenza di qualsiasi guasto (errore, interruzione, ...) prima del commit, il motore del database eliminerà tutti gli effetti della transazione stessa ripristinando lo stato iniziale, come se nulla fosse successo;

- **coerenza**: quando inizia una transazione il database si trova in uno stato coerente e quando la transazione termina il database deve essere in uno stato coerente, ovvero non deve violare eventuali vincoli di integrità, quindi non devono verificarsi contraddizioni tra i dati archiviati nel database;

- **isolamento**: ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione;

- **durabilità**: detta anche **persistenza**, si riferisce al fatto che una volta che una transazione abbia richiesto un commit work, i cambiamenti apportati non dovranno essere più persi. Per evitare che nel lasso di tempo fra il momento in cui la base di dati si impegna a scrivere le modifiche e quello in cui li scrive effettivamente si verifichino perdite di dati dovuti a malfunzionamenti, vengono tenuti dei registri di log dove sono annotate tutte le operazioni sul database.

Problemi di consistenza e livelli di isolamento

Lo standard SQL-99 definisce quattro livelli di isolamento, ovvero delle regole che specificano come una transazione deve essere isolata da modifiche apportate da altre transazioni, tutti supportati da SQL Server. Queste regole si rendono necessarie per evitare potenziali problemi come:

- **Dirty reads** (letture sporche): si potrebbero leggere dati che, non ancora committati, potrebbero subire un rollback (ovvero si potrebbero leggere dati mai esistiti)

- **Nonrepeatable reads** (letture inconsistenti): si potrebbe, all'interno di una transazione, leggere dati che, prima di essere riletti, vengano modificati da un'altra transazione
- **Phantoms** (dati fantasma): si potrebbe, all'interno di una transazione, leggere quante righe stanno in una tabella e, prima di apportare un update su tutte queste un'altra transazione effettua, su quella stessa tabella, un'operazione di insert (di fatto si farebbe un update su un numero differente di righe rispetto a quelle calcolate inizialmente)
- **Lost updates** (perdita di aggiornamento): si potrebbe, all'interno di una transazione T1 leggere una colonna e, prima di modificarne il valore, avere una seconda transazione T2 che effettui una differente modifica (che, quindi, andrebbe persa venendo soprascritta dalla transazione T1).

SQL Server consente di specificare un livello di isolamento per ogni transazione. Tale impostazione andrà quindi a determinare il tempo per il quale sarà necessario mantenere dei blocchi che consentono a chi ne fa richiesta di utilizzare o meno delle informazioni, su oggetti e risorse. I livelli di isolamento a disposizione sono:

- Read uncommited
- Read committed
- Repeatable read
- Serializable

Il meccanismo di lock

Affinché tutto funzioni, SQL Server gestisce un meccanismo di blocco per garantire le proprietà ACID delle transazioni. Le operazioni di blocco possono essere di tipo condiviso, esclusivo o di aggiornamento e possono avere una granularità a livello di riga, di singola pagina di dati, di intera tabella o di intero database.
Il blocco condiviso viene applicato durante operazioni di lettura e consente, ad altre transazioni, di avere sugli stessi dati un blocco dello stesso tipo (le operazioni di lettura non bloccano altre letture).
Il blocco esclusivo viene applicato in fase di scrittura e, per poter ottenere un blocco di questo stesso tipo, nessuna altra transazione può avere operazioni in corso sugli stessi dati (le scritture bloccano altre scritture sugli stessi dati).

L'ultimo blocco disponibile, di aggiornamento, funziona come una sorta di "diritto di precedenza". Un processo che richiede una modifica su dati bloccati li marca con un blocco di aggiornamento che rimarrà tale fino a che non verrà elevato a blocco esclusivo.
Riepilogando i meccanismi di blocking ricordiamo che:

- Le scritture bloccano altre scritture. Sempre
- Le scritture bloccano le letture in Read Committed o con livelli superiori
- Le letture bloccano le scritture in Repeatable Read o con livelli superiori

Sicuramente costruire transazioni solide e non limitarsi ad utilizzare il livello di default potrebbe complicare la costruzione del database e del suo codice (procedure, funzioni), ma è un requisito irrinunciabile per mantenere la corretta integrità dei dati. Infine è importante sottolineare alcuni aspetti pratici da tenere in considerazione:

- Utilizzare il miglior (meno caro) livello di isolamento andando a includere, all'interno delle transazioni, solo le istruzioni che devono effettivamente far parte dell'unità di lavoro (durata)
- Evitare di aprire transazioni durante operazioni di analisi dei dati (la transazione, di fatto, non serve)
- Mantenere transazioni più brevi possibili
- Nelle transazioni accedere alla quantità minima di dati possibile (riducendo così al minimo i lock sulle risorse)

Il livello **Read Uncommitted**, da usare con la massima attenzione e cautela, consente di leggere dati che non sono ancora stati committati, e non offre quindi alcuna garanzia circa la presenza dei dati letti al termine della transazione.
Il livello **Read Committed**, impostato come livello di default in SQL Server, consente di vedere e utilizzare solo dati che sono stati richiesti e quindi effettivamente materializzati nella nostra base dati. Consente un elevato livello di coerenza dei dati che potrebbe portare a pagare della concorrenza nell'accesso alle risorse.
Il livello **Repeatable Read** preserva dalla modifica da parte di altri utenti tutti i dati utilizzati dalla transazione. Consente l'inserimento di nuovi dati e quindi potrebbe portare ad avere un numero di righe differente da quello

eventualmente letto e, quindi, un risultato differente.
L'ultimo livello, **Serializable**, è il più restrittivo disponibile. Non consente modifiche, né inserimenti, con i dati che sono coerenti con i criteri di query in uso. Tale livello impedisce quindi, andando a bloccare a livello di dati ed anche a livello di indici, che il resultset interessato da un'istruzione subisca delle modifiche.
È importante sottolineare come non sia possibile arrestare le transazioni, né influenzare le proprietà di atomicità, consistenza e durata. L'unica possibilità è quella di intervenire sul livello di isolamento.
Nella Tabella 2.1 si vede un riepilogo di come i vari livelli di isolamento consentano di vedere o no dati fantasma, letture ripetibili o sporche:

	Dirty Read	**Non-Repeatable Read**	**Phantom Read**
Read uncommitted	SI	SI	SI
Read committed	NO	SI	SI
Repeatable read	NO	NO	SI
Serializable	NO	NO	NO

Tabella 2.1: Tabella riassuntiva

2.4 Java DataBase Connectivity

JDBC è un connettore per database che consente l'accesso alle basi di dati da qualsiasi programma scritto con il linguaggio di programmazione Java, indipendentemente dal tipo di DBMS utilizzato. È costituita da una API che serve ai client per connettersi a un database, fornisce metodi per interrogare e modificare i dati, è orientata ai database relazionali e agli oggetti.

L'architettura di JDBC prevede l'utilizzo di un *driver manager*, che espone alle applicazioni un insieme di interfacce standard e si occupa di caricare a *run-time* i driver opportuni per pilotare gli specifici DBMS. Le applicazioni Java utilizzano le JDBC API per parlare con il JDBC driver manager, mentre il driver manager usa le JDBC driver API per parlare con i singoli driver che pilotano i DBMS specifici.
Ogni DBMS ha il proprio driver che può essere presente all'interno del pacchetto oppure prodotto da terze parti.
I driver JDBC si dividono in 4 categorie:

1. **Tipo 1.** è denominato JDBC-ODBC Bridge. I driver di questa categoria si connettono ai database passando per ODBC. Il driver non fa

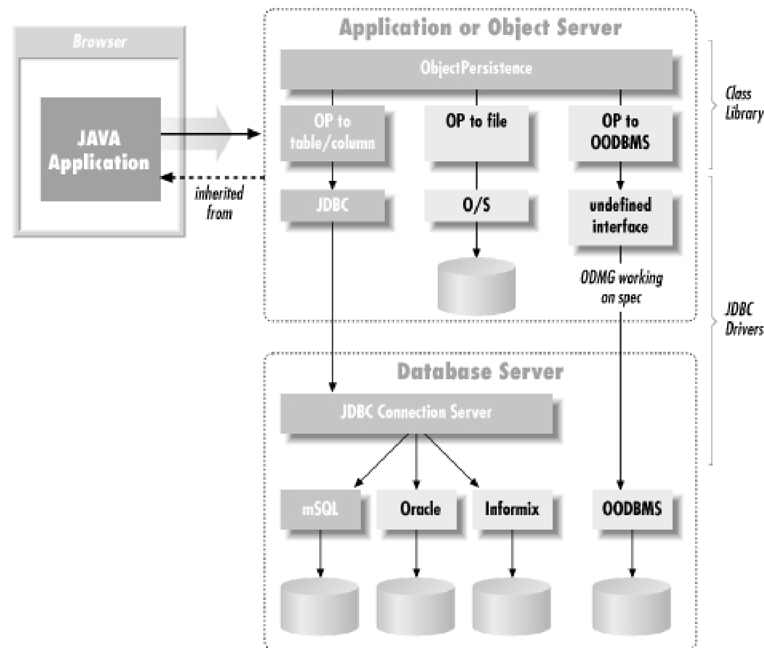


Figura 2.2: JDBC

altro che convertire chiamate JDBC in ODBC. All'interno delle API JDBC è presente un bridge JDBC-ODBC sviluppato da Sun. I driver di tipo 1 in genere non brillano in prestazioni e necessitano che su ogni macchina client sia installato il corrispondente driver ODBC.

2. **Tipo 2.** questa tipologia di driver converte le chiamate JDBC in chiamate API native del DBMS per il quale drive è stato progettato. In termini di prestazioni un driver di tipo 2 è nettamente superiore a un driver di tipo 1. Anch'esso però presenta due svantaggi: non essendo scritto interamente in Java, il proprio bytecode non può essere eseguito su diverse piattaforme, inoltre è necessaria l'installazione del client-side del DBMS sulla macchina su cui gira l'applicazione.

3. **Tipo 3.** il driver di tipo 3 è scritto interamente in Java e usa un determinato protocollo di rete (per esempio il TCP/IP) per comunicare con un server intermedio (middleware server) che si occuperà di interagire direttamente con il DBMS. Il server consente la connessione delle applicazioni client Java con diversi DBMS in modo totalmente trasparente. I net-driver (così sono chiamati i driver di questo tipo) sono leggeri, veloci, multi piattaforma e facilmente scaricabili dalla rete. Sulla

macchina che ospita il client non è necessaria l'installazione di nessun software aggiuntivo.

4. **Tipo 4.** è un driver scritto totalmente in java che usa un protocollo di rete proprietario per connettersi direttamente al DBMS. A differenza dei driver di tipo 3 non richiede un server intermedio. Questo tipo di driver è veloce multi-piattaforma e installabile automaticamente dalla rete. Sulla macchina che ospita il client non è necessaria l'installazione di nessun software aggiuntivo.

Il connettore usato per MySQL è chiamato **MySQL Connector/J** ed è un driver di tipo 4 scritto interamente in Java che converte le chiamate JDBC dentro il protocollo di rete usato dal database MySQL. Nel Codice 2.4 si può vedere un breve esempio nel quale viene utilizzato il connettore JDBC.

Codice 2.1: Esempio Java di utilizzo del connettore

```
try {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    Connection conn = null;
    conn = DriverManager.getConnection("jdbc:mysql://" +
    "localhost/tinyos?user=root&password=******");
    Statement stmt = conn.createStatement();
    String SQL = "SELECT * FROM Packet";
    ResultSet result = stmt.executeQuery(SQL);
    while(result.next()) {
        System.out.println(result.getInt("PacketID"));
    }
}catch (SQLException ex) {
        System.out.println(ex.getMessage());
        System.out.println(ex.getSQLState());
        System.out.println(ex.getErrorCode());
    }
```

Come si può notare dal Codice 2.1 l'interazione con il database avviene in 5 fasi:

1. Viene caricato il driver JDBC mediante l'istruzione *Class.forName(com.mysql.jdbc.Driver).newInstance()*

2. Viene creata la connessione al database mediante l'istruzione *conn = DriverManager.getConnection()*

3. Viene creata un'istanza dello Statement mediante l'istruzione *Statement stmt = conn.createStatement()*

4. Viene eseguita la query, in questo esempio si tratta di una query di selezione; SQL contiente l'istruzione in linguaggio SQL da eseguire. *ResultSet result = stmt.executeQuery(SQL)*

5. All'interno del ResultSet sono presenti i records risultati dall'esecuzione della query, che andranno poi elaborati.

Capitolo 3

Protocolli di routing per Wireless Sensors Networks

In questo capitolo vengono illustrate le caratteristiche e i requisiti dei protocolli di routing per le WSN.
Data la similarità di queste reti con le reti Ad Hoc verranno illustrati precedentemente gli algoritmi di routing adottati in questo tipo di reti e verranno spiegati quali sono i problemi relativi all'adozione di questi algoritmi per le WSN.
Verrà infine effettuata una classificazione degli algoritmi di routing per le WSN.

3.1 Mobile Ad Hoc Networks

Una Mobile Ad Hoc Networks (MANET) [9] é un sistema autonomo di terminali mobili, connessi con collegamenti di tipo wireless che sono uniti formando un grafo di forma arbitraria. Tali terminali sono liberi di muoversi casualmente e di auto organizzarsi arbitrariamente, sebbene la topologia wireless vari rapidamente ed in modo imprevedibile. Tale rete può operare da sola oppure essere connessa alla rete Internet. Le reti Ad-hoc vengono costruite all'occorrenza ed utilizzate in ambienti estremamente dinamici, non necessariamente con l'aiuto di una infrastruttura già esistente, come ad esempio dopo catastrofi naturali, durante conflitti militari o altre situazioni d'emergenza.
In una rete ad hoc, i nodi non hanno quindi alcuna conoscenza a priori della topologia della rete nella quale si trovano, per cui devono scoprirla comunicando con gli altri nodi. Generalmente ogni nodo annuncia la sua presenza nella rete ed ascolta la comunicazione tra gli altri nodi, che dunque diventano

conosciuti. Col passare del tempo ogni nodo acquisisce la conoscenza di tutti i nodi della rete e di uno o più modi per comunicare con loro.

Gli algoritmi di routing in reti di questo tipo devono:

- Assicurarsi che le tabelle di routing siano ragionevolmente piccole, anche alla luce delle risorse ridotte delle quali spesso dispongono i nodi in una rete ad hoc;
- Riuscire a scegliere il miglior percorso per raggiungere gli altri nodi (in base a vari parametri, come la velocità, l'affidabilità e l'assenza di congestione);
- Tenere le proprie tabelle di routing aggiornate nel caso la topologia di rete cambi;
- Raggiungere il funzionamento ottimale in poco tempo e inviando un numero esiguo di pacchetti;
- Eventualmente offrire path multipli per raggiungere una destinazione, magari ordinando i path in ordine crescente di costo.

Gli algoritmi di routing per MANET possono essere classificati in questo modo:

- *Proactive protocols*: La caratteristica principale dei protocolli proattivi è quella di cercare di mantenere costantemente informazioni sui percorsi di routing tra ogni coppia di nodi presenti nella rete propagando aggiornamenti ad intervalli regolari (routing overhead alto). Tutti i cammini possibili sono calcolati indipendentemente dal loro effettivo uso. Il vero vantaggio dei protocolli di questa famiglia è che il percorso verso la destinazione è disponibile in ogni momento e non si verificano ritardi quando l'applicazione deve inviare messaggi. Protocolli di questo tipo sono:
  Destination-Sequenced Distance-Vector (DSDV) [10], Optimized Link State Routing (OLSR) [11].
- *Reactive protocols*: Questa categoria di protocolli prevede di calcolare i percorsi di routing solo al momento del bisogno, ovvero quando è necessaria una trasmissione di dati, e tali informazioni sono mantenute soltanto fin quando sono necessarie. Perciò questo approccio non ha bisogno di tabelle di routing su ogni nodo e in assenza di traffico l'attività di routing è totalmente assente anche se la rete varia la propria topologia. Questi protocolli vengono chiamati anche protocolli

on-demand. Un protocollo reattivo è caratterizzato dalle seguenti fasi: Route Discovery, Route Maintenance e Route Deletion. Protocolli di questo tipo sono:
Dynamic Source Routing (DSR) [12], Ad hoc On Demand Distance Vector (AODV) [11], Temporally Ordered Routing Algorithm (TORA) [13].

- *Hibrid protocols*: I protocolli ibridi combinano gli approcci proattivi e reattivi cercando di mettere insieme i vantaggi di entrambe. In pratica, il calcolo del percorso di routing avviene in parte in modo proattivo e parte in modo reattivo: i percorsi verso i nodi vicini vengono calcolati in anticipo come previsto nei protocolli proattivi, mentre i percorsi verso nodi lontani vengono calcolati on demand sfruttando le funzionalità reattive. Protocolli di questo tipo sono:
Zone-Based Hierarchical Link State Routing Protocol (ZRP) [14] e Location Aided Protocol (LAR) [15].

3.2 Wireless Sensors Networks

Sfortunatamente, molti degli algoritmi usati nelle reti ad hoc non sono compatibili con i requisiti delle WSN.
I principali motivi derivano dal fatto che:

- Il numero di nodi che compongono una rete di sensori può essere di alcuni ordini di grandezza maggiore rispetto al numero di nodi in una rete ad hoc;
- I nodi in una WSN sono disposti con un'alta densità;
- I nodi sono soggetti a guasti;
- La topologia di una rete di sensori può cambiare frequentemente a causa di guasti ai nodi o della loro mobilità.
- I nodi utilizzano un paradigma di comunicazione broadcast mentre la maggior parte delle reti ad hoc sono basate su una comunicazione di tipo punto-punto;
- I nodi sono limitati rispetto ad alimentazione, capacità di calcolo e memoria;
- I nodi necessitano di una stretta integrazione con le attività di rilevamento.

Per questi motivi, questa tipologia di rete necessita di algoritmi pensati e realizzati in maniera specifica al fine di soddisfare le precedenti esigenze.

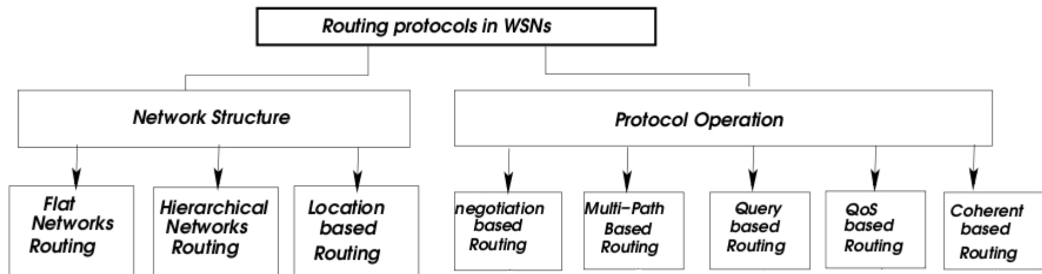


Figura 3.1: Classificazione dei protocolli di routing per WSNs.

In generale i protocolli di routing utilizzati in reti di sensori si possono classificare come si vede in Figura 3.1 [16].
Si possono suddividere in *flat-based* routing, *hierarchical-based* routing, e *location-based* routing a seconda della struttura della rete.
Nei protocolli *flat-based* routing, tutti i nodi svolgono lo stesso ruolo nella rete, possiedono le stesse funzionalità e collaborano insieme per rilevare la struttura della rete. In *hierarchical-based* routing, esistono dei nodi con funzioni particolari e/o con più carico energetico che sono utilizzati per elaborare e trasmettere le informazioni. Mentre i nodi a basso consumo energetico possono essere utilizzati per effettuare le misurazioni in prossimità dell'obiettivo. La creazione di *cluster* e l'assegnazione di compiti specifici ai nodi più potenti può contribuire alla scalabilità del sistema, all'aumento della durata di vita e dell'efficienza energetica. In *location-based* routing, le posizioni dei nodi sono sfruttate per inoltrare i dati nella rete. Le posizioni possono essere stimate dalla potenza del segnale ricevuto dai nodi vicini oppure da un segnale GPS se previsto. Questi algoritmi richiedono ai nodi nei quali non vi è attività di entrare in fase di *sleep*, per salvaguardare la durata delle fonti di energia.
I protolli di routing si possono inoltre suddividere in *multipath-based*, *query-based*, *negotiation-based*, *QoS-based*, e *coherent-based* routing a seconda delle operazioni effettuate dal protocollo.
I protocolli *multipath-based* routing sono a conoscenza di percorsi diversi per inoltrare i pacchetti. Questo aumenta l'affidabilità della rete, a discapito di una maggior traffico di segnalazione e consumo energetico. In *query-based* routing, un nodo che necessita di un informazione, invia la richiesta nella rete,

e il nodo che possiede questa informazione la invia al nodo che ha iniziato la richiesta. In *negotiation-based* routing, l'idea principale è quella di eliminare le informazioni duplicate e prevenire che vengano inviati dati ridondanti ad uno stesso sensore, conducendo una trasmissione di messaggi di segnalazione prima di trasmettere i dati reali. In *QoS-based* routing, i protocolli devono soddisfare determinate QoS metriche (ritardo,energia,banda,ecc.), bilanciando la rete tra consumo energetico e qualità di dati trasmessi. In *coherent-based* routing, in generale i sensori collaborano tra di loro per processare i dati che circolano in rete. Nei protocolli di routing non coerenti i nodi processano i dati localmente per poi trasmetterli ad altri nodi. Nei protocolli coerenti i nodi trasmettono i dati a dei nodi speciali chiamati aggregatori che si incaricano di processare tutti i messaggi. Per migliorare l'efficienza energetica vengono normalmente scelti algoritmi coerenti.

3.3 Valutazione delle prestazioni di un algoritmo di routing

In questo capitolo verranno descritti gli strumenti utilizzati per fare una valutazione di un algoritmo di routing in una *ad-hoc* WSN. Inanzitutto quando si vuole fare una valutazione prestazionale di un algoritmo bisogna definire sulla base di quali metriche si andranno a confrontare i risultati; i parametri principali che si andranno a studiare in questo documento sono:

- **Packet Delivery Ratio**: è il rapporto tra i pacchetti che giungono correttamente a destinazione (sink) e il numero totale di pacchetti trasmessi. E' una delle metriche più importanti perché lo scopo principale della trasmissione di un pacchetto in una WSN è che giunga correttamente a destinazione.

- **End-To-End Delay**: è il tempo necessario affinché un pacchetto venga trasmesso attraverso la rete da una sorgente a una destinazione.

- **Forward Delay**: è il ritardo di inoltro di un nodo che si trova tra una sorgente e una destinazione e deve inoltrare un pacchetto.

- **Normalized Routing Load**: è il numero di pacchetti di routing trasmessi da tutta la rete affinché un pacchetto dati arrivi a destinazione.

3.4 Descrizione dell'algoritmo di Routing testato e dell'applicazione usata negli esperimenti

In questa sezione verrà descritto il funzionamento dell'algoritmo di routing testato e verrà descritta l'applicazione utilizzata nei test. Per capire il funzionamento di entrambi verranno prima presentate le strutture dei pacchetti generati.

3.4.1 Formato dei pacchetti

Nel riquadro 3.3 si può vedere un esempio di pacchetti generati da sensori che montano il sistema operativo *TinyOS* [6]. Ogni riga rappresenta un pacchetto, più o meno lunga a seconda della dimensione del messaggio. Ogni coppia di caratteri rappresenta 1 byte, e rappresentano dei valori esadecimali, come si vede dalla presenza di lettere alfabetiche.

Codice 3.1: Esempio di traffico

```
00 FF FF 00 14 01 00 64 02
00 FF FF 00 17 01 00 64 02
00 00 28 80 17 0A 00 65 00 17 00 32 02 01 00 17 00 32
00 00 28 80 14 0A 00 65 00 14 00 3F 02 01 00 14 00 3F
00 00 28 80 17 0A 00 65 00 17 00 32 02 01 00 32 00 28
00 00 28 80 47 09 00 80 AA 00 1B 07 45 00 47 00 34
00 00 28 80 11 09 00 80 AA 00 1B 2E 30 00 22 00 27
```

I campi che compongono la struttura del messaggio sono i seguenti:

- **Destination address (DA)** (2 bytes)
- **Link source address (LSA)** (2 bytes)
- **Message length (MSGL)** (1 byte)
- **Group ID (GID)** (1 byte)
- **Active Message handler type (HID)** (1 byte)
- **Payload** (fino a 28 bytes)
- **Routing Tag** (4 bytes)
  - **Hop Source Address (HSA)** (2 bytes)
  - **Hop Destination Address (HDA)** (2 bytes)

Possiamo quindi interpretare i pacchetti nel riquadro 3.1 nel modo seguente come si vede in Figura 3.2:

		DA	LSA	MSGL	GID	HID	payload	HSA	HDA
Pkt1)	00	FF FF	00 14	01	00	64	02		
Pkt2)	00	FF FF	00 17	01	00	64	02		
Pkt3)	00	00 28	80 17	01	00	65	00 17 00 32 02 01	00 17	00 32
Pkt4)	00	00 28	80 14	0A	00	65	00 14 00 3F 02 01	00 14	00 3F
Pkt5)	00	00 28	80 17	0A	00	65	00 17 00 32 02 01	00 32	00 28
Pkt6)	00	00 28	80 47	09	00	80	AA 00 1B 07 45	00 47	00 34
Pkt7)	00	00 28	80 11	09	00	80	AA 00 1B 2E 30	00 22	00 27

Figura 3.2: Campi del pacchetto

I campi *Link Source Address* e *Destination Address* rappresentano sorgente e destinazione end-to-end della rotta, mentre i campi *Hop Source Address* e *Hop Destination Address*, che compongono il *RoutingTag*, rappresentano sorgente e destinazione del singolo Hop dove è avvenuta la trasmissione. *Msg Length* rappresenta la lunghezza in bytes del messaggio (*payload* + *RoutingTag*), *payload* racchiude l'informazione da trasmettere. *GroupID* serve per definire un gruppo di appartenza del nodo e *HandlerID* serve per differenziare il tipo di messaggio, ad esempio se è un messaggio di segnalazione (routing) o di applicazione. Tutti i campi hanno lunghezza fissa tranne il campo *payload*, che può variare a seconda dell'informazione da trasmettere. Si nota inoltre che il *RoutingTag* non è presente in tutti i tipi di messaggio. Per riconoscere i messaggi dove è presente viene impostato il primo bit del campo *Link Source Addr* a 1. Infatti dalla Figura 3.2 si nota che il valore del campo *Link Source Addr* ha come cifra iniziale il valore 8, nei pacchetti dove è presente il *RoutingTag* (in un indirizzo a 16 bit = 4 bytes, impostando il bit più significativo a 1 si ottiene come valore in esadecimale il valore 8).

In Figura 3.2 si può notare che nell'esempio di traffico del riquadro 3.1 esistono tre tipi diversi di pacchetto, in quanto esistono 3 valori diversi del campo *HandlerID*.
Nell'esempio, i tre diversi identificativi corrispondono a:

- 64: pacchetto *HelloMsg* dell'algoritmo di routing.
- 65: pacchetto *BestHopCountRoutingMsg* dell'algoritmo di routing.
- 80: pacchetto di applicazione.

Riassumendo i contenuti visti in questa sezione si è capito che:

- I campi evidenziati in viola nella Figura 3.2, sono sempre presenti in tutti i tipi di messaggio
- Il campo *HandlerID* è fondamentale per riconoscere il tipo di messaggio
- Il campo *payload* rappresenta l'informazione del messaggio ed è di lunghezza variabile.
- Il RoutingTag non è sempre presente (Es. Nei pacchetti Pkt1) e Pkt2) della Figura 3.2 il campo *Destination Addr* ha valore *FFFF*, che significa che il messaggio non ha una destinazione specifica ma è trasmesso in broadcast. Quindi in messaggi di questo tipo il RoutingTag non serve.)

3.4.2 Algoritmo di routing testato

I test svolti in questo documento sono stati effettuati con un algoritmo di routing sviluppato all'interno del gruppo di ricerca SIGNET del Dipartimento di Ingegneria dell'Informazione. L'algoritmo sviluppato è molto semplice ed è stato creato più che altro per dare delle dimostrazioni di come funzioni un algoritmo di routing, non ci si aspetta quindi delle alte prestazioni.
Il modulo di routing testato è chiamato *BestHopRoutingCount* ed è possibile connetterlo in modo trasparente ad una qualsiasi applicazione che necessiti di effettuare delle trasmissioni *multi-hop* ad esempio verso un *sink* lontano.
Questo algoritmo periodicamente invia messaggi di segnalazione nella rete per costruire e mantenere le rotte e quindi rientra nella categoria degli algoritmi proattivi. Inoltre si può classificare come un algoritmo gerarchico come descritto in Sezione 3.2. Nella rete testata infatti è sempre presente un nodo sink al quale fanno capo tutti i restanti nodi della rete. I messaggi di applicazione (dati) verranno sempre o destinati al sink o provenienti dal sink.
I messaggi trasmessi dall'algoritmo possono essere di due tipi:

- *BestHopCountRoutingMsg*
- *HelloMsg*

Codice 3.2: BestHopRoutingCount.h

```
enum {
        HOPCOUNT_ADV_MILLI = 30720
};

typedef nx_struct BestHopCountRoutingMsg {
```

```
  nx_uint8_t hopcount;
} BestHopCountRoutingMsg;

typedef nx_struct HelloMsg {
  nx_uint16_t id;
  nx_uint16_t next_hop;
  nx_uint8_t hopcount;
  nx_uint8_t age;
} HelloMsg;
```

Nel Codice 3.2 si può vedere il file *BestHopRoutingCount.h* che definisce i tipi di messaggio che utilizza l'algoritmo. Un messaggio di tipo *BestHop-CountRoutingMsg* viene trasmesso sempre in broadcast e serve essenzialmente per comunicare ai nodi vicini che il nodo è attivo. Un messaggio di tipo *HelloMsg* viene trasmesso sempre al sink e serve per comunicare ad esso il next hop usato dal nodo. Con quest'ultimo tipo di messaggio sia il sink sia i nodi intermedi saranno in grado di costruire le proprie tabelle di routing.

Il processo con cui avviene l'invio dei messaggi di segnalazione si ripete in modo periodico, con un periodo definito dalla variabile *HOPCOUNT_ADV_MILLI* visibile nel Codice 3.2 e la prima trasmissione avviene sempre dal nodo sink. Il calcolo delle rotte e dei next hop avviene tramite un parametro chiamato *hopcount* che tiene conto della distanza dal sink in termini di numero di hop. Per definizione il sink possiede sempre *hopcount*=0. All'accensione della rete il valore della variabile *hopcount* è impostato a infinito in tutti i nodi a parte il sink. La segnalazione inizia dal sink che invia in broadcast un messaggio di tipo *BestHopCountRoutingMsg* che contiene il valore della sua variabile *hopcount* che sarà uguale a 0. Tutti i nodi che ricevono un messaggio di questo tipo confrontano il valore del proprio *hopcount* con il valore ricevuto nel messaggio. Se come in questo caso il valore ricevuto è minore del valore in possesso, viene aggiornato il proprio *hopcount* incrementando di 1 il valore ricevuto. Si otterrà quindi che i nodi in copertura del sink avranno tutti *hopcount* pari a 1. Oltre a modificare il proprio valore di *hopcount*, ogni nodo memorizzerà il nodo da cui ha ricevuto il messaggio in una variabile chiamata *nexthop*. Si creerà quindi una relazione padre-figlio, dove il padre è colui che ha *hopcount* minore. Iterativamente anche i restanti nodi della rete invieranno in broadcast un messaggio di tipo *BestHopCountRoutingMsg* in modo che, alla fine del processo di inondazione della rete, ogni nodo avrà impostato correttamente un valore della variabile *hopcount* e sarà associato ad un padre. Ora un qualsiasi nodo foglia sarà a conoscenza di una rotta per arrivare al sink.
Dopo questa prima fase tutti i nodi, fuorché il sink, trasmetteranno a quest'ultimo un messaggio di tipo *HelloMsg*.

Codice 3.3: Esempio dei due tipi di messaggio scambiati dall'algoritmo

```
00 FF FF 00 14 01 00 64 02
00 FF FF 00 17 01 00 64 02
00 00 28 80 17 0A 00 65 00 17 00 32 02 01 00 17 00 32
00 00 28 80 14 0A 00 65 00 14 00 3F 02 01 00 14 00 3F
00 00 28 80 17 0A 00 65 00 17 00 32 02 01 00 32 00 28
```

Un nodo che invia un messaggio di questo tipo, comunica al sink che lo può raggiungere attraverso qualche altro nodo e che si trova ad una certa distanza (numero di hop). Nel riquadro 3.3 si può vedere un esempio dei 2 tipi di messaggio trasmessi dall'algoritmo. I primi due sono messaggi di tipo *BestHopCountRoutingMsg* i quali hanno lunghezza pari a 1 byte e provengono rispettivamente dai nodi *00 14* e *00 17* i quali hanno entrambi la variabile *hopcount* impostata a 2. I restanti sono messaggi di tipo *HelloMsg* i quali hanno lunghezza pari a 10 bytes e sono destinati tutti al sink (*00 28*). Ad esempio il primo dei tre, proviene dal nodo *00 17* e comunica al sink che lo può raggiungere tramite il nodo *00 32* e si trova a distanza pari a 2 Hop.
Grazie a questi messaggi il sink può costruire la sua tabella di routing ed è conoscenza di una rotta per comunicare con i nodi nella rete. Si può vedere un esempio nel Codice 3.4.

Codice 3.4: Esempio di composizione di una tabella di routing

```
TABELLA DI ROUTING
[0] dst:0x19 nh:0x3d
[1] dst:0x3 nh:0x31
[2] dst:0x15 nh:0x37
[3] dst:0x40 nh:0x3d
[4] dst:0x1b nh:0x9
[5] dst:0x39 nh:0x3d
[6] dst:0x45 nh:0x9
[7] dst:0x20 nh:0x9
[8] dst:0x35 nh:0x9
[9] dst:0x31 nh:0x39
[10] dst:0x1d nh:0x35
```

Ogni riga della tabella è formata da due campi: il campo *dst* che rappresenta la destinazione conosciuta e il campo *nh* (next hop) che rappresenta il nodo a cui trasmettere il messaggio per arrivare alla destinazione. I nodi sono identificati univocamente nella rete da una variabile interna chiamata *TOS_NODE_ID*. I valori dei campi *dst* e *nh* rappresentano il valore di questa variabile in esadecimale.
Quando il nodo sink necessita di trasmettere un messaggio ad un certo nodo nella rete andrà a ricercare nella tabella se esiste una riga che contiene nel campo destinazione (*dst*) il valore del *TOS_NODE_ID* corrispondente a quel nodo. Se viene trovato il valore, il messaggio verrà instradato verso il nexthop corrispondente. Se la ricerca da esito negativo il sink instraderà il pacchetto verso il nodo che è presente nella variabile *nexthop*. Si ricorda che il

contenuto di questa variabile rappresenta la provenienza dell'ultimo messaggio di tipo *BestHopCountRoutingMsg* ricevuto. In questo modo anche se in tabella non è presente la destinazione, il sink o qualsiasi altro nodo, possiede sempre un possibile nexthop. L'evento di ricerca negativa si verifica perché la dimensione della tabella è limitata, quindi una volta raggiunta la dimensione massima i nuovi inserimenti andranno a sovrascrivere le righe vecchie. É possibile comunque variare questa dimensione impostando la variabile associata.
In Figura 3.3 e Figura 3.4 è rappresentato come avviene l'inondazione della rete in due fasi. La prima, che consiste nella trasmissione dei messaggi di tipo *BestHopRoutingCountMsg* a partire dal sink, e la seconda che consiste nella trasmissione da parte dei nodi periferici dei messaggi di tipi *HelloMsg* verso il sink.

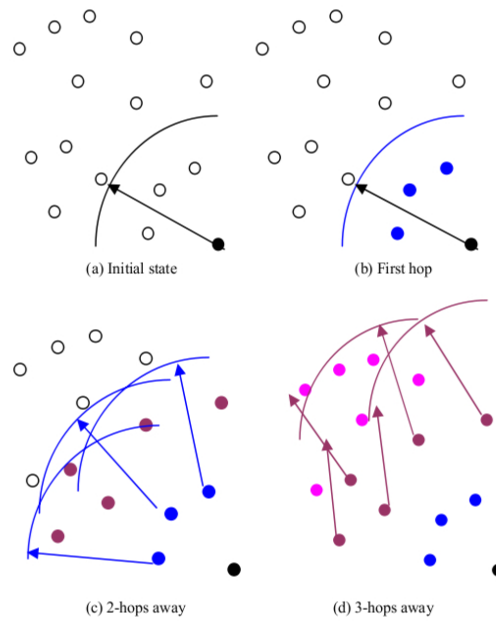


Figura 3.3: Prima fase dell'algoritmo di routing

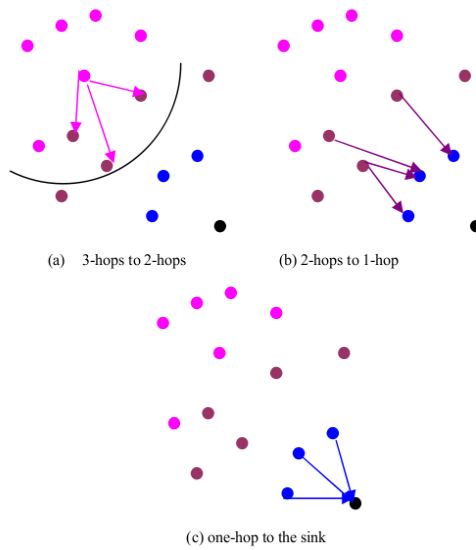


Figura 3.4: Seconda fase dell'algoritmo di routing

3.4.3 Applicazione usata negli esperimenti

Per testare un algoritmo di routing è ovviamente indispensabile analizzare del traffico generato da un'applicazione nella quale i percorsi dei pacchetti non siano solamente a singolo hop ma siano a più hop. Per i test svolti in questo documento è stata utilizzata un applicazione chiamata *Echo* nella quale tutti i nodi diversi dal sink generano una richiesta ad esso a intervalli di tempo a scelta, e il nodo sink risponde al mittente con una risposta che contiene la stessa informazione della richiesta ricevuta. Per semplificare i test si è scelto di utilizzare un solo nodo sink, ma è possibile usarne più di uno. Ad esempio in una rete molto estesa è plausibile scegliere di utilizzare più di un nodo sink, in modo da distribuire meglio e/o diminuire il traffico ed evitare che un pacchetto arrivi a destinazione dopo aver attraversato l'intera rete (caso peggiore se ci fosse solo un sink).

Vediamo ora in dettaglio com'è composto il *payload* dei pacchetti dell'applicazione.

Codice 3.5: Struttura del messaggio di Echo

```
typedef nx_struct EchoMsg {
  nx_uint8_t code;
  nx_uint32_t timestamp;
} EchoMsg;
```

Il *payload* del pacchetto è formato da due campi: il campo *code* che serve per differenziare le richieste (nodi → sink) dalle risposte (sink → nodi) e il campo *timestamp* che rappresenta l'informazione contenuta nel messaggio. In pratica ogni nodo si sveglia a intervalli periodici o casuali a seconda delle scelte, crea un messaggio con il tempo attuale di sistema, inoltra il pacchetto al sink attraverso la rete e attende la risposta da parte del sink che genera il pacchetto inserendo nel campo *timestamp* lo stesso valore ricevuto. Si vuole precisare che al posto del tempo di sistema si sarebbe potuto scegliere una qualsiasi altra informazione da trasmettere. Si tratta di un'applicazione semplice, ma sufficiente per valutare le prestazioni di un qualsiasi algoritmo. Nel Codice 3.6 si possono vedere i metodi pricipali dell'applicazione.

Codice 3.6: Echo.nc

```

void sendEcho( uint8_t code, am_addr_t dst, uint32_t timestamp ) {
  EchoMsg* echopkt =
          (EchoMsg*)(call SoftPacket.getPayload(pkt, 0));

  call SoftPacket.clear( pkt );

  echopkt->code = code;
  echopkt->timestamp = timestamp;
```

```
  pkt = call TX.send(pkt, dst, sizeof(EchoMsg), RADIO_DEFAULT_POWER, 0, 1);
  call Leds.led0Toggle();
}

event message_t* RX.receive(message_t* msg, void* payload, uint8_t len){
  if ( len == sizeof(EchoMsg) ) {
    EchoMsg* echo = (EchoMsg *)payload;
    switch (echo->code) {
      case APPECHO_REQUEST:
        sendEcho(APPECHO_REPLY, call SoftPacket.source(msg),echo->timestamp);
        break;
      case APPECHO_REPLY:
        break;
      default:
    }
  }
  return msg;
}

event void Timer.fired() {
  if (TOS_NODE_ID != SINK_ADDRESS)
          sendEcho( APPECHO_REQUEST, SINK_ADDRESS, call LocalTime.get() );

  shift = call Random.rand16()>>6;

  call Timer.startOneShot(min + shift);
}
```

La raccolta e l'elaborazione del traffico generato dall'applicazione sarà uno degli argomenti principali del capitolo seguente, nel quale si andrà a descrivere in dettaglio come verranno ricavate tutte le informazioni relative alla rete a partire dal traffico.

Capitolo 4

Routing Layer

In questo capitolo verrà spiegato in dettaglio il lavoro svolto, a partire dalla raccolta del traffico dalla rete, dalla memorizzazione di esso nella base di dati, e infine di come visualizzare tutte le informazioni tramite l'applicazione web, più in particolare attraverso il livello creato: Routing Layer.

4.1 Struttura della Base di Dati

La parte fondamentale di questo lavoro è il database al quale si interfaccia l'applicazione web; è qui che vengono salvate tutte le informazioni, relative a traffico, rotte, percorsi, Hop, perdite, ritardi, accodamenti che subiscono i pacchetti. Ogni operazione fatta dal browser in genere, si traduce in una *Remote Procedure Call* (RPC), ovvero una query fatta dal lato client al database presente nel lato server, il quale risponde con il risultato dell'interrogazione che servirà all'applicazione per mostrare graficamente le informazioni richieste. Vediamo in dettaglio la struttura del database per capire come vengono memorizzate tutte le informazioni. Figura 4.1

4.1.1 Tabelle

- **Hop** Contiene le informazioni riguardo a tutti gli Hop possibili nella rete; per Hop si intende una coppia di nodi in copertura, sorgente e destinazione, tra i quali è avvenuta almeno una trasmissione (L'Hop è unidirezionale: l'Hop che rappresenta la trasmissione da A a B è diverso dall'Hop che rappresenta la trasmissione da B ad A).

- **Path** (**Percorso**) Contiene le informazioni riguardo a tutti i percorsi (cammini), completi e incompleti, che sono stati attraversati dai pacchetti nella rete; con **percorso** si intende un insieme di Hop in sequen-

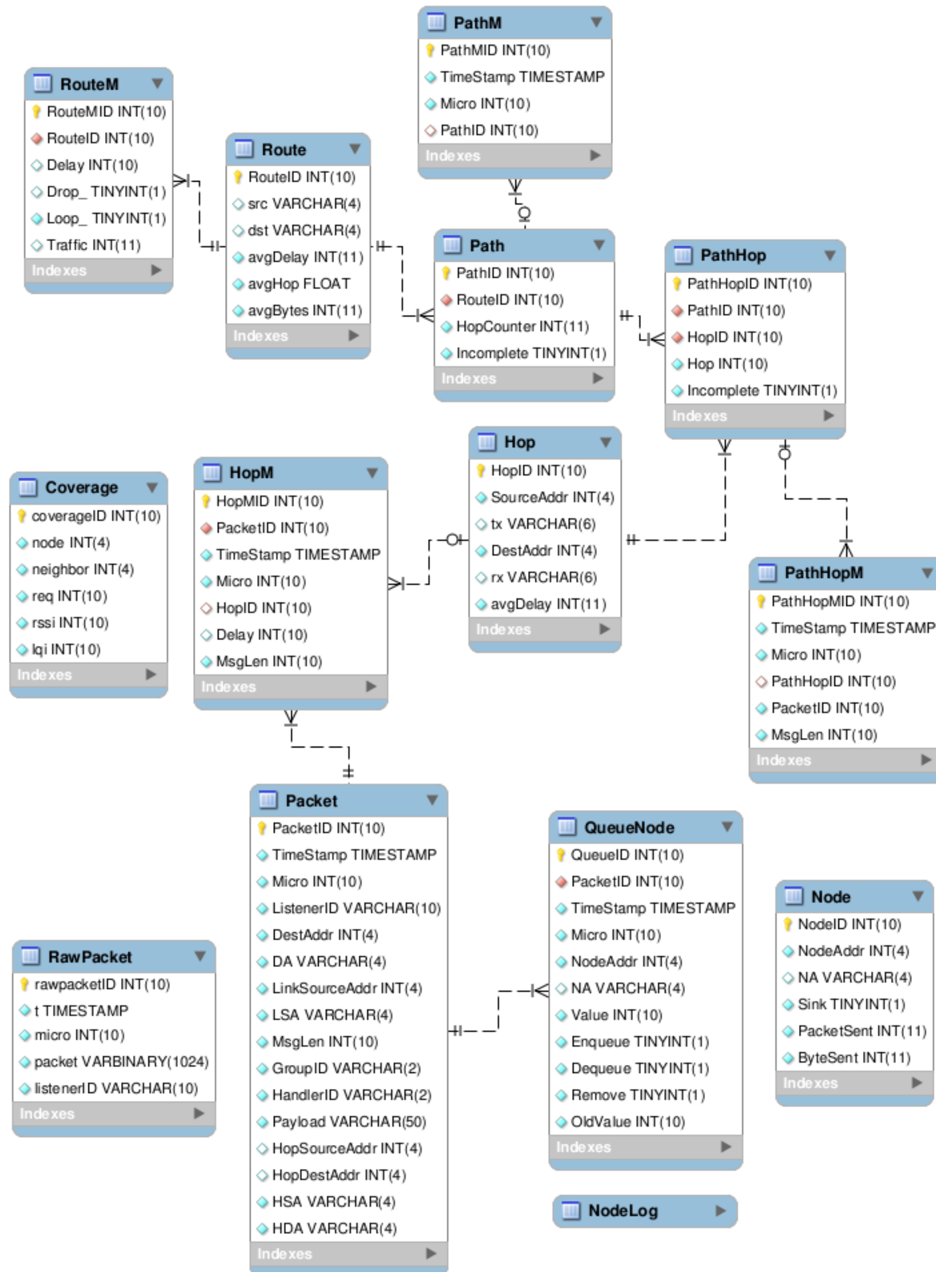


Figura 4.1: Database completo

za, cioè la destinazione dell'hop precedente coincide la sorgente dell'hop successivo, a partire da una sorgente iniziale a una destinazione finale. Con percorso completo si intende un percorso che ha raggiunto la destinazione finale, con percorso incompleto si intende un percorse che si

è interrotto prima della destinazione finale.

- **Route** (**Rotta**) Contiene le informazioni riguardo a ritardo medio, numero medio di Hop di tutte le rotte esplorate nella rete; con rotta si intende un qualsiasi percorso (cammino) tra una sorgente iniziale e una destinazione finale. Con riferimento all'applicazione usata descritta nella sezione 3.4.3, ogni rotta avrà o come sorgente o come destinazione il nodo *sink*.

- **PathHop** Contiene le informazioni riguardo alla composizione di tutti i percorsi nella rete, ovvero per ogni percorso vengono memorizzati tutti i nodi (Hop) che vengono attraversati. Ogni riga in questa tabella rappresenta un singolo Hop all'interno del singolo percorso e il campo *Hop* è un contatore incrementale che stabilisce la successione dei nodi nel percorso (Es. Hop=1 primo hop nel percoso, Hop=2 secondo hop nel percorso...).

Queste sono le tabelle fondamentali che tengono traccia di tutto il traffico in rete; è importante sottolineare che i percorsi salvati nella tabella Path sono unici, ovvero non esistono due righe distinte che rappresentano lo stesso percorso. Le informazioni riguardanti la frequenza di scelta dei percorsi verranno memorizzate nelle tabelle *Metrics* che seguono. Queste tabelle memorizzano le informazioni riguardo le singole occorrenze delle rotte, percorsi ,Hop... Ad esempio per la stessa rotta il ritardo può variare di volta in volta a seconda dello stato della rete; vengono perciò salvate tutte le singole istanze che serviranno successivamente per calcolare i valori medi.

- **RouteM** Contiene le informazioni riguardo a ritardo, traffico, presenza di loop, presenza di perdite nella singola rotta; in altre parole ogni volta che verrà iniziata la trasmissione verso una rotta verrà aggiunta una riga in tabella e le informazioni saranno relative a quella singola rotta.

- **PathM** Contiene le informazioni riguardo al singolo percorso attraversato, quali rotta associata, lunghezza in termini di numero di Hop, completo o incompleto. Verrà aggiunta una riga in tabella ogni qualvolta viene iniziato un percorso.

- **HopM** Contiene le informazioni riguardo al singolo Hop attraversato, quali ritardo e traffico transitato. Verrà aggiunta una riga in tabella ogni qualvolta viene trasmesso un pacchetto in un singolo Hop. Si avrà che il numero totale di righe in questa tabella sarà uguale al numero di pacchetti trasmessi nella rete. Servirà per tener traccia della quantità di traffico che attraversa gli Hop della rete.

- **PathHopM** Come **HopM** tiene traccia delle trasmissioni sugli Hop e in più è presente l'informazione dell'associazione tra il singolo Hop e il singolo percorso.

- **Packet** Contiene tutto il traffico che viaggia in rete nella durata dell'esperimento. Ogni riga rappresenta un pacchetto che è stato trasmesso nella rete. Come si vede in Figura 4.1, i campi di questa tabella rappresentano la struttura del messaggio già descritta in Sezione 3.4.1.

- **QueueNode** Contiene le informazioni relative all'andamento delle code di ricezione (dimensione della coda) di tutti i nodi nella rete. Verrà aggiunta una riga in tabella ogni qualvolta si verifica un evento di accodamento, di rimozione, o di perdita di un pacchetto.

Infine sono presenti delle tabelle che non fanno parte del modello relazionale, in quanto sono indipendenti dalle altre.

- **Node** Contiene le informazioni relative ai nodi utilizzati nella simulazione.

- **Coverage** Contiene le informazioni relative agli esperimenti di copertura, cioè quali sono i nodi "vicini" con i quali un nodo può comunicare. Inoltre contiene informazioni riguardo la qualità dei collegamenti in termini di Received Signal Strength Indication (RSSI) e Link Quality Indicator (LQI).

- **RawPacket** Contiene tutti pacchetti che viaggiano in rete nella durata dell'esperimento. Questo è il punto di partenza di tutta la base di dati. I nodi che ascoltano il traffico infatti inseriscono in tempo reale i pacchetti ricevuti all'interno di tabella. Memorizza il tempo di arrivo con una precisione del microsecondo e salva il pacchetto in formato binario.

4.1.2 Modello Relazionale o *Entity-Relationhship*

In Figura 4.2 si può vedere il modello *Entity-Relationship* (E-R) della base di dati.

Relazioni (1 - ∞)

- Route-Path: Ogni rotta, sorgente destinazione, può avere più percorsi.

- Path-PathHop: Ogni percorso è formato da più hop.

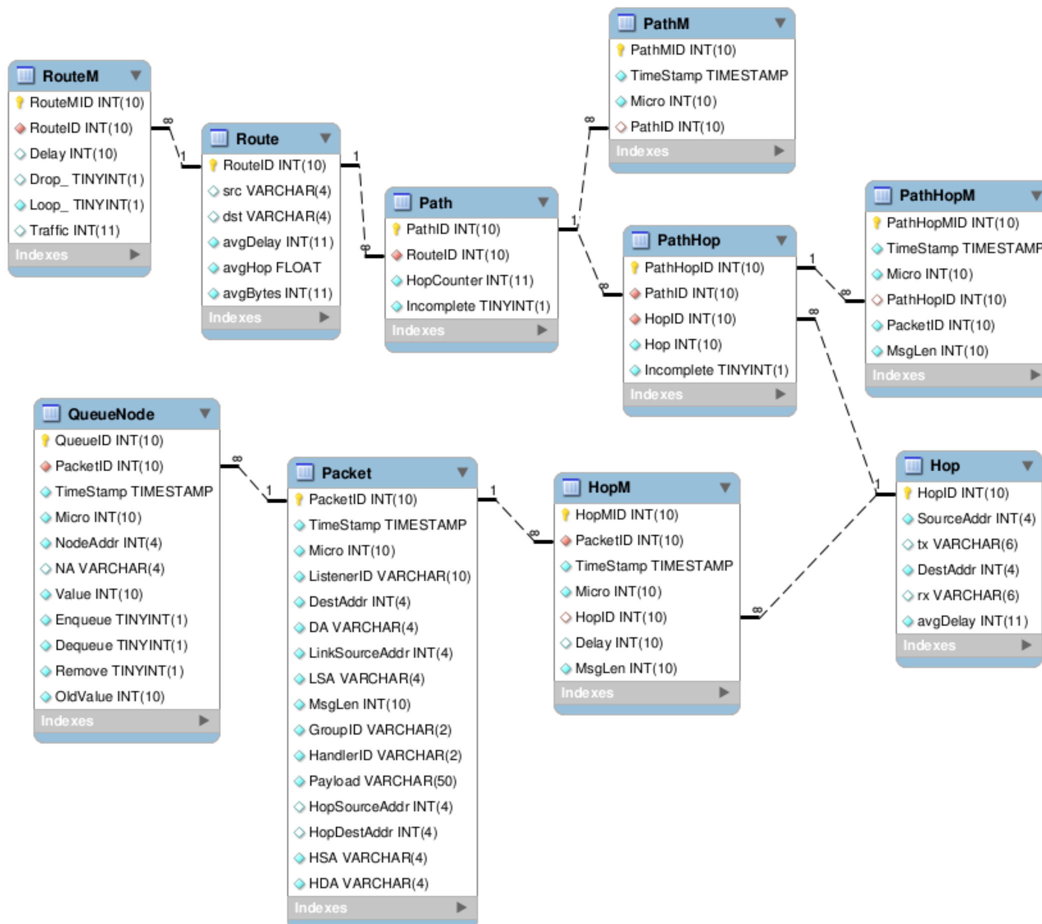


Figura 4.2: Modello E-R del database

- Hop-PathHop: Ogni hop può far parte di più percorsi.
- Route-RouteM: Ogni rotta può essere utilizzata più volte.
- Path-PathM: Ogni percorso può essere utilizzato più volte.
- Hop-HopM: Ogni hop può essere utilizzato più volte.
- PathHop-PathHopM: Ogni hop in un percorso può essere utilizzato più volte.

Relazioni (1 - 1) Queste relazioni non sono state create nel database ma vengono utilizzate al momento di effettuare delle query; il motivo per cui non sono state create le associazioni è per tenere separati diversi tipi di

informazioni in tabelle diverse. Le coppie di tabelle conterranno lo stesso numero di righe; l'inserimento in queste tabelle avviene in momenti diversi ma vengono inserite informazioni riguardanti lo stesso pacchetto.

- RouteM.RouteMID-PathM.PathMID: Se si guarda la rete da un punto di vista end-to-end, l'inizio di una trasmissione in una certa rotta dovrà essere associato alla trasmissione lungo un percorso.
- HopM.HopMID-PathHopM.PathHopMID: Anche in questo caso rappresentano la stessa informazione, cioè la trasmissione sul generico Hop.
- Packet.PacketID-HopM.PacketID: La trasmissione sul generico Hop è sempre associata alla trasmissione di un pacchetto che è transitato nella rete.
- Packet.PacketID-QueueNode.PacketID: Gli eventi di accodamento e rimozione dalla coda di un nodo, sono associati all'arrivo o alla trasmissione di un pacchetto che è transitato per quel nodo.

4.2 Popolazione della Base di Dati

In questa Sezione verrà descritto esaustivamente il processo che elabora i dati ricavati dal traffico e li mette a disposizione dell'applicazione web per poterli visualizzare graficamente.
Il processo si può riassumere in 3 fasi principali:

1. **Raccolta del traffico**: il traffico verrà raccolto da nodi dedicati (*sniffer*), che memorizzeranno i pacchetti ascoltati in una tabella temporanea del database chiamata *RawPacket*.
2. **Processing**: I dati presenti nella tabella *RawPacket* vengono elaborati e memorizzati nella tabella *Packet*.
3. **Populate**: I dati presenti nella tabella Packet, che rappresentano il traffico, vengono elaborati per completare il riempimento del database, mettendo così a disposizione i dati per la visualizzazione grafica dell'applicazione web.

4.2.1 Raccolta del traffico

La raccolta del traffico è ovviamente la prima fase con cui si ha a che fare se si vuole analizzare le prestazioni di un algoritmo di routing. E' una fase

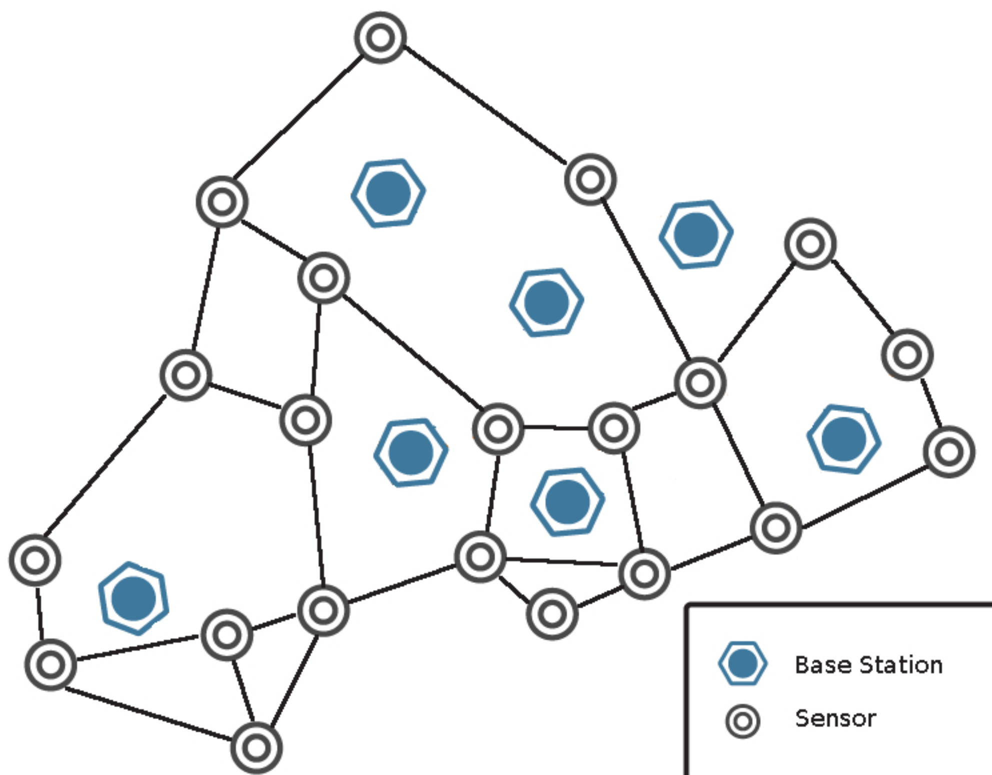


Figura 4.3: Raccolta del traffico

piuttosto delicata in quando è necessario assicurarsi che ogni pacchetto che viaggia nell'aria venga catturato per non compromettere già in partenza i dati ad analizzare. Questa operazione viene effettuata disseminando nella rete in esame dei nodi speciali che non andranno a far parte della rete coinvolta nella trasmissione delle informazioni. Come si vede in Figura 4.3 in questi nodi viene caricata un'applicazione chiamata BaseStation (BS)[6] che non è nient'altro che un'applicazione che agisce come ponte tra la porta seriale e la rete radio: quando riceve un pacchetto dalla porta seriale lo trasmette alla radio, quando riceve un pacchetto dalla radio lo trasmette alla porta seriale. Ovviamente sarà questa seconda funzionalità quella interessata per l'ascolto del traffico. La disseminazione delle BS viene effettuata in modo che ogni sensore abbia almeno una Base Station in copertura. Grazie all'infrastruttura del testbed, come descritto in Sezione 1.2, è possibile raccogliere tramite USB, tutti i pacchetti ascoltati dalle BS contemporaneamente e in parallelo. Sarà compito di un qualche processo catturare tutti i dati in arrivo sulla porta USB per poi trasferirli nel database.

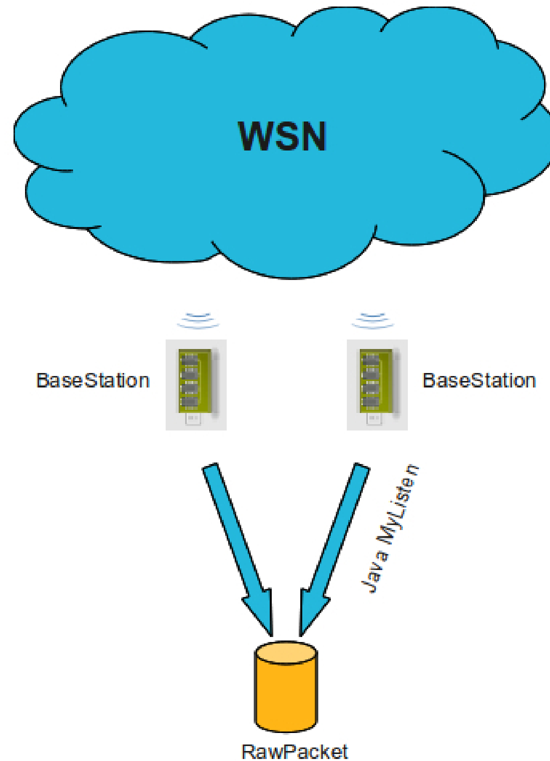


Figura 4.4: Raccolta del traffico

Dalla Figura 4.4 si può infatti notare che per ogni BS vi è un processo java chiamato *MyListen* in ascolto sulla porta seriale che gestisce la comunicazione con database. Esso infatti inserisce istantaneamente nella tabella *RawPacket* i pacchetti ricevuti. La comunicazione tra il sensore e il processo avviene tramite la creazione di un socket come descritto in Sezione 1.2.
Ricordiamo i campi che compongono la tabella *RawPacket*:

- rawpacketID: identificativo del pacchetto
- timestamp: tempo di arrivo *(DD:MM:YY HH:MM:SS)*
- micro: microsecondi
- packet: il pacchetto ricevuto in formato binario
- listenerID: identificativo della BS che ha sentito il pacchetto e che ha effettuato l'inserimento.

Il cuore di questo processo si può vedere nel Codice 4.1 dove vengono mostrate le operazioni effettuate a partire dall'arrivo del pacchetto. Si nota

che dopo l'arrivo del pacchetto viene effettuata una query di controllo per verificare se qualche altro processo ha già effettuato l'inserimento; se il pacchetto è presente nella tabella non è necessario inserirlo in caso contrario lo inserisce.

Uno dei compiti fondamentali di questo processo è che gestisce gli inserimenti come transazioni, descritte esaustivamente in Sezione 2.3.2. Questo meccanismo è importantissimo in questo tipo di situazioni e serve per evitare che si verifichino eventi di inserimento duplicati; è facile intuire che se un nodo si trova tra due BS e trasmette un pacchetto, questo venga ricevuto da entrambe. Ora, se non si fosse effettuato il controllo, si verificherebbe che entrambi i processi associati alle due BS inserirebbero lo stesso pacchetto all'interno della tabella, generando così un'informazione errata. Si potrebbe pensare che il nodo ha trasmesso due volte lo stesso pacchetto, quando in realtà è avvenuta solo una trasmissione.
Questo controllo purtroppo non è sufficiente poiché non riesce a gestire l'accesso concorrente al database da parte di più processi. É qui che entrano in gioco le transazioni. Se non venissero utilizzate potrebbe verificarsi l'evento che la query di controllo di un processo A abbia dato esito negativo, cioè non ha trovato il pacchetto, e tra l'istante di questa query e l'instante di inserimento sempre del processo A, avvenga l'inserimento dello stesso pacchetto da parte di un processo B (dati fantasma). Otterremmo anche in questo caso un doppio inserimento.
Per evitare quindi queste situazioni si è scelto il livello di isolamento *Serializable*, il più restrittivo possibile, in modo da bloccare le righe risultanti dalla query di controllo da qualsiasi altro processo.

Codice 4.1: MyListen.java

```
// ——— ARRIVO DEL PACCHETTO ———

byte[] packet = reader.readPacket();

...

  varbinary = HexToBin(hex);

  // ——— INIZIO TRANSAZIONE ———

  // Autocommit disabilitato per le transazioni
  conn.setAutoCommit(false);
  conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
  ...
  for (int i=0;i<N_TRY;i++){

    // —— CONTROLLO SE PKT GIA' INSERITO ——

    SQL = "SELECT * FROM RawPacket WHERE
```

```
    (ABS(TIMESTAMPDIFF(FRAC_SECOND, TIMESTAMPADD(FRAC_SECOND, micro, t),
    TIMESTAMPADD(FRAC_SECOND,"+micro+",NOW()))) < " + AIR_MAX_DELAY + ")
    AND packet = '"+ varbinary +"'";

    Statement stmt = conn.createStatement();
    ResultSet result = stmt.executeQuery(SQL);
    if (result.last() != false){
          bError = true;
          stmt.close();
          break;
    }
    stmt.close();

    // ----- INSERIMENTO IN TABELLA -----

    SQL = "INSERT INTO RawPacket (t,micro,packet,listenerID)
      VALUES (NOW(),'"+micro+"','"+varbinary+"','"+id+"')";

    ...

  }
  if( bError ){
    conn.rollback();
  }
  else{

    // -- COMMIT ---

    conn.commit();
  }

  // ------ FINE TRANSAZIONE ----------
```

4.2.2 Processing

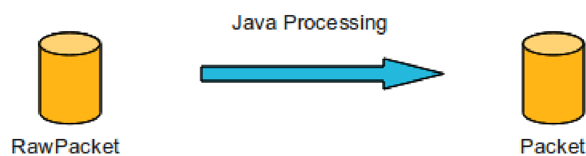


Figura 4.5: Processing

Questa seconda fase, non avviene in tempo reale come la fase precedente; infatti solo nella tabella RawPacket i dati fluiscono istantaneamente, mentre nelle restanti tabelle della base di dati il ritardo di inserimento può essere scelto a piacere per impostare il tempo di risposta della visualizzazione grafica. Tutti i risultati visibili dall'applicazione web infatti vengono raccolti dalle tabelle che andranno popolate dal processo descritto in questa Sezione e dal processo descritto nella Sezione successiva 4.3.6. Questo aspetto verrà chiarito in seguito quando si andrà a descrivere l'utilizzo del Routing Layer.

Il compito del processo in questione è quello di trasferire i dati dalla tabella *RawPacket* alla tabella *Packet*, effettuando una conversione da binario ad esadecimale. Dopo la conversione il processo andrà ad individuare i campi del pacchetto descritti in Sezione 3.4.1 per inserirli correttamente nella tabella *Packet*. Ricordiamo i campi che compongono la tabella *Packet*:

- PacketID: identificativo del pacchetto
- Timestamp: tempo di arrivo *(DD:MM:YY HH:MM:SS)*
- Micro: microsecondi
- ListenerID: BS che ha ascoltato il pacchetto
- DestAddr: destinazione finale
- LinkSourceAddr: sorgente iniziale
- MsgLen: lunghezza in bytes del messaggio
- GroupID: gruppo
- HandlerID: tipo di messaggio
- Payload: informazione
- HopSourceAddr: sorgente del singolo Hop
- HopDestAddr: destinazione del singolo Hop

Inoltre in successione ai campi che rappresentano indirizzi dei nodi (DestAddr,LinkSourceAddr,...) vi sono sempre i campi testuali che contengono i *nick* associati ad essi (es. il nodo *00 28* ha come *nick* "p8").
Nel riquadro 4.2 si può vedere un esempio del traffico catturato da una BS in formato esadecimale.

Codice 4.2: Esempio di traffico in formato esadecimale

```
00 00 28 80 0B 0A 00 65 00 0B 00 38 02 01 00 34 00 28
00 00 28 80 47 09 00 80 AA 00 1B 07 45 00 47 00 34
00 00 28 80 11 09 00 80 AA 00 1B 2E 30 00 22 00 27
00 FF FF 00 11 01 00 64 02
```

Ignorando il primo byte (00) che indica che questo pacchetto è un pacchetto di tipo Active Message (AM) [6], ricordiamo i campi che compongono la struttura del messaggio:

- Destination address (2 bytes)

- Link source address (2 bytes)
- Message length (1 byte)
- Group ID (1 byte)
- Active Message handler type (1 byte)
- Payload (fino a 28 bytes)
- Routing Tag (4 bytes)
  - Hop Source Address (2 bytes)
  - Hop Destination Address (2 bytes)

Quindi possiamo interpretare i pacchetti nel riquadro 4.2 nel modo seguente:

dest addr	link source addr	msg len	groupID	handler ID	payload	hop source addr	hop dest addr
00 28	80 0B	0A	00	65	00 0B 00 38 02 01	00 34	00 28
00 28	80 47	09	00	80	AA 00 1B 07 45	00 47	00 34
00 28	80 11	09	00	80	AA 00 1B 2E 30	00 22	00 27
FF FF	00 11	01	00	64	02		

Figura 4.6: Campi del pacchetto

Codice 4.3: Processing.java

```
1   try {
2  conn = DriverManager.getConnection("jdbc:mysql://10.1.128.1/bazzacol?
3  user=bazzacol&password=******");
4  Initialize_TosNodeIDMap(conn);
5
6  // ----- SELEZIONA I RECORD DALLA TABELLA RAWPACKET
7  if (id==-1) {
8    // prima chiamata a ProcEcho
9        SQL = "SELECT * FROM RawPacket";
10 } else {
11   SQL = "SELECT * FROM RawPacket WHERE rawpacketID > " + id;
12 }
13 Statement stmt = conn.createStatement();
14 ResultSet rs = stmt.executeQuery(SQL);
15 int count=0;
16 int pktid = 0;
17
18 while (rs.next()){
19   pktid = rs.getInt("rawpacketID");
20   String t = rs.getString("t");
21   String micro = rs.getString("micro");
22   String listenerID = rs.getString("listenerID");
23   String Packet = rs.getString("packet");
24   // ----- CONVERSIONE BIN TO HEX
25   String PacketHex = BinToHex(Packet);
26   // ----- INIZIO CATTURA DEI CAMPI -----
27   int DestAddr = Integer.parseInt(PacketHex.substring(L_PREAMB*2 +1,
```

```
28   L_DA*2+L_PREAMB*2),16);
29   String DA = (!TosNodeIDMap.containsKey(DestAddr)) ? "BCST" :
30   TosNodeIDMap.get(DestAddr);
31   int LinkSourceAddr = Integer.parseInt(PacketHex.substring(L_DA*2+
32   L_PREAMB*2 +1, (L_DA + L_LSA)*2+L_PREAMB*2),16);
33   String LSA = TosNodeIDMap.get(LinkSourceAddr);
34   String temp = PacketHex.substring((L_DA + L_LSA)*2+L_PREAMB*2, (L_DA
35   + L_LSA + L_ML)*2+L_PREAMB*2);
36   int MsgLen = Integer.parseInt(temp,16);
37   String GroupID = PacketHex.substring((L_DA + L_LSA + L_ML)*2+L_PREAMB*2,
38   (L_DA + L_LSA + L_ML + L_GID)*2+L_PREAMB*2);
39   String HandlerID = PacketHex.substring((L_DA + L_LSA + L_ML + L_GID)*2
40   +L_PREAMB*2, (L_DA + L_LSA + L_ML + L_GID + L_HID)*2+L_PREAMB*2);
41   String SourceAddr = "";
42   int HopSourceAddr = 0;
43   int HopDestAddr = 0;
44   String Payload = "";
45   String HSA = "";
46   String HDA = "";
47 if (fixed_L_PAYLOAD) {
48   if ((MsgLen == msglength)){
49           Payload = PacketHex.substring((L_DA + L_LSA + L_ML + L_GID +
50           L_HID)*2+L_PREAMB*2,(L_DA + L_LSA + L_ML + L_GID + L_HID +
51           L_PREAMB+L_PAYLOAD)*2);
52           HopSourceAddr = Integer.parseInt(PacketHex.substring(((L_DA +
53           L_LSA + L_ML + L_GID + L_HID + L_PREAMB+L_PAYLOAD)*2), (L_DA +
54           L_LSA + L_ML + L_GID + L_HID + L_PREAMB + L_PAYLOAD + L_HSA)*2),16);
55           HopDestAddr = Integer.parseInt(PacketHex.substring((L_DA + L_LSA
56           + L_ML + L_GID + L_HID + L_PREAMB + L_PAYLOAD + L_HSA)*2, (L_DA
57           + L_LSA + L_ML + L_GID + L_HID + L_PREAMB + L_PAYLOAD + L_HSA
58           + L_HDA)*2),16);
59           HSA = TosNodeIDMap.get(HopSourceAddr);
60           HDA = TosNodeIDMap.get(HopDestAddr);
61   } else {
62           Payload = PacketHex.substring((L_DA + L_LSA + L_ML + L_GID
63           + L_HID)*2+L_PREAMB*2);
64   }
65 } else {
66         if (MsgLen==1 || MsgLen==5) {
67                 // no Routing Tag
68                 Payload = PacketHex.substring((L_DA + L_LSA + L_ML +
69                 L_GID + L_HID)*2+L_PREAMB*2);
70                 HopSourceAddr = 0;
71                 HopDestAddr = 0;
72                 HSA = "";
73                 HDA = "";
74         } else {
75                 int ind = PacketHex.length()-(L_HDA*2);
76                   HopDestAddr = Integer.parseInt(PacketHex.substring(ind,
77                   PacketHex.length()),16);
78                   int ind2 = ind-(L_HSA*2);
79                   HopSourceAddr = Integer.parseInt(PacketHex.substring(ind2,
80                   ind),16);
81                   HSA = TosNodeIDMap.get(HopSourceAddr);
82                   HDA = TosNodeIDMap.get(HopDestAddr);
83                   Payload = PacketHex.substring((L_DA + L_LSA + L_ML + L_GID
84                   + L_HID)*2+L_PREAMB*2,ind2);
85         }
86
87 }
88   // ---- INSERIMENTO NELLA TABELLA Packet -----
89   if (!((HSA==null) || (HDA==null))) {
```

```
90            SQL = "INSERT INTO Packet (PacketID,TimeStamp,Micro,ListenerID,
91            DestAddr,DA,LinkSourceAddr,LSA,MsgLen,GroupID,HandlerID,SourceAddr,
92            Payload,HopSourceAddr,HopDestAddr,HSA,HDA) VALUES ('"+pktid+"',
93            '"+t+"','"+micro+"','"+listenerID+"','"+DestAddr+"','"+DA+"',
94            '"+LinkSourceAddr+"','"+LSA+"','"+MsgLen+"','"+GroupID+"',
95            '"+HandlerID+"','"+SourceAddr+"','"+Payload+"','"+HopSourceAddr+"',
96            '"+HopDestAddr+"','"+HSA+"','"+HDA+"')";
97            Statement stmt2 = conn.createStatement();
98            stmt2.executeUpdate(SQL);
99            count++;
100           stmt2.close();
101   }
```

Dal Codice 4.3 si può notare quali sono le operazione eseguite da questo processo. In successione si possono riassumere nel modo seguente:

1. Acquisisce i dati da una query di selezione sulla tabella *RawPacket* (Linea 6).

2. Converte il pacchetto da binario ad esadecimale (Linea 24).

3. Conoscendo la lunghezza in bytes dei campi del messaggio, estrae attraverso la funzione *substring* della classe *String* il valore dei campi (Linea 26).

4. Tutti i valori dei campi sono a disposizione, quindi effettua l'inserimento nella tabella *Packet* (Linea 88).

Come si vede in Figura 4.6 il tipo di messaggio con il campo *handlerID*=64 è sprovvisto dei campi che compongono il *RoutingTag*. Questo perché la trasmissione di questo tipo di messaggio non avviene a singolo Hop, ovvero tra una coppia di nodi, ma avviene in broadcast. Il processo *Processing* necessita quindi di differenziare il tipo di messaggio che sta elaborando, per evitare errori nella cattura dei valori dei campi. La finestratura con la funzione *substring* avviene in modo semplice fino al campo *handlerID*. Successivamente se il pacchetto non possiede il *RoutingTag* i restanti bytes fanno tutti parte del *payload* del messaggio. Se il pacchetto possiede il *RoutingTag*, si andrà a catturare i campi *HopSourceAddr* e *HopDestAddr*, partendo dalla fine della stringa. I bytes restanti formeranno il *payload*.

4.2.3 Populate

In successione al processo *Processing*, descritto nella Sezione precedente, viene eseguito un altro processo java chiamato *Populate*. Come si vede in Figura 4.7 esso si incarica di riempire tutta la base di dati descritta in precedenza; si incarica quindi di eseguire tutte le elaborazioni sul traffico necessarie per trovare le rotte, i percorsi, i ritardi, le perdite, gli accodamenti.

Visto il numero elevato di operazioni che effettua questo processo verranno messe in luce solo le parti più significative del codice che eseguono le operazioni più complesse. Per capire come avviene la successione delle operazioni di questo processo è utile dare uno sguardo al metodo *main* che raggruppa in modo sequenziale le operazioni.

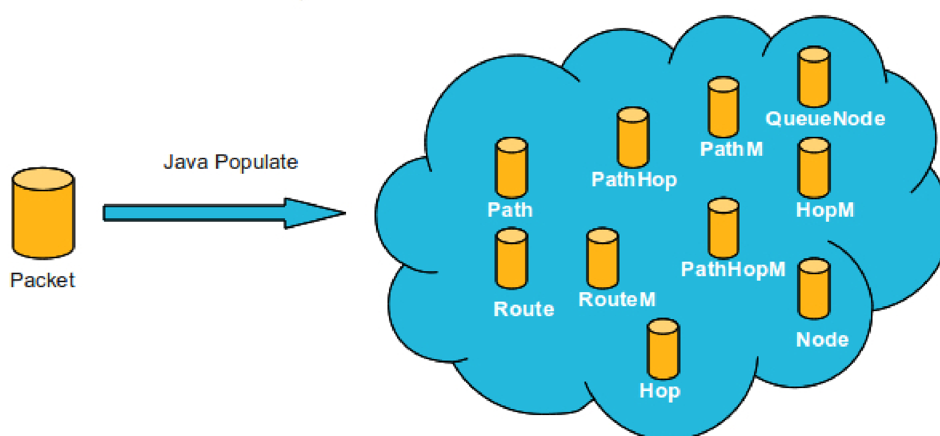


Figura 4.7: Populate

Codice 4.4: Metodi principali del processo Populate.java

```
1
2  FindRoutes(conn);
3  FindHops(conn);
4  FindPaths(conn);
5  FindMetrics(conn);
6      (all'interno del metodo FindMetrics)
7      - FindDelay(conn);
8      - AvgDelay(conn);
9      - FindHop(conn);
10     - AvgHop(conn);
11     - AvgTraffic(conn);
12 FindNodes(conn);
13 FindQueue(conn);
```

I primi due metodi, *FindRoutes* e *FindHops*, scandiscono tutte le righe della tabella *Packet* per trovare tutte le possibili rotte e i possibili Hop dove sono avvenute trasmissioni.
Il terzo metodo *FindPaths* si incarica di trovare, per ogni rotta, tutti i percorsi che hanno preso i pacchetti. Si ricorda che una rotta è un qualsiasi percorso (cammino), tra una sorgente e una destinazione e un percorso è un insieme qualsiasi di Hop in successione tra una sorgente e una destinazione. Questo metodo è molto importante e le parti più importanti del suo codice meritano di essere analizzate. Il Codice 4.5 si riferisce alle operazioni effettuate su una singola rotta.

Codice 4.5: Metodo FindPath

```
...

private final static int MAX_PAYLOAD_TIME = 15;

...

ArrayList<MyPayload> pld = Findpayloads(conn,r);

...

for (int i=0;i<pld.size();i++){
    String payload = (pld.get(i)).getP();
    Statement stmt = conn.createStatement();
    SQL = "SELECT  ....  AND TimeStamp >= \""+ (pld.get(i)).getTs() +"\"
    "AND TimeStamp <  TIMESTAMPADD(SECOND,"+MAX_PAYLOAD_TIME+",\""
    +(pld.get(i)).getTs()+"\") ORDER BY TimeStamp, Micro" :

...

    boolean isPath = isPath(result);

    if (!existPath(conn,path_hop_id,r)) {

    // aggiungi una riga a path

    if (isPath)
            SQL = "INSERT INTO Path (RouteID) ...
    } else {
            SQL = "INSERT INTO Path (RouteID,Incomplete) ...
    }

    // aggiungi righe a Path_Hop
    for (int j=1;j<=path_hop_id.size();j++) {
      if (isPath) {
        SQL = "INSERT INTO `PathHop` (PathID,HopID,Hop) ..
      } else {
        if (j==path_hop_id.size()) {
          SQL = "INSERT INTO `PathHop` (PathID,HopID,Hop,Incomplete) ....
            } else {
          SQL = "INSERT INTO `PathHop` (PathID,HopID,Hop) ...
            }
      }
    }
    // aggiungi una riga a PathM ogni volta che percorre un path
    // serve per tener conto dei del numero di volte che viene effettuato
    // un percorso

    // il metodo existPath con il flag boolean serve per cercare il pathid
    // a partire dagli hop
    int path_id = existPath(conn,path_hop_id,r,true);

    SQL = "INSERT INTO PathM (PathID,TimeStamp,Micro) ...

    // aggiungi righe a PathHopM ogni volta che percorre un path
    // serve per tener conto dei del numero di volte che viene percorso
    // un hop nello specifico path

    // aggiungi righe a Path_Hop_M
    for (int j=1;j<=path_hop_id.size();j++) {
    SQL = "INSERT INTO PathHopM (TimeStamp,Micro,PathHopID,PacketID,MsgLen)..
```

```
61      }
62
63
64  ...
```

Il primo metodo, chiamato *FindMyPayloads*, serve per identificare le singole richieste effettuate da un determinato nodo, che quindi corrisponde alla sorgente iniziale della rotta in esame. Come già descritto in Sezione 3.4.3, l'applicazione usata per i test trasmette una richiesta al nodo sink, il quale risponde con la stessa informazione contenuta nel *payload* del messaggio ricevuto. L'identificazione della richiesta avviene appunto attraverso il *payload* del messaggio. Fissando il campo *payload*, la sorgente iniziale, la destinazione finale, è possibile tramite una query sulla tabella *Packet*, tracciare il percorso della richiesta o risposta attraverso i vari Hop. La struttura dati che memorizza univocamente la richiesta o la risposta è chiamata *MyPayload*. Questa struttura contiene anche un campo *TimeStamp*, che indica l'istante nel quale è avvenuta la prima trasmissione; è necessario per poter identificare diverse richieste o risposte che contengono lo stesso *payload*. Infatti la query di selezione sul traffico che "finestra" il percorso fatto dall'i-esima richiesta, non solo impone il vincolo sul *payload* ma impone dei vincoli anche sulla finestra temporale in cui può trovarsi quel *payload*. Attraverso la costante *MAX_PAYLOAD_TIME* è possibile decidere per quanti secondi dopo la prima trasmissione si debba considerare quel payload facente parte sempre della stessa richiesta o risposta. Allo scadere di questo timeout un eventuale pacchetto contenente lo stesso payload, verrà considerato come una diversa richiesta o risposta. E' necessario avere un parametro di questo tipo per poter adattare il processo in funzione della topologia della rete che si vuole monitorare; ovviamente una rete con rotte che presentano molti Hop necessita di più tempo per completare una trasmissione end-to-end.
Una volta ottenute le righe dalla tabella Packet che compongono un percorso, viene controllato se il percorso in questione è completo o no e se è già presente un percorso identico o meno nella base di dati. Per percorso completo si intende un percorso che ha raggiunto la destinazione, in altre parole se l'ultimo pacchetto del percorso ha come valore del campo *Hop Destination Address* lo stesso valore del campo *Destination Address* allora si considera completo. Per verificare invece se è già stato inserito in precedenza quel percorso, si vanno a confrontare tutti i percorsi già inseriti nella tabella *PathHop*. Attraverso il campo *HopID* che identifica un Hop, è sufficiente verificare se è già presente una serie di HopID in successione identica a quella in esame.

L'operazione si conclude effettuando gli inserimenti nelle tabelle *Path*, *PathHop*,*PathM*,*PathHopM* abilitando o meno i vari flag a seconda se il per-

corso è completo o incompleto. Per chiarire ancora la differenza tra la tabella *Path* e *PathM* si ha che, per ogni percorso in esame, verrà sempre effettuato un inserimento nella tabella *PathM* invece solo in presenza di un nuovo percorso si effettuerà un inserimento nella tabella *Path*. Si ricorda che la memorizzazione completa di un percorso avviene nella tabella *PathHop* precisamente si andrà ad inserire una riga per ogni hop presente in quel percorso.

Il metodo *FindMetrics* calcola i ritardi sugli Hop e i ritardi end-to-end, i ritardi medi, il numero di Hop end-to-end dei percorsi e il numero medio di Hop delle rotte, il traffico sugli Hop e sulle rotte end-to-end.
Il ritardo di un Hop che va dal nodo A al nodo B viene calcolato facendo la differenza tra il tempo relativo alla trasmissione da A a B e tra il tempo relativo alla ricezione del pacchetto da parte di A. Ad esempio, se il percorso in esame è formato da tre Hop, e quindi la query di selezione (dalla tabella Packet che contiene anche il campo *timestamp*) è formata da tre righe (3 trasmissioni, 4 nodi coinvolti), il ritardo del terzo Hop sarà dato dalla differenza tra il terzo e il secondo *timestamp*, mentre il ritardo del secondo Hop dalla differenza tra il secondo e il primo *timestamp*. Il primo Hop ha sempre ritardo nullo.
Il ritardo end-to-end della singola istanza della rotta viene calcolato facendo la somma dei ritardi dei singoli Hop che compongono il percorso, mentre il ritardo end-to-end medio viene calcolato mediando su tutti le istanze presenti escludendo i percorsi incompleti e i percorsi completi che presentano dei *loop* ritenendo che questi siano casi particolari verificatesi in situazioni particolari. Per *loop* si intende una situazione nella quale un pacchetto viene “rimbalzato“ tra due Hop per almeno una volta, cioè se un nodo A trasmette ad un nodo B e successivamente B reinoltra il pacchetto ad A.
Il numero medio di Hop di una rotta viene calcolato prendendo l’intero più grande rispetto alla media aritmetica dei valori ottenuti nei singoli percorsi. Si è scelta questa soluzione in modo che le rotte con numero medio di Hop pari a 1, siano interessate sempre e solo da percorsi formati da un singolo Hop. Così facendo per la definizione di ritardo a un Hop data sopra, queste rotte avranno tutte un ritardo medio nullo.
Il traffico sugli Hop ed end-to-end viene calcolato sommando il campo *MsgLen* per ogni pacchetto in transito.
Il metodo *FindNodes* serve solamente per trovare quali sono i nodi interessati nell’esperimento, informazione che serve principalmente all’applicazione grafica per una corretta visualizzazione.
Infine il metodo *FindQueue* calcola gli accodamenti in ricezione per tutti i nodi; vediamo brevemente anche in questo caso il suo codice per capirne il funzionamento.

Gli eventi che si possono catturare dal traffico sono essenzialmente la ricezione e la trasmissione di un pacchetto; inoltre grazie al Routing Tag è possibile sapere se il pacchetto è solo transitato per quel determinato nodo (inoltro) o se la destinazione finale era precisamente quel nodo.

Volendo calcolare l'accodamento per un nodo X, gli eventi possibili sono:

1. ricezione di un pacchetto con destinazione finale il nodo X (*DestAddr* = X).

2. ricezione di un pacchetto con destinazione finale non il nodo il X (*DestAddr* != X, *HopDestAddr* = X).

3. trasmissione di un pacchetto con sorgente iniziale non il nodo X (*LinkSourceAddr* != X, *HopSourceAddr* = X).

Dalla Figura 4.8 si può ottenere un chiarimento riguardo agli eventi appena descritti. Nella figura il primo evento si ottiene quando il nodo sink riceve il pacchetto, il secondo evento si ottiene quando uno dei nodi intermedi (B,C,D) riceve il pacchetto da inoltrare, il terzo evento si ottiene quando uno dei nodi intermedi ritrasmette (inoltra) il pacchetto.

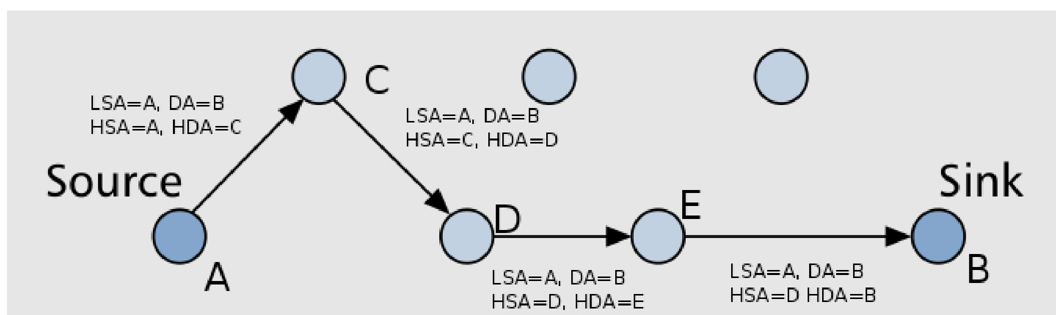


Figura 4.8: Variazione dei campi LSA,DA,HSA,HDA in una trasmissione multi hop

Non viene tenuto conto delle trasmissioni che hanno come sorgente iniziale il nodo X (*LinkSourceAddr* = X) perché questi eventi sono indipendenti e generati internamente dall'applicazione e non è possibile dal traffico correlarli con l'accodamento. Come si vede dal Codice 4.6, per ogni nodo viene fatta una query di selezione in modo da ottenere tutto il traffico transitato per esso e il calcolo della dimensione della coda viene fatto tramite una struttura dati ad *array*. Ad un evento di ricezione corrisponde un inserimento in coda, ad un evento di trasmissione una rimozione dalla coda; ad ogni iterazione l'evento avvenuto e la corrispondente dimensione della coda vengono riportati in una

riga nella tabella *QueueNode*. Attraverso i campi *Enqueue*,*Dequeue*,*Remove* è possibile sapere l'evento corrispondente alla riga della tabella. Questi sono infatti dei campi booleani, nei quali l'abilitazione di uno di essi, porta ad avere tutti gli altri disabilitati. Più in dettaglio vediamo dal Codice 4.6 che se il pacchetto ricevuto è per il nodo X (*DestAddr* = X) vi è l'inserimento e l'immediata rimozione, stando ad indicare che il pacchetto è sì entrato in coda ma l'applicazione avendo riconosciuto la proprietà del pacchetto lo ha processato immediatamente. Se il pacchetto ricevuto non è per il nodo X (*DestAddr* != X, *HopDestAddr* = X) avverrà un normale inserimento in coda; il pacchetto appena inserito resterà in coda finché non verrà ritrasmesso ad un next hop. A quest'ultimo evento è associata la rimozione dalla coda e mediante il campo *Old Value* è possibile anche tener traccia della posizione in coda al momento dell'ingresso. Infatti prima di aggiungere la riga corrispondente alla rimozione dalla coda, viene fatta una query di selezione per ricavare il valore della coda al momento della ricezione del pacchetto.
Infine, ad ogni iterazione, viene effettuato un controllo per verificare se sono presenti in coda pacchetti "scaduti", che probabilmente sono stati persi e che quindi non verranno ritrasmessi.
Attraverso il parametro *QUEUE_TIMEOUT*, è possibile stabilire per quando tempo un pacchetto può restare in attesa di essere ritrasmesso; a seconda della dimensione della coda, della topologia della rete, del tipo di sensori, può essere utile far variare questo parametro per permettere accodamenti più o meno lunghi. Allo scadere del *timeout*, il pacchetto scaduto verrà rimosso dalla coda e verrà segnalato un evento di perdita inserendo una riga nella tabella *QueueNode* con il campo *Remove* abilitato.

Codice 4.6: Metodo FindQueue

```
1   for (int i=0;i<nodes.size();i++){
2     ...
3     // seleziona tutti i pacchetti ricevuti e trasmessi
4     // dal nodo i
5     SQL = "SELECT .... FROM Packet ....
6     ...
7     while (result.next()) {
8
9     ...
10
11      if (da==me) {
12              // ---- pkt is for ME ----
13              coda.add(p);
14              size = coda.size();
15              coda.remove(coda.size()-1);
16              sql = "INSERT INTO QueueNode ...
17
18      } else if (da!=me & hda==me) {
19              // ---- pkt not for ME Received -----
20              coda.add(p);
21              size = coda.size();
```

```
22              sql = "INSERT INTO QueueNode ...
23
24      } else if (da!=me & hsa==me) {
25              // ---- pkt not for ME Trasmitted -----
26              int idx = coda.indexOf(p);
27              if(idx!=-1){
28                      MyPayload item = coda.get(idx);
29                      coda.remove(p);
30                      ...
31                      sql = "SELECT Value FROM QueueNode ...
32                      ...
33                      int old = r.getInt("Value");
34                      ...
35                      size = coda.size();
36                      sql = "INSERT INTO QueueNode ...
37                      ...
38              }
39      }
40      // --- CHECK OUT OF DATE PAYLOAD ----
41      for (int j=0;j<coda.size();j++){
42              if (TimeStampDiff(...) >= QUEUE_TIMEOUT) {
43                  ...
44                  coda.remove(j);
45                  ...
46              }
47      }
```

Aggiornamento dei dati nell'applicazione web I due processi appena descritti, *Processing* e *Populate*, vengono eseguiti sequenzialmente dall'applicazione web, per aggiornare le informazioni da visualizzare tramite l'interfaccia grafica. Si ricorda che durante un esperimento, i dati vengono inseriti continuamente nella tabella *RawPacket*. Ora se il primo processo *Processing* andasse ad elaborare tutti i dati presenti in quella tabella questa operazione sarebbe corretta solo alla prima iterazione. Infatti già dalla seconda iterazione si andrebbero a rielaborare tutti i pacchetti presenti prima dell'inizio di questa iterazione. La cosa avverrebbe in modo analogo per il secondo processo *Populate*.
Questo problema viene gestito attraverso degli indici che memorizzano il numero dell'ultima riga elaborata nell'iterazione precedente. In questo modo la singola iterazione andrà ad acquisire i dati da elaborare a partire da quell'indice fino alla fine della tabella.

4.2.4 Copertura

Una funzionalità implementata nel *RoutingLayer* è quella di poter visualizzare la copertura di tutti i nodi, ovvero per ogni nodo quali sono i suoi vicini con cui può comunicare e la qualità del collegamento misurata in funzione del *Received signal strength indication* (RSSI). La raccolta di queste informazioni è una parte a se stante rispetto a tutto il resto del procedimento di analisi

dell'algoritmo di routing. É stata infatti sviluppata un'applicazione Ad Hoc chiamata *Coverage*. Essa non fa altro che inviare una richiesta di copertura in broadcast, e attende le risposte dei vicini che hanno sentito la richiesta; dopo di che invia alla porta seriale tutti pacchetti ricevuti per memorizzare le risposte ricevute. La richiesta contiene un contatore per tener conto del numero di richieste inviate. Il messaggio di risposta è composto dai seguenti campi:

1. *TOS_NODE_ID* del nodo che risponde
2. identificativo del numero di richiesta ricevuta
3. valore di RSSI del messaggio di richiesta ricevuto

In questa maniera è possibile creare una statistica sulla qualità dei collegamenti correlando l'informazione dell'RSSI, all'informazione delle perdite, calcolate dal rapporto tra le risposte ricevute e le richieste totali trasmesse. Va sottolineato che il successo avviene se il ricevitore riceve la richiesta e il trasmettitore riceve la risposta, quindi diciamo che il calcolo delle perdite (*packet loss*) è calcolato su dei *link* bidirezionali.
Gli esperimenti di copertura sono stati eseguiti su tutti i nodi del testbed e quindi per evitare che due nodi effettuino la richiesta nello stesso istante o in istanti vicini, sono stati utilizzati dei periodi di tempo casuali molto lunghi; in media un nodo effettua una richiesta ogni 8 minuti. Quindi per avere un numero consistente di richieste in modo da creare una statistica significativa, gli esperimenti possono durare molte ore, ad esempio una notte intera. Anche per le risposte sono stati fatti degli accorgimenti; per evitare collisioni al ricevitore, quando un nodo riceve una richiesta non risponde subito, ma dopo un tempo aleatorio con medio 2 secondi. In questo modo si evita che le risposte arrivino tutte contemporaneamente al ricevitore.
Tutte le risposte che giungono ad un nodo fluiscono all'interno della tabella *Coverage* nella quale verrà inserita una riga per ogni risposta ricevuta. Con tutte queste informazioni è possibile valutare il numero di richieste inviate da un nodo, il numero di risposte ricevute da ogni vicino, l'RSSI medio di ogni singolo collegamento, e quindi creare una statistica sull'affidabilità dei *link*.

4.3 Guida all'uso del Routing Layer

In questa sezione verranno illustrate tutte le caratteristiche del *plugin* sviluppato, e inoltre verrà fornita una piccola guida di come estrarre le informazioni necessarie per valutare le prestazioni di un algoritmo di routing.

4.3.1 Presentazione grafica

In Figura 4.13 si possono vedere le varie maschere che compongono il *RoutingLayer* con le quali si può interagire per ottenere le informazioni richieste.

Descrizione in dettaglio delle funzioni dei pulsanti

Nel *Traffic Layer*:

- *Start* Routing Analisys: avvia il processo di raccolta dei dati e periodicamente aggiorna le informazioni.
- *Stop*: Ferma l'aggiornamento.
- *Set*: imposta il valore di aggiornamento al valore contenuto nella casella di testo a sinistra.
- *Sink Node*: imposta il nodo sink desiderato per permettere determinate visualizzazioni; se una rete è composta da più sink questa opzione permette ad esempio di visualizzare solo il traffico destinato al nodo sink scelto.
- *Options*: opzioni di visualizzazione dei percorsi: solo completi, solo incompleti, tutti.
- *Single Tree*: Visualizza un percorso alla volta; dopo aver impostato la rotta in *Route*, è possibile scegliere il percorso in *Path*
- *Complete Tree*: Visualizza tutto il traffico: solo messaggi di richiesta verso il sink, solo messaggi di risposta dal sink, tutto il traffico.
- *Build*: Disegna sulla mappa il percorso o traffico selezionati in precedenza.
- *Remove Route*: Pulisce la mappa.

Nel *Statistics Layer*:

- *Route*: visualizza la rotta scelta nel Traffic Layer: in questa maschera quasi tutte le informazioni visualizzabili si riferiscono a una singola rotta.
- *Get Route Statistics*: Disegna la rotta e mette a disposizioni informazioni su ritardo end-to-end, numero medio di hop, traffico medio.

Figura 4.9: *Traffic*

Figura 4.10: *Statistics*

Figura 4.11: *Global*

Figura 4.12: *Database*

Figura 4.13: Maschere che compongono il Routing Layer

- *Remove Routes*: Pulisce la mappa.
- *Next Hop*: Visualizza come sceglie il next hop l'algoritmo di routing. Dopo aver selezionato un nodo presente nella rotta in esame (SHIFT + click), questo pulsante visualizzerà tutti i possibili next hop che sono stati scelti; ad ogni link assocerà inoltre la perdita di pacchetti ottenuta nell'esperimento in corso.
- *Build*: a seconda dell'opzione selezionata a fianco, visualizzerà sulla mappa tutto il traffico associato a quella rotta oppure i ritardi medi ottenuti nell'esperimento.
- *Show Complete Routes*: Ripete le operazioni relative al pulsante *Get Route Statistics* per tutte le rotte che si sono completate con successo almeno una volta. E' possibile scegliere tramite l'opzione a lato se si desiderano le rotte in andata (nodi → sink) o in ritorno (sink → nodi).
- *Export to excel displayed routes*: Esporta in un foglio excel i valori medi relativi alle rotte visualizzate con il comando precedente.

Nel *Global Layer*:

- Coverage: Dopo aver selezionato un nodo (SHIFT + click), questo pulsante visualizzerà i dati di copertura presenti nella tabella *Coverage*: verrà disegnato un link per ogni vicino al nodo in esame.
- Coverage Reliability: Dopo aver selezionato un nodo (SHIFT + click), questo pulsante visualizzerà una finestra con le informazioni qualitative dei link in termini di affidabilità, RSSI medio, LQI medio.
- Rssi Threshold: É possibile visualizzare la copertura di un nodo definendo una soglia di RSSI, in modo da visualizzare solo i vicini che dai quali è possibile ricevere un segnale con valore di RSSI maggiore uguale alla soglia scelta.

4.3.2 Visualizzazione del traffico in tempo reale

Dopo aver caricato l'applicazione/i nei sensori desiderati e dopo aver lanciato gli script per l'ascolto del traffico, si può visualizzare come si distribuisce il traffico nella rete in tempo reale. Cliccando sul pulsante *Start Routing Analisys* si da inizio al caricamento delle informazioni nelle tabelle del database e nel Routing Layer, infatti esso non fa nient'altro che eseguire ciclicamente i due processi descritti in Sezione 4.2 ad intervalli di tempo

definiti dalla casella di testo *Routing refresh period*. Supponendo ad esempio che si voglia visualizzare il traffico totale in tempo reale basterà visualizzarlo la prima volta e automaticamente le linee di traffico sulla mappa verranno aggiornate fintanto che resterà attivo il pulsante *Start Routing Analisys*. In Figura 4.14 si può vedere un esempio di come viene visualizzato il traffico. Ogni linea rappresenta un link tra due nodi nel quale è avvenuta almeno una trasmissione e lo spessore della linea indica la quantità di traffico transitata nel collegamento. A linea spessa corrisponde molto traffico a linea sottile corrisponde poco traffico. Ovviamente all'inizio di una simulazione non saranno presenti link trafficati, ma all'aumentare del tempo di simulazione compariranno nuovi link o varieranno gli spessori in funzione dei percorsi presi dai pacchetti.

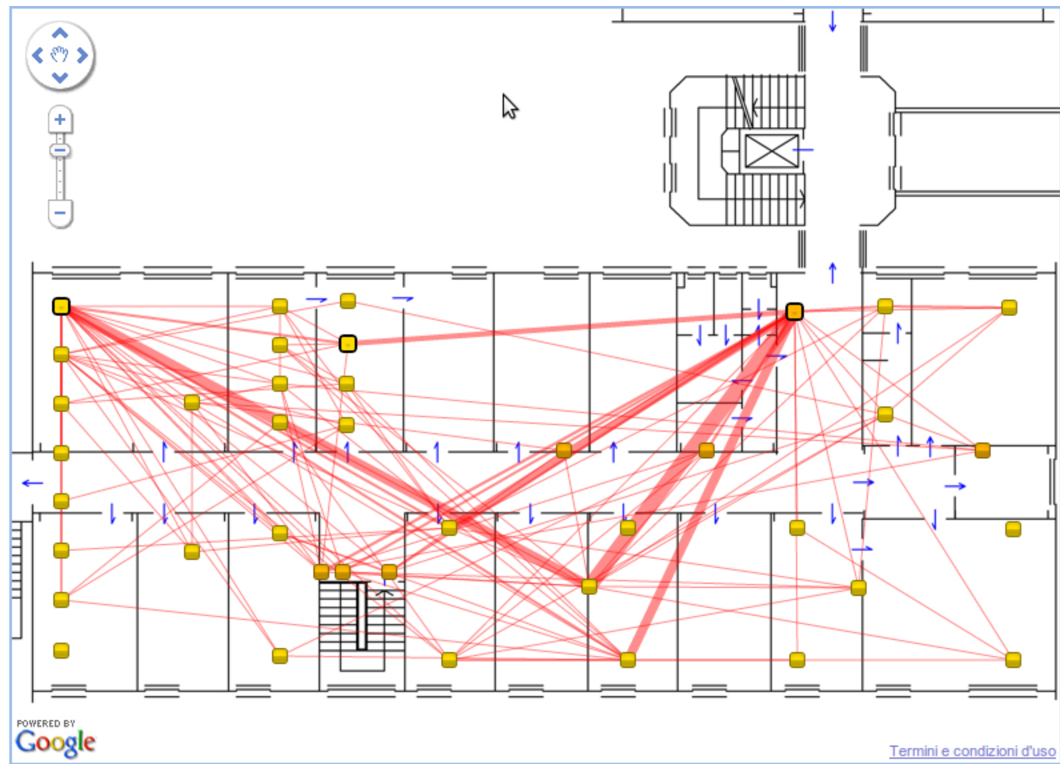


Figura 4.14: Esempio di visualizzazione del traffico

4.3.3 Visualizzazione del traffico su un singolo link

E' possibile visualizzare informazioni più dettagliate sul traffico in transito sul singolo link semplicemente spostandosi con il cursore del mouse sopra

la linea che rappresenta il link interessato; comparirà un piccolo pannello con informazioni riassuntive sul link quali ritardo medio, traffico transitato e numero di pacchetti transitati. Si potrà esplorare ancora più in dettaglio il traffico con un click sulla linea, al quale seguirà la comparsa di una finestra che darà informazioni dettagliate su tutti i pacchetti transitati, quali identificativo del pacchetto e dell’hop, timestamp, payload, valore della coda di ricezione nel nodo di destinazione. Vedi Figura 4.15.

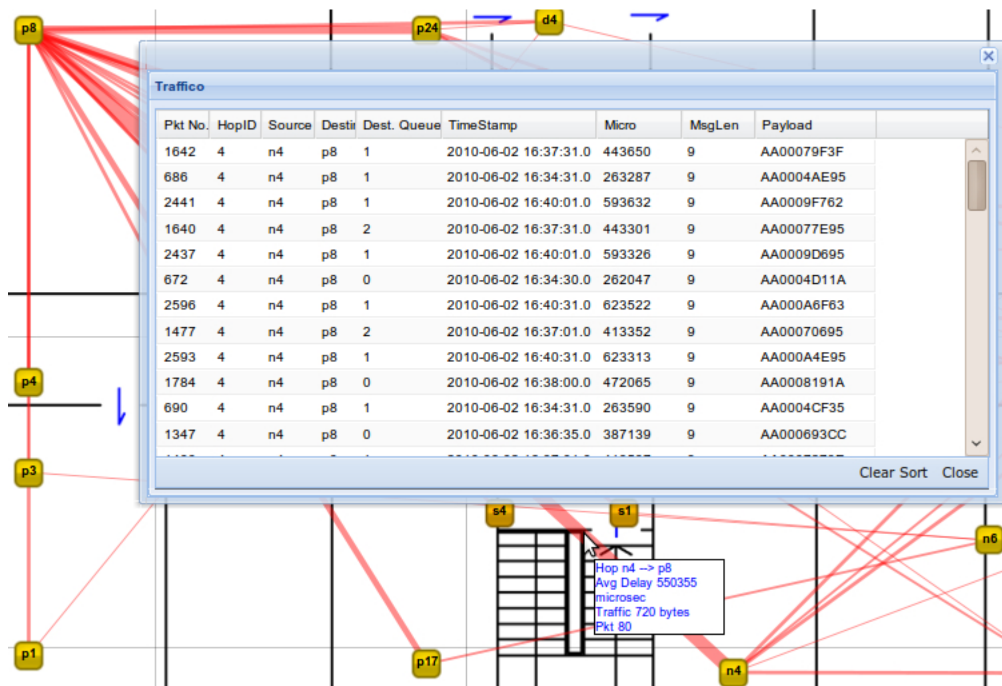


Figura 4.15: Esempio di visualizzazione dei pacchetti trasmessi sul link n4 → p8

4.3.4 Visualizzazione del traffico su una singola rotta

Volendo analizzare come si distribuisce il traffico solo in una singola rotta è possibile sia visualizzare uno per uno tutti i percorsi scelti dai pacchetti Figura 4.16, completi e incompleti, sia tracciare un albero globale del traffico dove tutti i percorsi hanno ovviamente la stessa sorgente e confluiscono tutti nella stessa destinazione Figura 4.17.

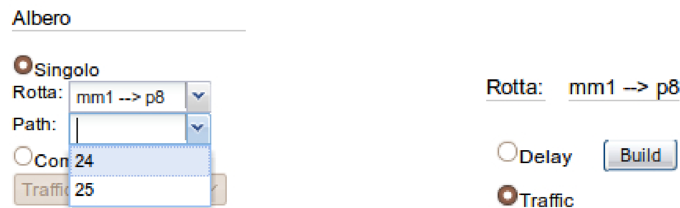


Figura 4.16: Visualizzazione percorso singolo

Figura 4.17: Visualizzazione traffico totale nella rotta

4.3.5 Costruzione del percorso hop by hop

Attraverso questa funzionalità è possibile dopo aver fissato una rotta, vedere iterativamente quali sono i possibili next hop scelti dall'algoritmo di routing durante la simulazione. Come si vede in Figura 4.18 si è scelto la rotta mm1 → p8, e sono stati visualizzati i possibili next hop della sorgente mm1. Per fare ciò è stato selezionato il nodo mm1 (SHIFT + click) ed è stato premuto il pulsante *Next Hop*. Iterativamente è possibile mostrare i next hop dei nodi successivi ripetendo le operazioni appena descritte. Infine, le linee che rappresentano i link sono colorate in base al *packet loss*; a colore più intenso corrisponde un collegamento con poche perdite, a colore più tenue corrisponde un collegamento con perdita maggiore. Anche qui, come nella visualizzazione del traffico, spostandosi con il cursore sopra una linea è possibile far visualizzare le informazioni relative al link, in questo caso la percentuale di perdita di pacchetti.

4.3.6 Visualizzazione dei ritardi sui link di una singola rotta

Sempre concentrando l'attenzione su una singola rotta è possibile vedere tutti i ritardi ottenuti sui singoli link interessati nella rotta in esame. In Figura 4.19 sono visualizzati i ritardi relativi alla rotta mm2 → p8 e più in particolare si sono esplorate tutte le singole occorrenze nel link c8 → p8. Spostandosi con il cursore sopra una linea di ritardo viene visualizzata l'informazione del ritardo medio del link, mentre al click viene visualizzata la finestra con i singoli ritardi. Come per il traffico a linea spessa corrisponde un ritardo più elevato, a linea sottile corrisponde un ritardo più ridotto. Come già descritto in sezione i ritardi dei primi link sono definiti nulli, infatti si può notare che linee che escono dalla sorgente hanno lo spessore minimo che indica un ritardo uguale a zero.

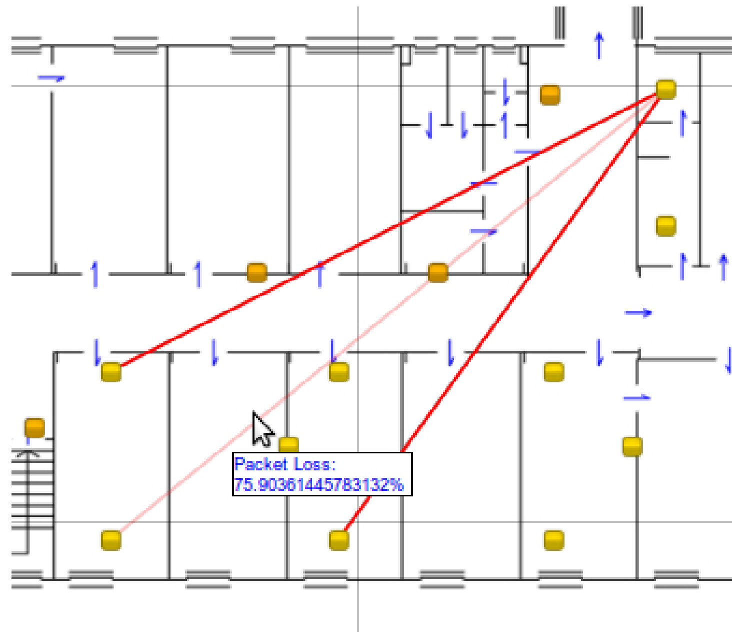
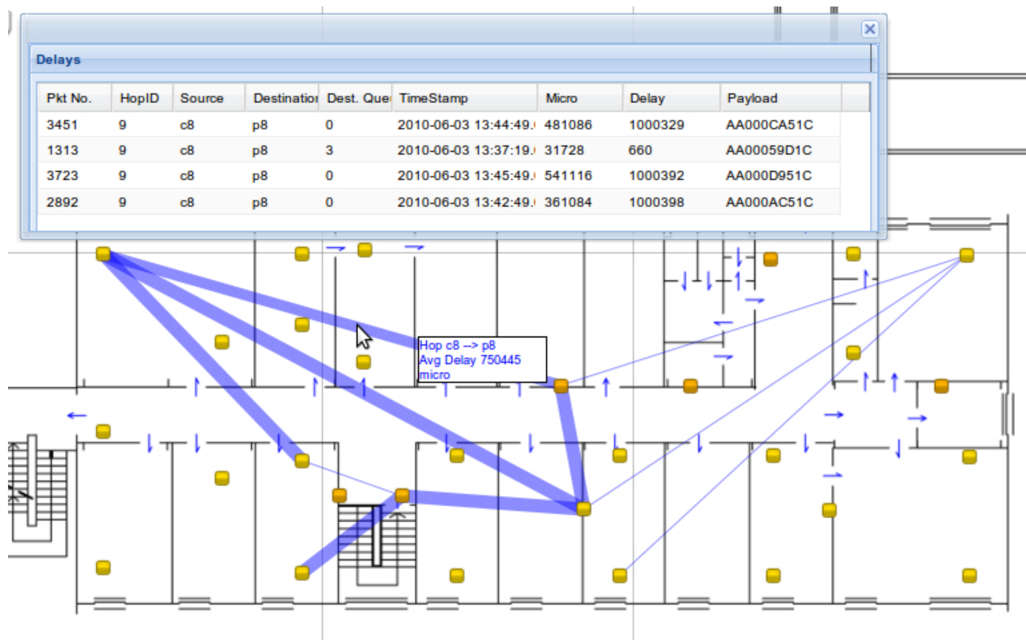


Figura 4.18: Esempio dei possibili next hop del nodo mm1



Pkt No.	HopID	Source	Destinatioı	Dest. Queı	TimeStamp	Micro	Delay	Payload
3451	9	c8	p8	0	2010-06-03 13:44:49.	481086	1000329	AA000CA51C
1313	9	c8	p8	3	2010-06-03 13:37:19.	31728	660	AA00059D1C
3723	9	c8	p8	0	2010-06-03 13:45:49.	541116	1000392	AA000D951C
2892	9	c8	p8	0	2010-06-03 13:42:49.	361084	1000398	AA000AC51C

Figura 4.19: Esempio della visualizzazione dei ritardi ottenuti sul link c8 $\rightarrow$ p8

4.3.7 Copertura

In Figura 4.20 si può vedere un esempio di copertura con relative informazioni di affidabilità, RSSI e LQI dei link. Le linee che rappresentano i collegamenti sono più o meno intense a seconda della qualità del link, a linea tenue corrisponde un link poco affidabile, a linea intensa corrisponde un link molto affidabile. Passando con il cursore sopra le linee è possibile vedere il valore medio di RSSI e LQI corrispondente. Attraverso questo strumento è possibile quindi capire in quanti Hop può completarsi una rotta e come varia questa quantità se viene scelta una soglia di RSSI per mantenere una certa affidabilità sui collegamenti.

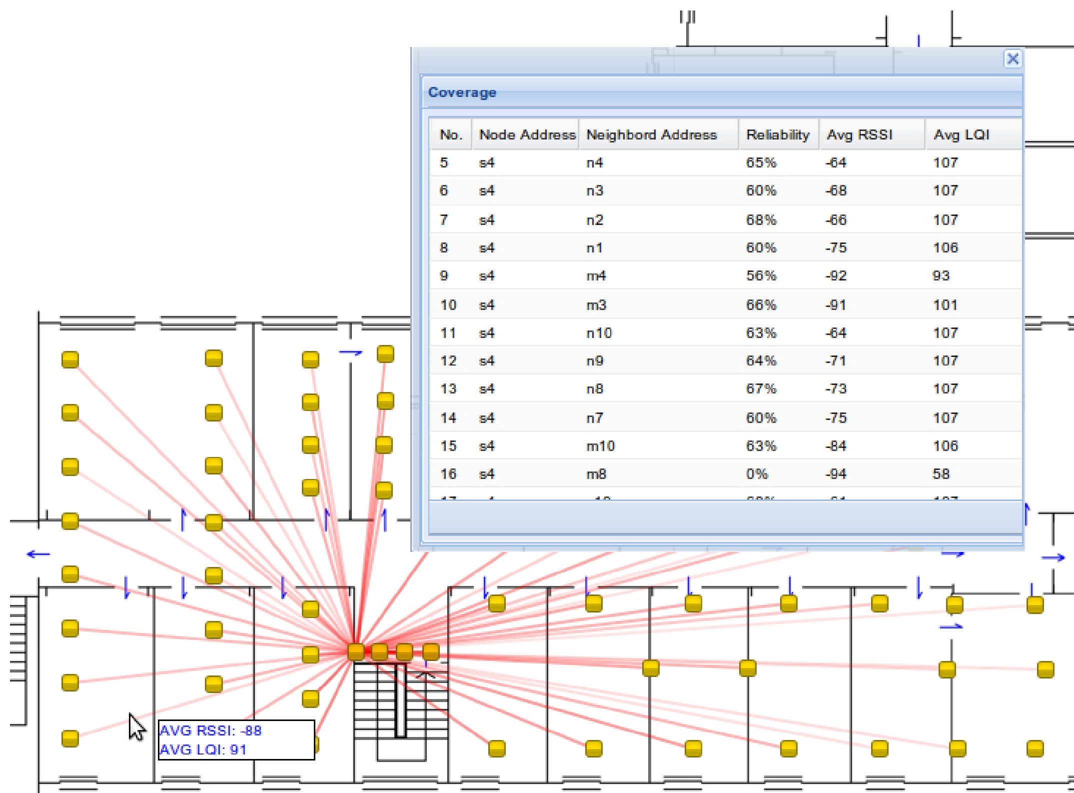


Figura 4.20: Esempio di copertura del nodo s4

4.3.8 Quantità di traffico che transita per un nodo

Può essere interessante capire quanto traffico passa per un determinato nodo per verificare ad esempio se esso fa parte di un "collo di bottiglia" nella rete. In Figura 4.21 si può vedere come viene rappresentata questa

informazione nella mappa; vengono forniti dati riguardo al numero totale di pacchetti ricevuti e trasmessi, alla percentuale di pacchetti propri e di inoltro. Nell'esempio si nota che il nodo in questione ha ricevuto 138 pacchetti e ne ha trasmessi 305, di cui il 39.95% erano pacchetti di sua proprietà, cioè destinati a lui o provenienti da lui, mentre il 60,05% erano pacchetti destinati ad altri nodi e che quindi ha ritrasmesso.

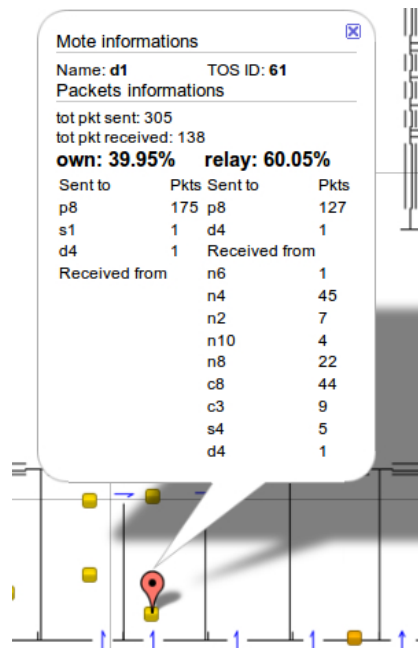


Figura 4.21: Esempio della quantità di traffico transitata per un nodo

4.3.9 Esportare i dati

Uno strumento molto utile per poter vedere elaborare e graficre i risultati di qualsiasi genere è il collegamento *Export to Excel SQL Query* (vedi Figura 4.12). Avendo una chiara conoscenza dello schema relazionale del database è possibile effettuare delle query mirate inserendo il testo della query nella casella di testa al di sopra del collegamento ed esportare il risultato in un foglio di calcolo.

Capitolo 5

Problemi riscontrati e risultati

5.1 Testbed utilizzato negli esperimenti

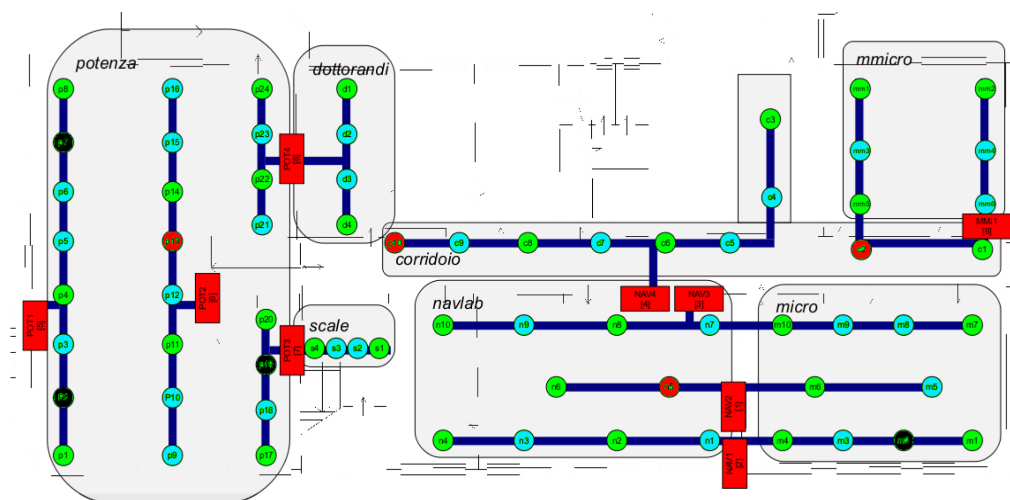


Figura 5.1: Area utilizzata nei test

In Figura 5.1 si può vedere il testbed utilizzato negli esperimenti, formato in totale da 68 nodi, di cui 4 non verranno utilizzati (nero), 4 nodi saranno utilizzati per ascoltare il traffico (BaseStation) (rosso),in 30 nodi verrà caricata l'applicazione *Echo* (verde) e 30 nodi serviranno solo da supporto ai nodi di applicazioni per inoltrare i pacchetti (blu). I nodi di applicazione generano sia traffico applicativo sia traffico di segnalazione, mentre i nodi di inoltro solo traffico di segnalazione dell'algoritmo di routing.
In tutti i test svolti verranno mantenuti fissi i nodi di applicazione e si faranno variare i nodi di inoltro per verificare come variano le prestazione alla

variare di essi.
Il nodo sink è sempre il nodo p8 in alto a sinistra e il periodo di aggiornamento delle tabelle di routing dell'algoritmo è mantenuto fisso a 30 secondi.
Considerando il raggio massimo di copertura dei nodi è possibile raggiungere il nodo sink in 3 Hop se si considerano i nodi più lontani. Tuttavia si vedrà in seguito che si formeranno percorsi con un numero maggiore di Hop.
La durata degli esperimenti varia a seconda della metrica da analizzare e verrà specificata nella sezione corrispondente.
La dimensione della coda di ricezione dei pacchetti è mantenuta fissa a 15.

5.2 Traffico sopportato dalla rete

In questo paragrafo vediamo come la rete riesce a sopportare la quantità di traffico in circolazione, graficando come varia la *Packet Delivery Ratio* in funzione della quantità di traffico immessa dai nodi che montano l'applicazione. Si intuisce che una rete di questo tipo abbia dei limiti di capacità causati sia dalla vasta densità dei nodi che quindi sono fonte di collisioni sia dalla scarse capacità di calcolo dei sensori che quindi non possono gestire ad esempio elevati accodamenti. I risultati ottenuti da un esperimento con soli 30 nodi di applicazione sono visibili in Figura 5.2. In ordinata è rappresentata la PDR mentre in ascissa il tasso di traffico immesso da ciascun nodo (il tasso di traffico è misurato da un parametro λ dove $\lambda = \#$ richieste al minuto). La PDR è stata calcolata facendo il rapporto tra tra il numero di richieste arrivate correttamente a destinazione (sink) e il numero totale di richieste trasmesse. Le curve rappresentano la media pesata su tutti i nodi a distanza di 1,2,3 Hop.

Dal grafico si nota che i nodi in copertura, 1 Hop, presentano una probabilità di successo pari a 1, cioè trasmettono sempre correttamente al sink, mentre diverso è il comportamento per i nodi a distanza di 2 o 3 Hop. Si nota inoltre che le 3 curve rimangono pressoché costanti fino ad un certo valore di λ. Ciò sta a significare che fino ad una certa quantità di traffico immessa i nodi riescono a gestire correttamente gli inoltri, invece superando questa soglia il traffico è troppo elevato e le collisioni e gli alti accodamenti fanno crollare le prestazioni. Si può vedere che per i nodi a distanza di 2 Hop la soglia λ è uguale a 20 pacchetti al minuto, mentre per i nodi a distanza di 3 Hop vale 12 pacchetti al minuto.
Per ricavare questo tipo di grafico (PDR in funzione di λ) è stato effettuato un esperimento per ogni tasso di traffico. La durata del singolo esperimento varia a seconda del tasso di traffico. Per creare una statistica significativa si

è cercato di far durare l'esperimento in modo tale che ogni nodo trasmetta in media 300 richieste al nodo sink (Questo valore sarà raggiunto tanto prima quanto più elevato è il tasso di traffico analizzato).

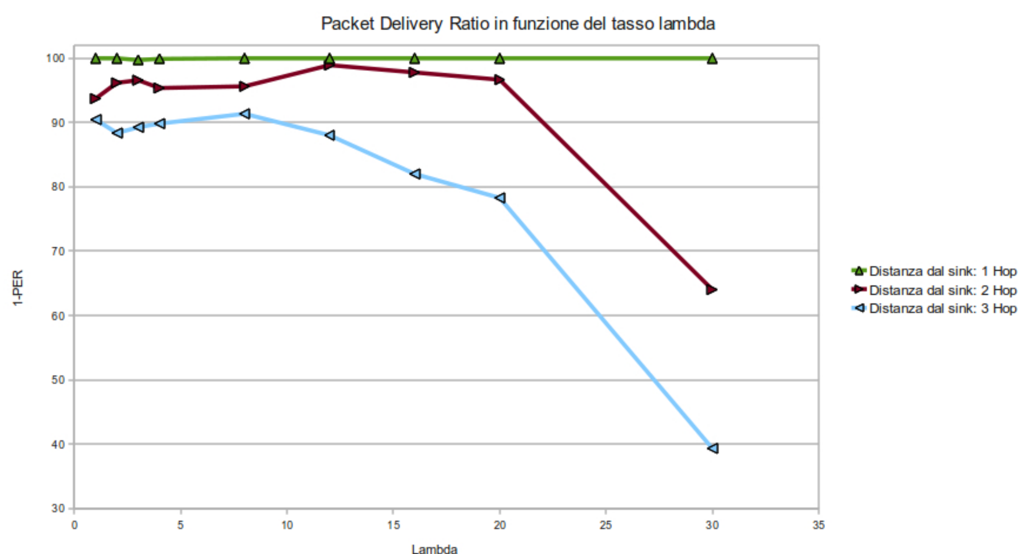


Figura 5.2: PDR di una configurazione con 30 nodi di applicazione

5.3 Presenza di nodi di relay

In questo paragrafo si vede come variano le prestazioni, in termini sempre di PDR, se oltre ai nodi di applicazione introduciamo dei nodi di relay. La presenza di questi nodi introdurrà un maggiore traffico di segnalazione, ma si presume che aiuteranno a distribuire meglio il traffico che non dovrà più concentrarsi solo sui nodi di applicazione come nell'esperimento precedente. In Figura 5.3 si possono vedere i risultati dove ogni coppia di curve rappresenta ancora la distinzione in base al numero di Hop di distanza dal sink. Dal grafico si nota che le prestazioni dei nodi in copertura sono identiche alla situazione precedente. Questo era intuitivamente ovvio in quanto non necessitano di trasmissioni multi hop. Per i nodi più distanti invece la situazione è diversa, infatti si nota che effettivamente le prestazioni sono migliori in presenta di questi nodi supplementari. La curva che rappresenta questa nuova situazione sta sempre al di sopra della sua corrispondente senza nodi di inoltro.
Guardando ad esempio le curve relative ai nodi a distanza di 3 Hop, per λ fissato, si è ottenuto un miglioramento fino al 10% della PDR. Nei nodi a

distanza di 2 Hop si ottiene un miglioramento massimo del 5% fino ad un tasso di traffico λ pari 20. Il miglioramento invece arriva fino al 20% con un tasso di traffico λ pari 30.

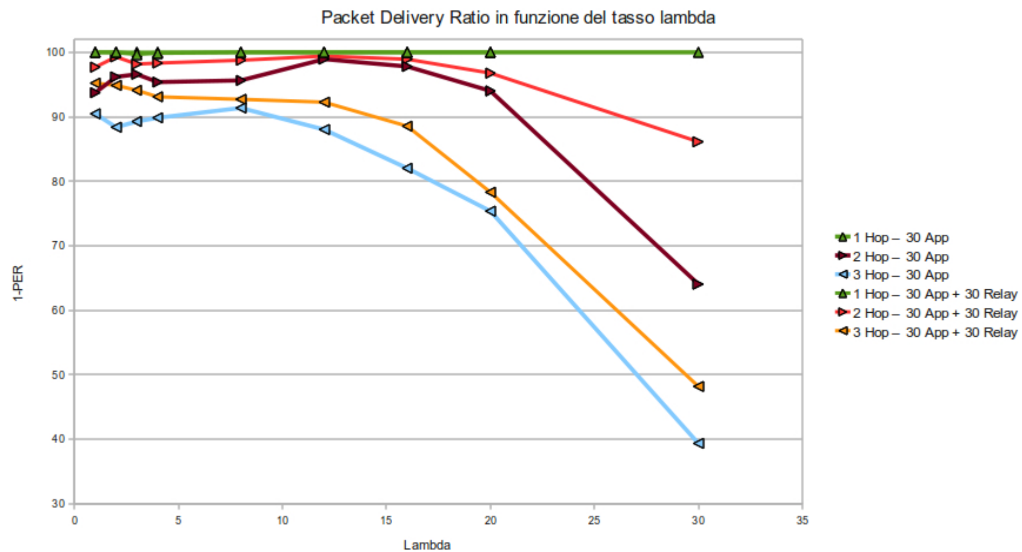


Figura 5.3: Differenza tra un esperimento senza nodi di relay e con nodi di relay

5.4 Ritardo End-To-End

In questo paragrafo si andrà a misurare il ritardo ent-to-end, ovvero il tempo trascorso tra la prima trasmissione di un pacchetto e la ricezione da parte della destinazione finale. Anche questo parametro è molto significativo in quanto riesce a dare una misura di congestione della rete. La causa di congestione di una rete è l'elevato traffico in circolazione. Infatti un nodo che deve inoltrare molti pacchetti necessita di più tempo per processarli (alti accodamenti) e questo crea ritardi nella ritrasmissione dei pacchetti che aumentano il ritardo end-to-end. Questo parametro dà inoltre una buona stima della latenza nella comunicazione tra nodi distanti e il sink. Si andrà inoltre a verificare anche in questo caso se la presenza di nodi di inoltro aumenta le prestazioni o le peggiora. Per ogni tasso di λ è stato graficato l'andamento di 4 possibili configurazioni:

- 30 nodi di applicazione + 0 nodi di relay
- 30 nodi di applicazione + 10 nodi di relay

- 30 nodi di applicazione + 20 nodi di relay
- 30 nodi di applicazione + 30 nodi di relay

Nelle Figure 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, si nota come al variare del tasso di traffico immesso nella rete le diverse configurazioni assumano andamenti diversi. Innanzitutto all'aumentare di λ si può vedere che aumenta sia il ritardo end-to-end sia il numero di Hop utilizzati. Ad esempio per un tasso di traffico $\lambda = 4$ (Figura 5.4) si è ottenuto un valore massimo di ritardo pari a 1000 ms, e un numero massimo di Hop pari a 3, invece per un tasso di traffico $\lambda = 30$ (Figura 5.9) si è ottenuto un valore massimo di ritardo pari a 6200 ms, e un numero massimo di hop pari a 5. L'aumento di ritardo end-to-end è causato come già detto dai ritardi che avvengono nei nodi intermedi. L'aumento del numero di Hop dei percorsi è causato probabilmente dalle collisioni che avvengono con maggior frequenza in presenza di un traffico più elevato. In questa situazione un messaggio trasmesso ad un nodo lontano ha più probabilità di collidere rispetto ad un messaggio trasmesso ad un nodo vicino. Questa considerazione vale anche per i messaggi di segnalazione dell'algoritmo di routing necessari per costruire le tabelle di instradamento. Sarà quindi più probabile che si creino delle relazioni padre-figlio, come descritto in Sezione 3.4.2, tra nodi più vicini, rispetto a nodi più lontani.

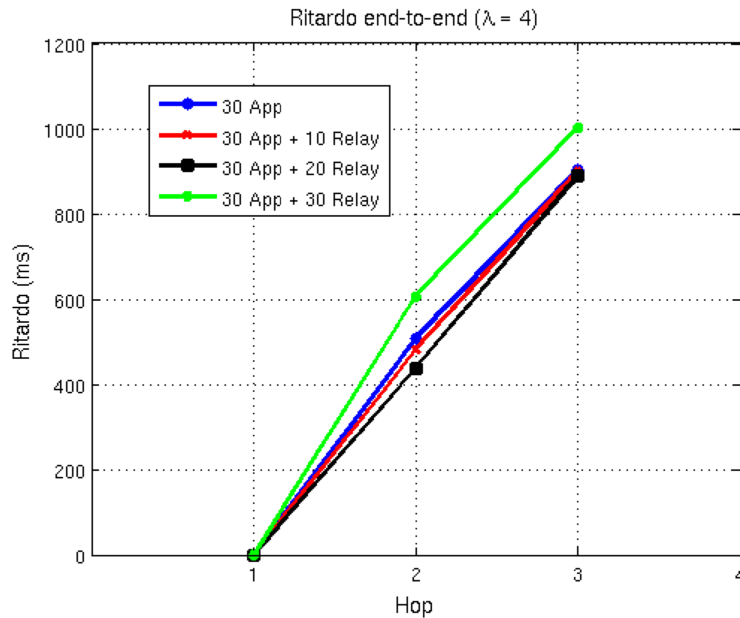


Figura 5.4: $\lambda = 4$

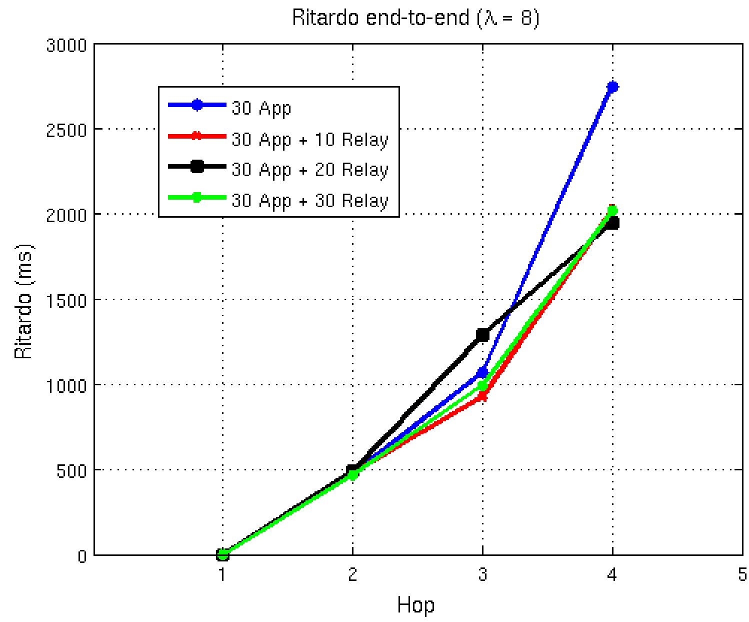


Figura 5.5: $\lambda = 8$

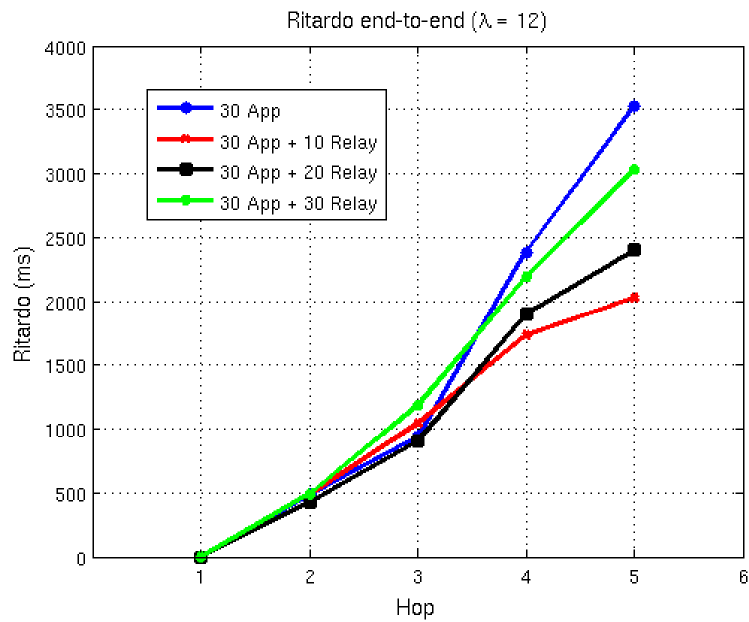


Figura 5.6: $\lambda = 12$

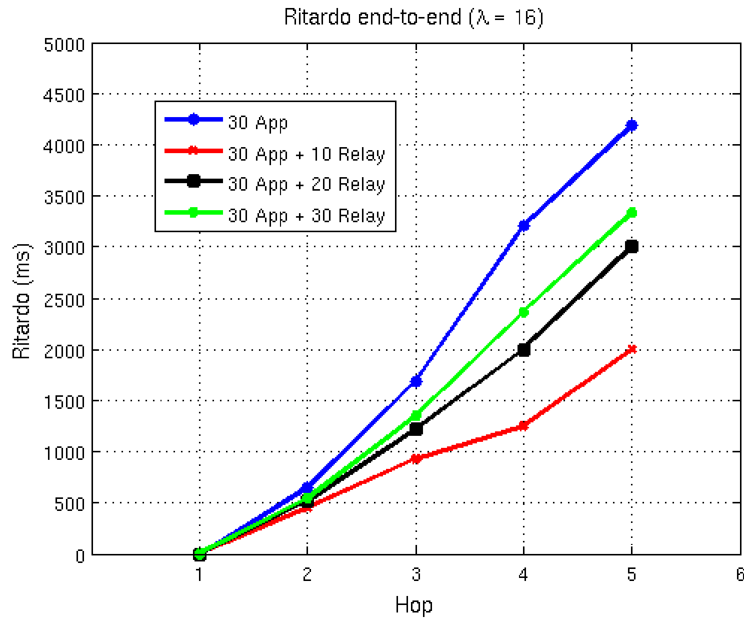


Figura 5.7: $\lambda = 16$

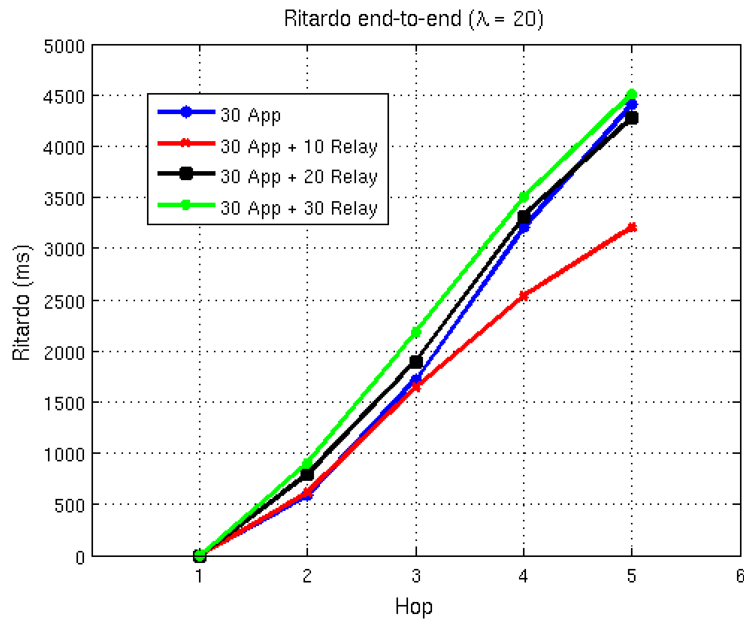


Figura 5.8: $\lambda = 20$

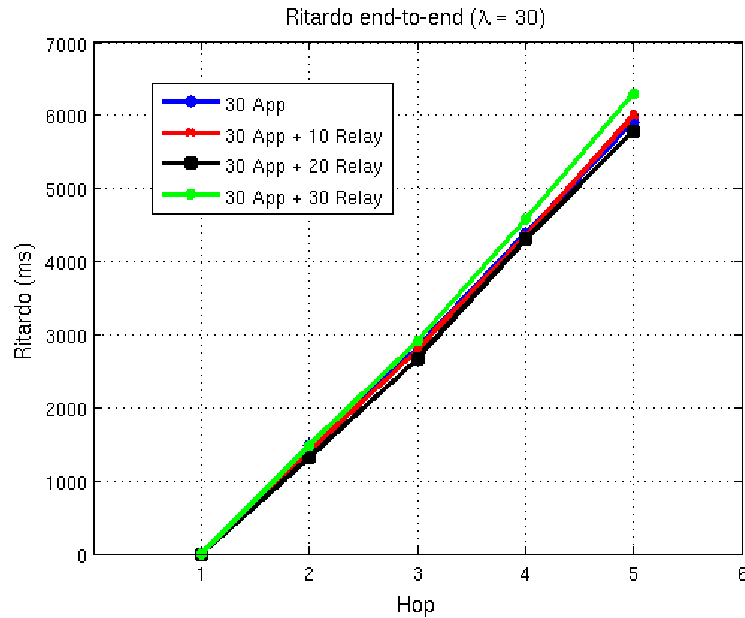


Figura 5.9: $\lambda = 30$

Le prestazioni che si andranno ora a commentare sono in funzione del numero di Hop, per λ fissato. L'introduzione di nodi di inoltro ha portato a dei miglioramenti, solo in alcuni casi particolari; infatti si nota che in presenza di traffico limitato che circola in rete Figura 5.4, l'andamento delle curve che rappresentano le varie configurazioni è pressoché uguale, mentre all'aumento graduale del traffico l'andamento delle curve evidenzia delle situazioni che riescono a ridurre il ritardo. Come si vede in Figura 5.6 e Figura 5.7 infatti, la presenza di nodi di inoltro diminuisce il ritardo end-to-end.

Inoltre, a parità del numero di Hop, il gap tra la configurazione che presenta il ritardo massimo e quella che presenta il ritardo minimo, aumenta all'aumentare della distanza dal sink. Si vede ad esempio nella figura corrispondente a $\lambda = 16$ come aumenti la distanza tra le curve in corrispondenza degli hop 4 e 5. La configurazione che presenta un miglior andamento è quella formata da 30 nodi di applicazione e 10 nodi di relay che ha ottenuto una riduzione significativa del ritardo pari a 2300 ms rispetto alla configurazione peggiore ovvero quella formata da 30 nodi di applicazione.
Aumentando ancora il traffico (Figura 5.8) questa configurazione (30+10) ottiene ancora risultati migliori rispetto alle altre configurazioni ma in maniera minore rispetto al caso precedente; si è ottenuto una riduzione di circa 1000 ms rispetto al caso peggiore. Infine si è giunti ad una situazione di saturazione della rete in corrispondenza di un fattore λ pari a 30 nella quale la rete non

ottiene più miglioramenti dall'introduzione di nodi supplementari, infatti le curve sono sovrapposte. Le configurazioni 30+20 e 30+30, portano a dei miglioramenti rispetto alla configurazione peggiore, ma in maniera ridotta rispetto alla configurazione 30+10. L'aggiunta dei nodi di inoltro migliora le prestazioni ma introduce del traffico supplementare di segnalazione. Esisterà quindi un punto di minimo (numero di nodi di inoltro) tale per cui la rete ottiene il miglioramento massimo in termini di ritardo end-to-end. La configurazione 30+10 è quindi quella che si avvicina di più a questo punto.
Dall'analisi di questi grafici si è visto che l'aumento del ritardo end-to-end in funzione del numero di Hop è pressoché lineare e che in situazioni di particolare traffico nella rete esistono configurazioni che riducono il ritardo. Alla luce di questo si è voluto graficare, sempre con dati ricavati dallo stesso esperimento, come variano le curve in funzione questa volta del tasso di traffico λ per poter effettivamente verificare per ogni tasso di traffico qual'è la configurazione migliore.

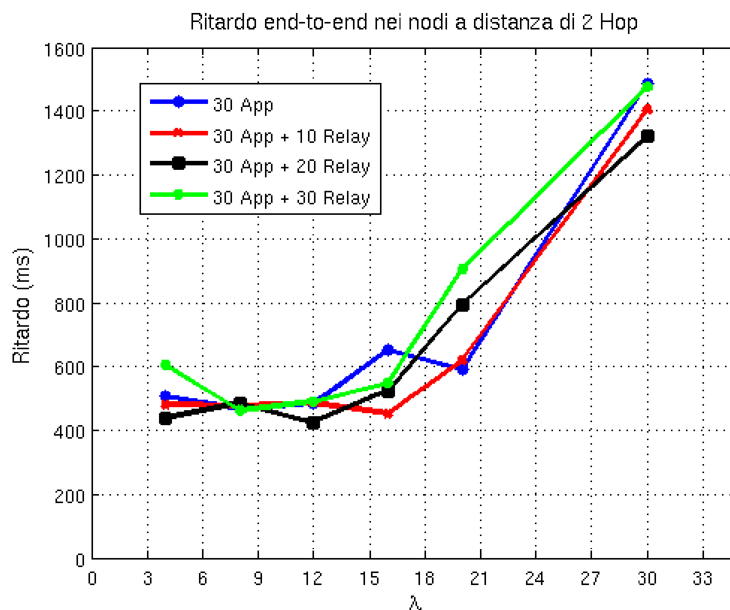


Figura 5.10: Differenza tra un esperimento senza nodi di relay e con nodi di relay

Dai grafici ottenuti si può notare innanzitutto il fatto che l'incremento di ritardo non sia lineare con l'aumentare di λ; infatti come si vede in Figura 5.10 e in Figura 5.11 il ritardo è pressoché costante fino ad una certa soglia di traffico. Dopo di chè la rete inizia ad entrare in saturazione e i ridardi di inoltro nei nodi si fanno sempre più significativi e ne consegue una crescita

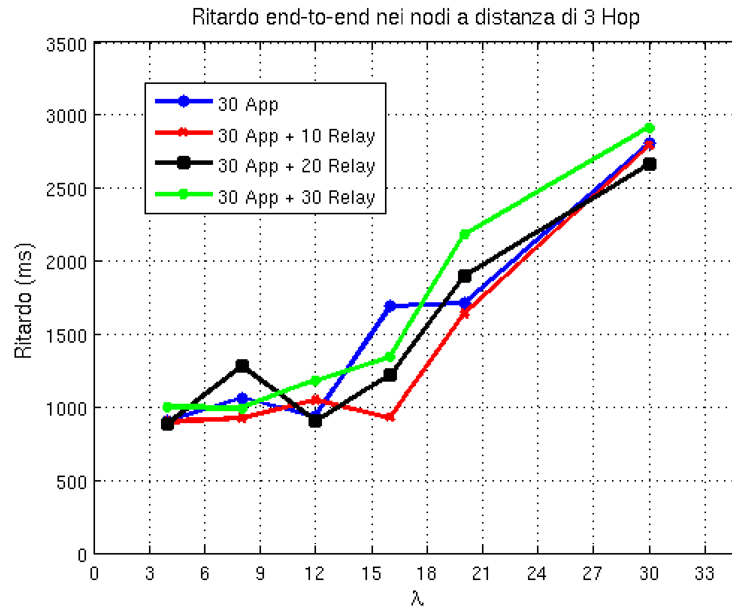


Figura 5.11: Differenza tra un esperimento senza nodi di relay e con nodi di relay

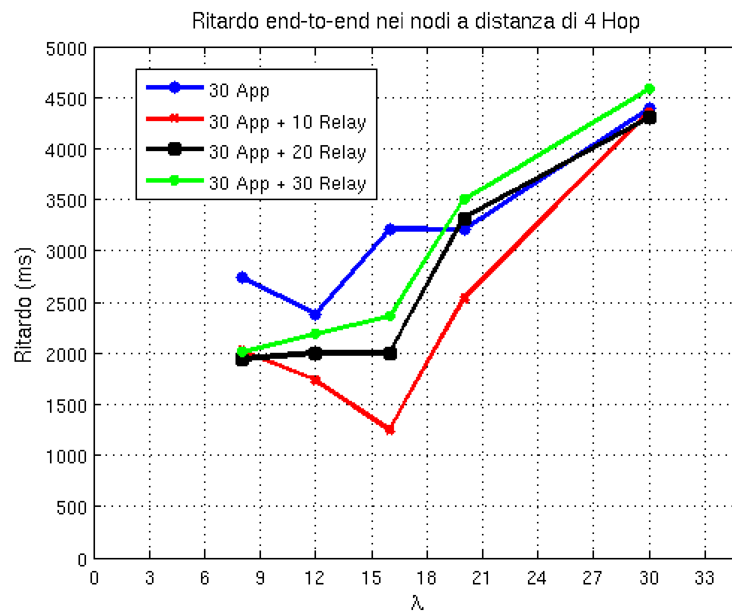


Figura 5.12: Differenza tra un esperimento senza nodi di relay e con nodi di relay

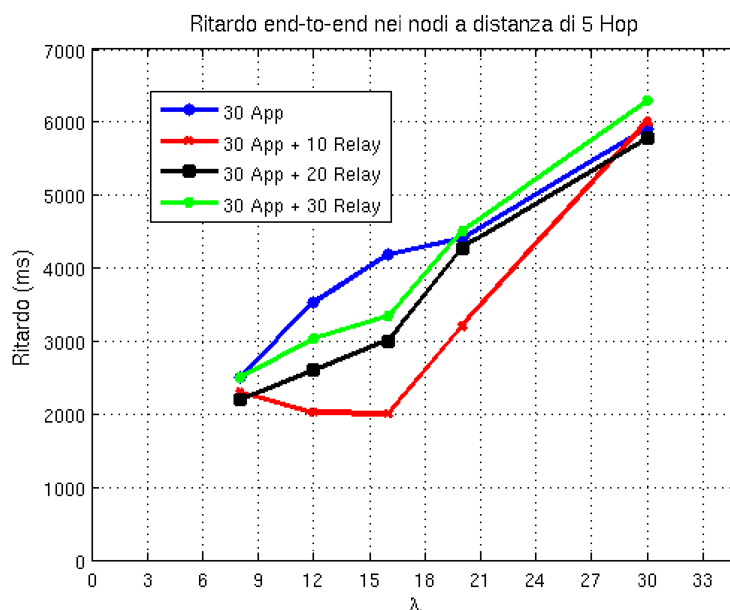


Figura 5.13: Differenza tra un esperimento senza nodi di relay e con nodi di relay

esponenziale del ritardo end-to-end.
Si noti come le soglie di capacità della rete trovate in precedenza dalla Figura 5.2 vengano riscontrate anche in questi grafici che rappresentano una metrica diversa.

Nei due grafici relativi ai nodi a distanza a 2 Hop (Figura 5.10) e a 3 Hop (Figura 5.11) non viene evidenziata una configurazione che abbia un miglioramento sostanziale nella riduzione del ritardo, infatti le curve seguono quasi lo stesso andamento. La situazione è diversa invece nei grafici relativi ai nodi a 4 hop (Figura 5.12) e 5 Hop (Figura 5.13) nei quali per ogni valore di λ la curva relativa alla situazione 30+10 rimane sempre al di sotto delle altre confermando quindi i risultati riscontrati in precedenza. Dalle Figure 5.4,...,5.9 si era visto infatti che in tutti i grafici (tasso di traffico), la configurazione 30+10 aveva prestazioni migliori.

5.5 Accodamento nei nodi

In questo paragrafo si andrà ad analizzare l'aspetto dell'accodamento nei nodi, causa di ritardi di inoltro dei pacchetti e perdite, come già visto in

precedenza. Si andrà quindi a quantificare il tempo necessario ad un nodo per inoltrare un pacchetto, cioè il tempo tra la ricezione e la ritrasmissione in funzione della posizione in ingresso nella coda di ricezione. Questo parametro è ovviamente influenzato dal traffico in rete in quanto ad esempio un maggiore traffico causerà delle code più lunghe e quindi maggiori ritardi.
Il modo con cui sono stati calcolati questi ritardi in funzione della posizione in coda è differente dal modo descritto in Sezione 4.3.6. In quella Sezione un pacchetto ricevuto veniva sempre inserito nel vettore per il calcolo della coda e nel caso quel pacchetto non fosse ritrasmesso veniva eliminato dalla coda da un processo apposito. Questo veniva fatto per tener conto contemporaneamente sia delle perdite sia della dimensione della coda. In questo paragrafo invece l'interesse è rivolto solo ai ritardi di inoltro e questi possono essere calcolati solo tramite pacchetti che vengono ritrasmessi. Prima di inserire un pacchetto ricevuto nel vettore della coda si effettua quindi un controllo per verificare se sarà inoltrato al successivo next hop. Si è effettuato un esperimento con 30 nodi di applicazione e un tasso di traffico $\lambda = 20$.

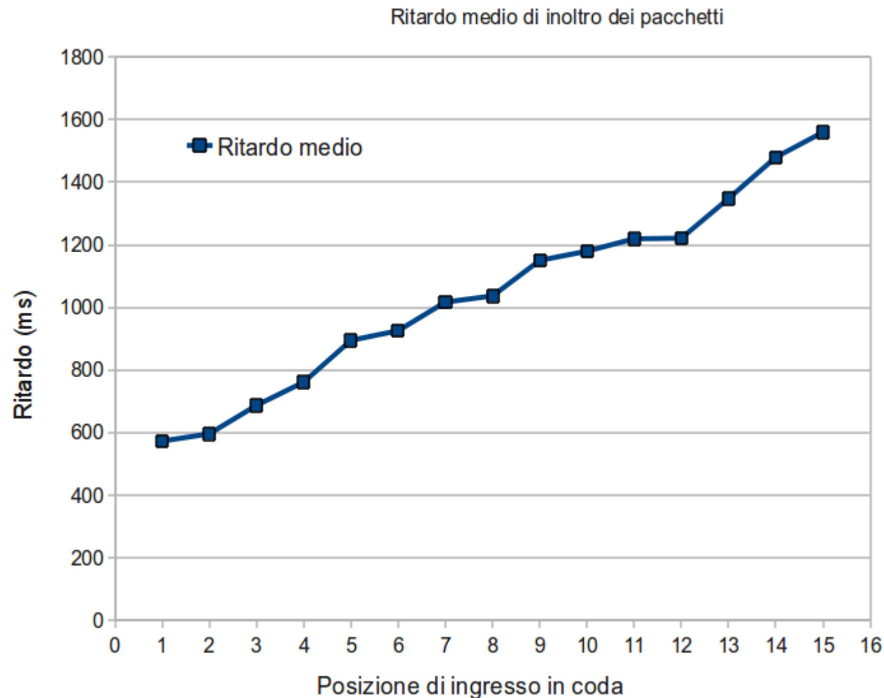


Figura 5.14: Ritardo medio di inoltro dei pacchetti in funzione della posizione in coda

In Figura 5.14 si nota come l'incremento del ritardo di inoltro cresca linearmente con la posizione in coda. Nel nodo preso in esame il ritardo medio

minimo è stato di 580 ms quando la coda di ricezione era vuota, mentre il ritardo medio massimo è stato di 1580 ms per i pacchetti entrati in coda nella posizione 15. Si ricorda che la dimensione della coda di ricezione dei pacchetti è pari a 15. Nel caso avvenga l'arrivo di un pacchetto e la coda è già piena, questo verrà scartato dal nodo.

Si andrà ora ad analizzare un altro aspetto importante cioè, la perdita dei pacchetti dovuta all'accodamento dei nodi. In questo caso il calcolo della dimensione della coda è stato fatto come descritto in Sezione 4.3.6. Si ricorda l'uso del parametro $MAX_QUEUE_TIMEOUT$ che rappresenta il tempo massimo per cui un determinato pacchetto può aspettare in coda di essere inoltrato. Dall'esperienza maturata si è visto che venivano raggiunti al massimo ritardi di inoltro pari a 2 secondi. Si è effettuata quindi la stima della probabilità di perdita di un pacchetto in funzione della posizione in ingresso in coda, scegliendo il parametro $MAX_QUEUE_TIMEOUT$ = 2,5 secondi. Per ogni valore della posizione di ingresso in coda (x), la stima della probabilità di perdita viene calcolata:

$$P_e(x) = 1 - \frac{\#pkt\ ritrasmessi}{\#pkt\ ricevuti\ in\ posizione\ x} \tag{5.1}$$

Si è effettuato l'esperimento sempre con un tasso di traffico pari a λ=20 e con una configurazione composta da 30 nodi di applicazione.
Dalla Figura 5.15 di nota come la stima della probabilità di perdita cresca in modo piuttosto lineare con la posizione di ingresso in coda. Si è ottenuto un valore minimo della stima di pari al 0,08 in corrispondenza della coda vuota, e un valore massimo dello 0,33 in corrispondenza dell'ingresso in coda nell'ultimo posto disponibile.

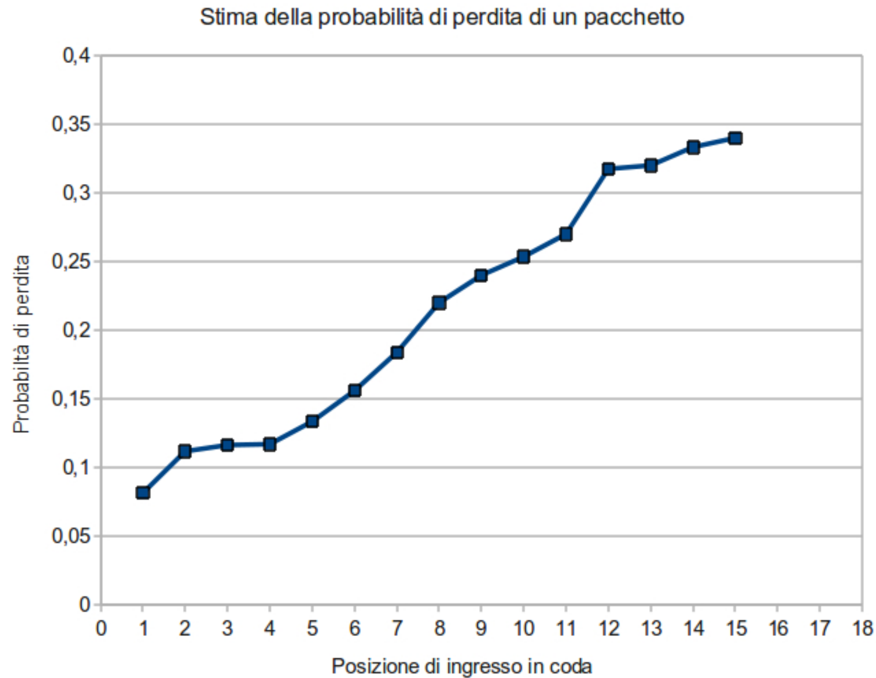


Figura 5.15: Ritardo medio di inoltro dei pacchetti in funzione della posizione in coda

5.6 Link poco affidabili

Incrociando le informazioni date dallo strumento di copertura dei nodi e i risultati ottenuti della PDR, si è scoperto un possibile problema dell'algoritmo di routing testato. In un test effettuato si è scoperto ad esempio che la rotta mm1 → p8 presentava un basso tasso di successo, quindi andando a visualizzare il traffico relativo a questa rotta si è visto, come si vede in Figura 5.16, che il link mm1 → s1 presentava una alta perdita. Infatti graficamente si vede che il traffico in ingresso al nodo s1 è maggiore rispetto al traffico in uscita e si intuisce che in questo collegamento vi è una perdita importante e probabilmente è una delle cause possibili che causano delle basse prestazioni. Inoltre dalla visualizzazione di copertura si è visto, come si vede in Figura 5.17 che il link in questione ha appunto una scarsa affidabilità (7%). Si è concluso quindi che questa situazione sia fonte di perdite dovute dal fatto che l'algoritmo di routing ha scelto un next hop non affidabile; al momento di aggiornamento delle proprie tabelle di routing il nodo mm1, ha sentito come possibile next hop verso il sink il nodo s1, non tenendo conto della qualità del link e lo ha memorizzato in tabella. Ma questo può essere avvenuto in un istante in cui il canale si trovava momentaneamente

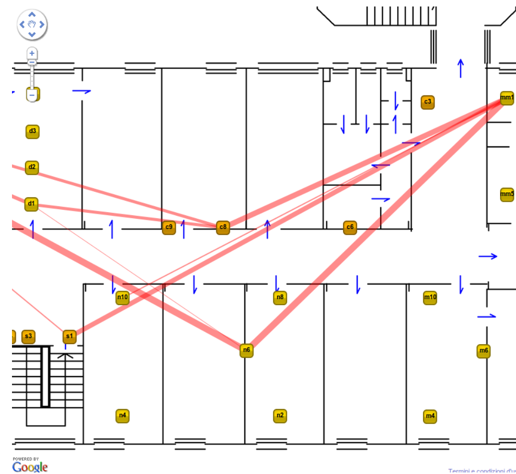


Figura 5.16: Esempio di canale scadente

in uno stato favorevole. In seguito se il canale peggiora (fading), il nodo mm1 continuerà a trasmettere al nodo s1 per tutta la durata del periodo di aggiornamento della tabella, causando la perdita di pacchetti.
Si è pensato quindi di inserire una soglia di RSSI all'interno del modulo di routing, in modo che un pacchetto ricevuto con un valore di RSSI inferiore al limite scelto verrà scartato e non sarà quindi utilizzato per aggiornare le tabelle. Questo permetterà di avere trasmissioni solo su link che rispettino i criteri di affidabilità scelti.
Dato che l'RSSI è una misura di potenza ricevuta e quindi strettamente legato con la distanza, con questa modifica verrà ridotto il raggio di copertura dei nodi. I percorsi dei pacchetti quindi necessiteranno di più Hop per arrivare a destinazione e verrà generato più traffico causato dal numero maggiore di ritrasmissioni.

Per stabilire le soglie di RSSI da inserire nell'algoritmo di routing si è cercato di ottenere un grafico che mostrasse come varia l'affidabilità in un link in funzione della sua qualità (RSSI). Per ottenerlo si è utilizzata l'applicazione di copertura. Come già descritto in Sezione 4.2.4 nella tabella che memorizza i dati di copertura vengono salvate tutte le risposte ricevute dai nodi che hanno generato le richieste. É quindi possibile ottenere per ogni valore di RSSI un valore di affidabilità media del link. Il calcolo di questo valore è stato fatto raggruppando tutti i link con lo stesso valore di RSSI medio e mediando le loro affidabilità corrispondenti. Si ricorda che il calcolo dell'affidabilità è dato dal rapporto tra le risposte ricevute e le richieste inviate. Si è quindi ottenuto il grafico che si vede in Figura 5.18

Coverage

No.	Node Address	Neighbord Address	Reliability
21	mm1	c9	29%
22	mm1	c8	34%
23	mm1	c7	32%
24	mm1	c6	35%
25	mm1	c5	41%
26	mm1	c4	44%
27	mm1	c3	41%
28	mm1	s1	7%
29	mm1	d4	0%
30	mm1	mm2	42%
31	mm1	mm4	42%
32	mm1	mm6	42%
33	mm1	c1	34%

Figura 5.17: Esempio di canale scadente

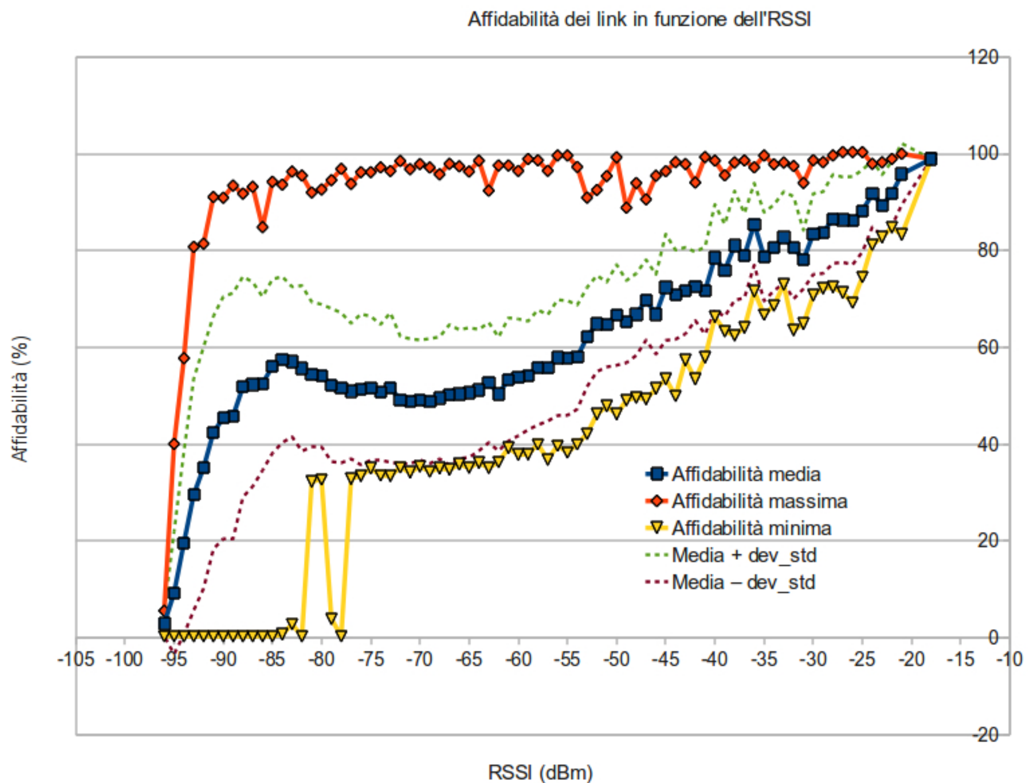


Figura 5.18: Andamento dell'affidabilità di un link in funzione dell'rssi

Nell'analisi di questo grafico va tenuto conto di un aspetto molto importante. L'evento di successo si verifica quando il pacchetto di richiesta giunge al destinatario e il pacchetto di risposta giunge al mittente. Quindi questi

valori di affidabilità sono calcolati su dei link bidirezionali. La scelta della soglia di RSSI dovrà tener conto di questo fatto.
Dalla Figura 5.18 si nota che il valore di RSSI minimo raggiunto è -97dBm e quello massimo è -18dBm. Il grafico è stato ottenuto con i risultati di un esperimento di copertura di durata di una notte, dove ogni nodo ha trasmesso mediamente 300 richieste di copertura.

Si è scelta una soglia tale che possa garantire una affidabilità media non inferiore al 50%. Dal grafico si può ricavare il valore di RSSI corrispondente: -65dBm. Con lo strumento di copertura si è voluto verificare se la scelta appena fatta fosse ragionevole per la tipologia della rete usata negli esperimenti.

Impostando nella casella di testo del Routing Layer la soglia di -65 richiesta, si è graficato, iterando la funzione copertura, in quanti Hop è possibile raggiungere il sink a partire da un nodo lontano. In Figura 5.25 sono rappresentate una alla volta le iterazioni eseguite. Il nodo selezionato rappresenta il nodo su cui si è richiesta la copertura. Dalla figura si vede come i collegamenti siano molto più corti rispetto ad esempio a quelli visti i Figura 5.16 e sono necessari 6 Hop per raggiungere il sink. Dalla pratica con la rete in esame, si è visto che una rotta di questo tipo, può completarsi in 3 Hop. Con questa scelta sull'RSSI il numero di Hop si è raddoppiato e si ritiene quindi che questo valore non sia appropriato. Con dei collegamenti così corti il traffico generato aumenterebbe in modo esponenziale e causerebbe delle prestazioni pessime anche con un tasso di traffico λ ridotto.
Si sono quindi fatte delle scelte, tenendo conto anche dell'esperienza maturata su questo tipo di reti, in particolare scegliendo dei valori di RSSI ragionevoli per la rete in esame. Sono stati ripetuti i test relativi alla PDR con 3 soglie differenti scelte in questo modo:

1. **-75 dBm**: rappresenta l'ultimo valore tale per cui l'affidabilità minima è non nulla.
2. **-85 dBm**: rappresenta l'ultimo valore tale per cui l'affidabilità media decresce lentamente; dopo di questa soglia la curva decresce in modo molto di rapido.
3. **-90 dBm**: rappresenta l'ultimo valore tale per cui l'affidabilità massima non cala drasticamente.

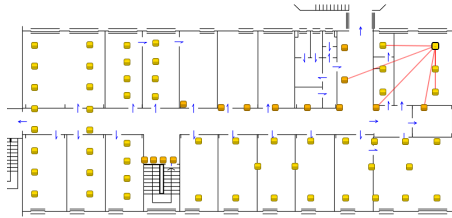


Figura 5.19: Primo Hop

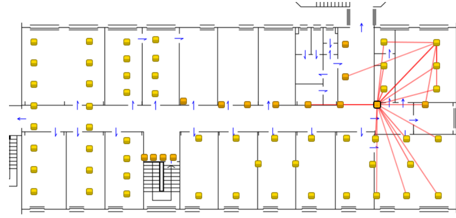


Figura 5.20: Secondo Hop

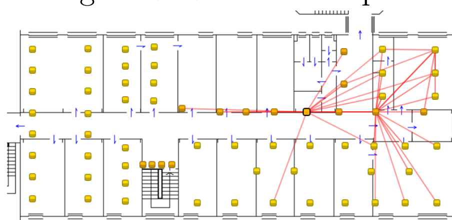


Figura 5.21: Terzo Hop

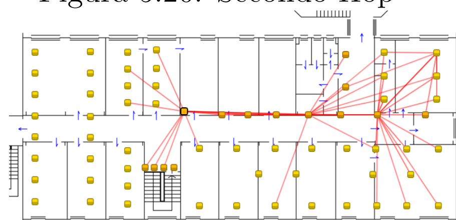


Figura 5.22: Quarto Hop

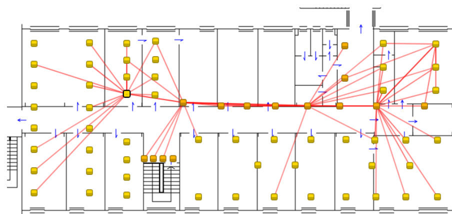


Figura 5.23: Quinto Hop

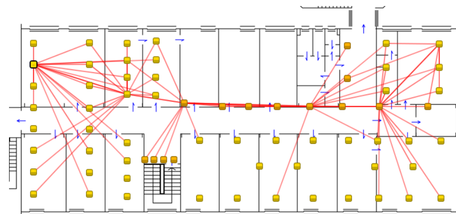


Figura 5.24: Sesto Hop

Figura 5.25: Iterazione dello strumento di copertura per verificare il numero di Hop necessari per raggiungere la destinazione

In Figura 5.26 si possono vedere i risultati di un esperimento effettuato con 30 nodi di applicazione e 30 nodi di relay. Le curve si riferiscono ai nodi a distanza di 3 hop.

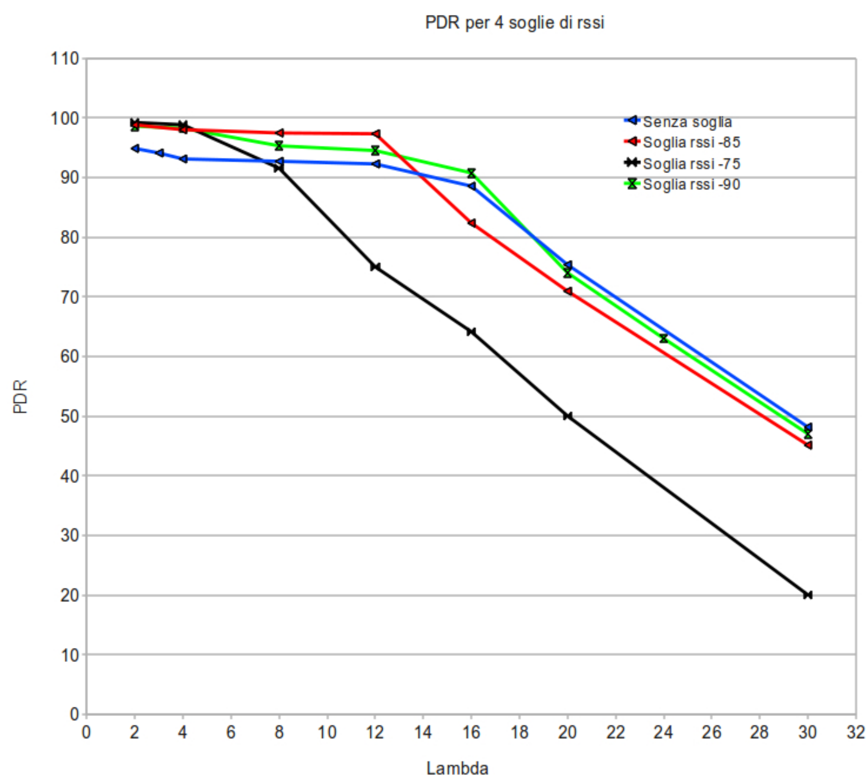


Figura 5.26: Andamento della PDR per 4 soglie di rssi

La curva di riferimento è quella di colore blu che rappresenta il caso in cui non viene inserita nessuna soglia di RSSI. Dai risultati nei grafici relativi alla PDR si era individuata una soglia di traffico λ tale per cui le prestazioni della rete subivano un calo approssimativo del 40 % per valori di λ superiori a questa soglia. Dal grafico si nota che la soglia di traffico diminuisce all'aumentare della soglia di RSSI. Una soglia di RSSI più selettiva, implica il formarsi di link più corti che quindi riducono il raggio di copertura dei nodi. L'incremento di traffico, dovuto al numero maggiore di Hop necessari per completare una rotta, porta al raggiungimento della capacità della rete tanto prima, quanto più grande è l'incremento. Vediamo ad esempio come una soglia di RSSI pari a -85 dBm causi lo spostamento della soglia di traffico da $\lambda = 16$ a $\lambda = 12$. D'altra parte la soglia di RSSI garantisce dei collegamenti con maggiore affidabilità che portano ad un miglioramento delle prestazioni

della PDR pari al 5% fintanto che la il traffico immesso è minore della capacità della rete. Dopo il superamento della soglia invece le prestazioni sono minori rispetto al caso senza soglia sull'RSSI. Sempre con la soglia di -85 dBm si è ottenuto un peggioramento del 5%. Con una soglia di RSSI pari a -75dBm, la soglia di traffico si è spostata da $\lambda = 16$ a $\lambda = 4$. Le prestazioni migliorano del 7% prima di raggiungere la capacità della rete e peggiorano del 30% dopo. Con una soglia di RSSI pari a -90dBm, la soglia di traffico non sembra spostarsi e le prestazioni sono simili alla curva di riferimento.
Si è visto come l'introduzione di un controllo sulla qualità dei link, migliori le prestazioni fino ad una certa quantità di traffico immessa. Se il tasso di traffico λ_1 introdotto dai nodi di applicazione è conosciuto a priori, è possibile inserire una soglia di RSSI, che abbia come valore della soglia di traffico λ_{thrs} lo stesso valore del tasso di traffico immesso ($\lambda_{thrs} = \lambda_1$). Ad esempio se il traffico λ è pari a 12, si andrà ad inserire una soglia di RSSI pari a -85dBm. Inoltre se l'applicazione montata nei sensori, genera un tasso di traffico variabile, ed è possibile risalire a questa quantità, si può pensare di adottare un settaggio adattivo che vari automaticamente la soglia di RSSI in funzione del tasso di traffico immesso, riuscendo sempre a sfruttare la massima capacità della rete.

5.7 Normalized Routing Load

Il *Normalized Routing Load* (NRL) è un parametro molto utile per misurare il traffico di *overhead* introdotto da un algoritmo di routing. É il numero di pacchetti di routing necessario affinchè un pacchetto di applicazione arrivi correttamente a destinazione. Il calcolo è stato effettuato sommando tutti i pacchetti di routing trasmessi nella rete e non solo quelli trasmessi dai nodi che sono stati attraversati dal percorso in esame.
In particolare questo parametro è molto utile in reti nelle quali è prevista anche la mobilità dei sensori per verificare le prestazioni di algoritmi che si adattano alla modifiche strutturali della rete. Tuttavia anche in reti statiche, come quella utilizzata negli esperimenti, da una misura del traffico di segnalazione necessario per mantenere aggiornate le tabelle di routing. Si ricorda che il periodo di aggiornamento con cui trasmette i messaggi di segnalazione l'algoritmo di routing è fisso e vale 30 secondi. Si è effettuato un esperimento con un tasso di traffico λ pari a 12. In Figura 5.27 si può vedere con varia il NRL in funzione della distanza dal sink (numero di Hop).

La configurazione con 30 nodi di applicazione presenta un valore massimo del NRL pari a 8 nei percorsi a distanza di 5 Hop. La configurazione con 30 nodi di applicazione e 30 nodi di inoltro presenta un valore massimo del NRL

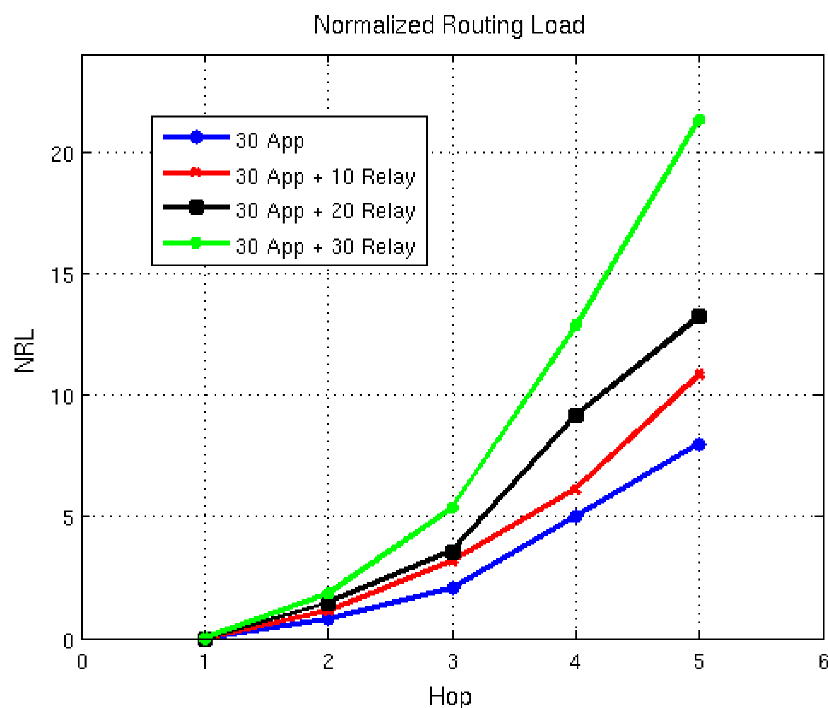


Figura 5.27: NRL in funzione del numero di Hop per 4 diverse configurazioni

pari a 21,5 sempre in corrispondenza dei nodi a distanza di 5 Hop. Ovviamente le configurazioni con i nodi di inoltro presentano dei valori più alti del NRL, in quanto vi è più traffico di segnalazione che circola nella rete. Dal confronto in letteratura con altri algoritmi di routing [17] si è visto che i valori questo parametro sono più elevati e soprattutto sono costanti. Questo è dovuto al fatto che nell'algoritmo testato i messaggi di segnalazione vengono trasmessi periodicamente. Ora in reti come quella testata (statiche), una segnalazione periodica è inutile in quanto la topologia della rete non cambia a meno di danneggiamenti nei sensori. Si propone quindi che questa caratteristica venga gestita con un algoritmo adattivo che si comporti a seconda se sono presenti nodi mobili oppure no. Ad esempio trasmettendo con periodi molto brevi all'accensione del nodo che ancora non conosce niente riguardo alla topologia della rete e successivamente aumentare progressivamente questo periodo, in modo da ridurre sempre più la segnalazione. Alla caduta di un link (mobilità) o alla comparsa di un nuovo nodo invece il periodo di aggiornamento tornerà al valore iniziale.

Conclusioni

Lo strumento sviluppato si inserisce bene nella richiesta di mercato attuale nella quale si prediligono sempre di più degli strumenti forniti di interfaccia grafica. Questa permette agli utenti di interagire con il sistema manipolando graficamente degli oggetti, svincolando dall'obbligo di imparare una serie di comandi da impartire da tastiera. Inoltre grazie all'interfaccia grafica vengono messi in luce in modo veloce eventuali problemi che si possono generare in una WSN. (Ad esempio perdite nel link, colli di bottiglia, ritardi elevati, ecc.)
La caratteristica principale dello strumento sviluppato è la sua portabilità. Esso è uno *sniffer* di traffico totalmente trasparente. Viene infatti caricato in una rete di sensori che esegue le sue normali applicazioni senza accorgersi che è presente uno strumento di questo tipo che esegue un monitoraggio costante del traffico in circolazione.
Inoltre lo strumento è stato creato per fare la valutazione delle prestazioni di un qualsiasi algoritmo di routing; è quindi indipendente dall'algoritmo utilizzato. Esso infatti ignora totalmente il traffico di segnalazione introdotto e si focalizza solo sul traffico applicativo interessato.
In seguito ai risultati ottenuti si sono evidenziate le seguenti possibile modifiche all'algoritmo testato:

- I messaggi di segnalazione sono fondamentali per un algoritmo di routing per costruire le tabelle di indirizzamento. In una rete statica, come quella utilizzata nei test, i nodi sono privi di mobilità. I collegamenti tra i nodi (vicini) rimangono fissi e anche la loro qualità (RSSI), a meno di qualche fluttuazione del canale. Si propone un algoritmo adattivo che trasmetta la segnalazione con periodi brevi quando avviene una modifica alla topologia della rete (nuovi nodi, mobilità) e invece incrementi il periodo di segnalazione man mano che topologia rimane fissa.

- Si è visto come a seconda della struttura della rete esista una qualche soglia di capacità che la rete può sopportare per mantenere le prestazioni

ad un alto livello. Si è visto inoltre che l'inserimento nell'algoritmo di una soglia di RSSI, porta a un miglioramento delle prestazioni a scapito di una minore capacità. Si propone quindi un algoritmo adattivo che a seconda del traffico immesso vari la soglia di RSSI, riuscendo a sfruttare sempre la massima capacità della rete.

Si evidenzia infine un miglioramento trovato agendo sulla struttura della rete: si è visto come l'introduzione di nodi di inoltro riesca a distribuire meglio il traffico, aumentando la capacità della rete che porta ad un miglioramento delle prestazioni sia in termini di PDR e ritardo end-to-end. Nei test eseguiti si è trovato che le prestazioni migliori si sono ottenute aggiungendo una quantità di nodi supplementari pari a un terzo dei nodi di applicazione. I dati a disposizione non sono sufficienti per affermare che questa quantità ha carattere generale, ma in ogni tipologia di rete possono essere aumentate le prestazione introducendo un numero aggiuntivo di nodi di inoltro.

Appendice A

Appendice

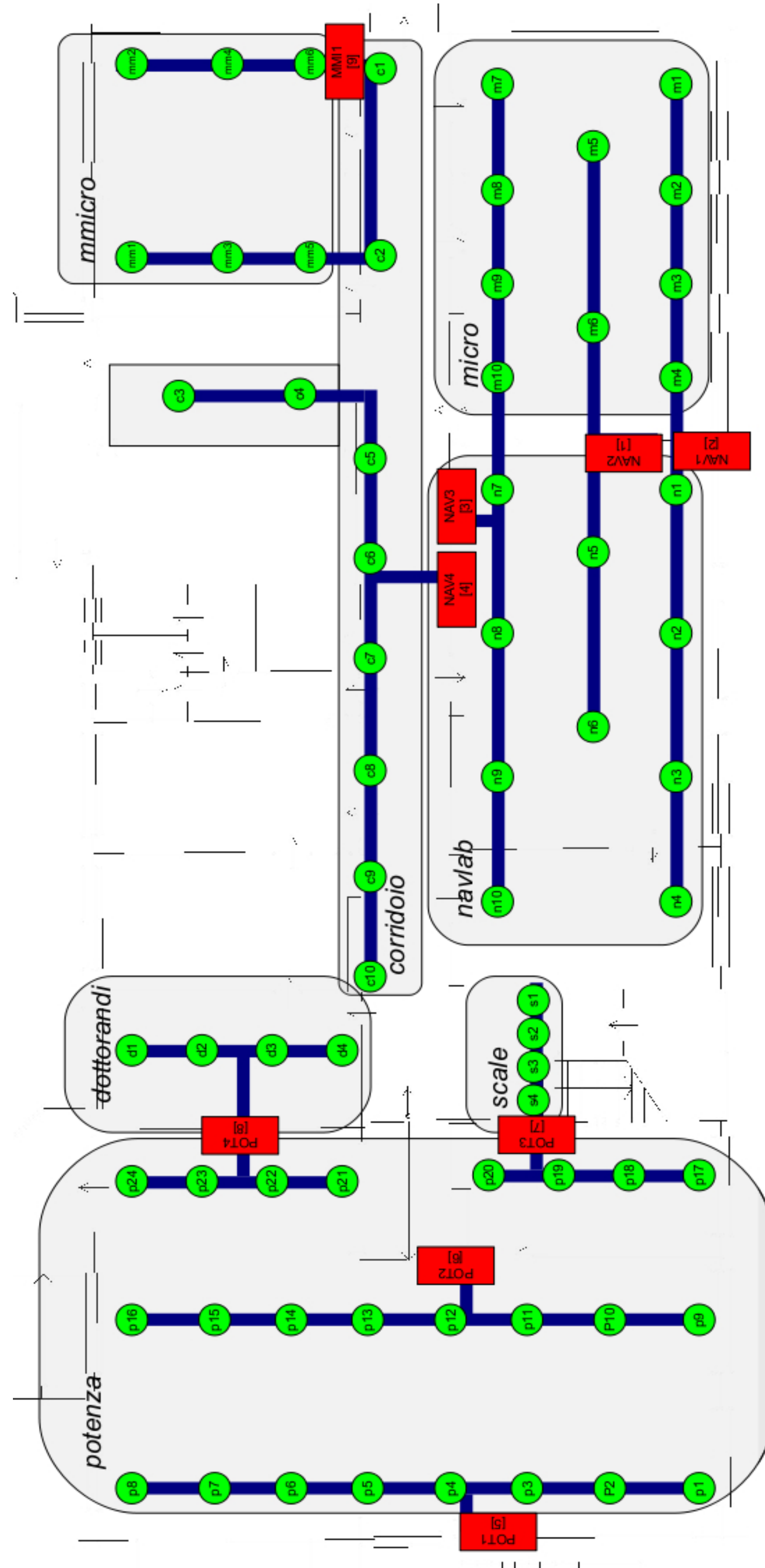


Figura A.1: Gruppi: MMICRO, MICRO, NAVLAB, CORRIDOIO, DOTTORANDI, SCALE, POTENZA

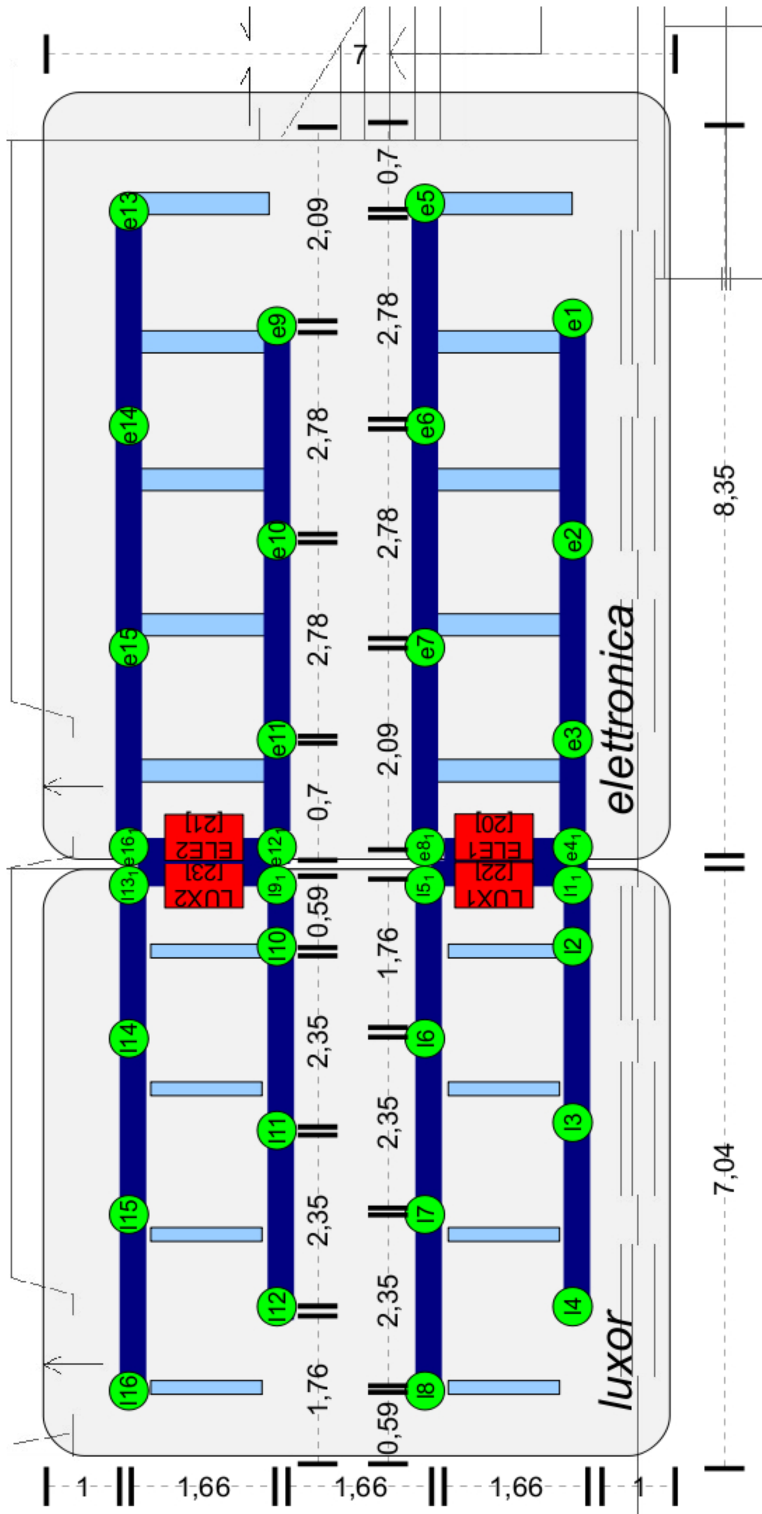


Figura A.2: Gruppi: ELTTRONICA,LUXOR

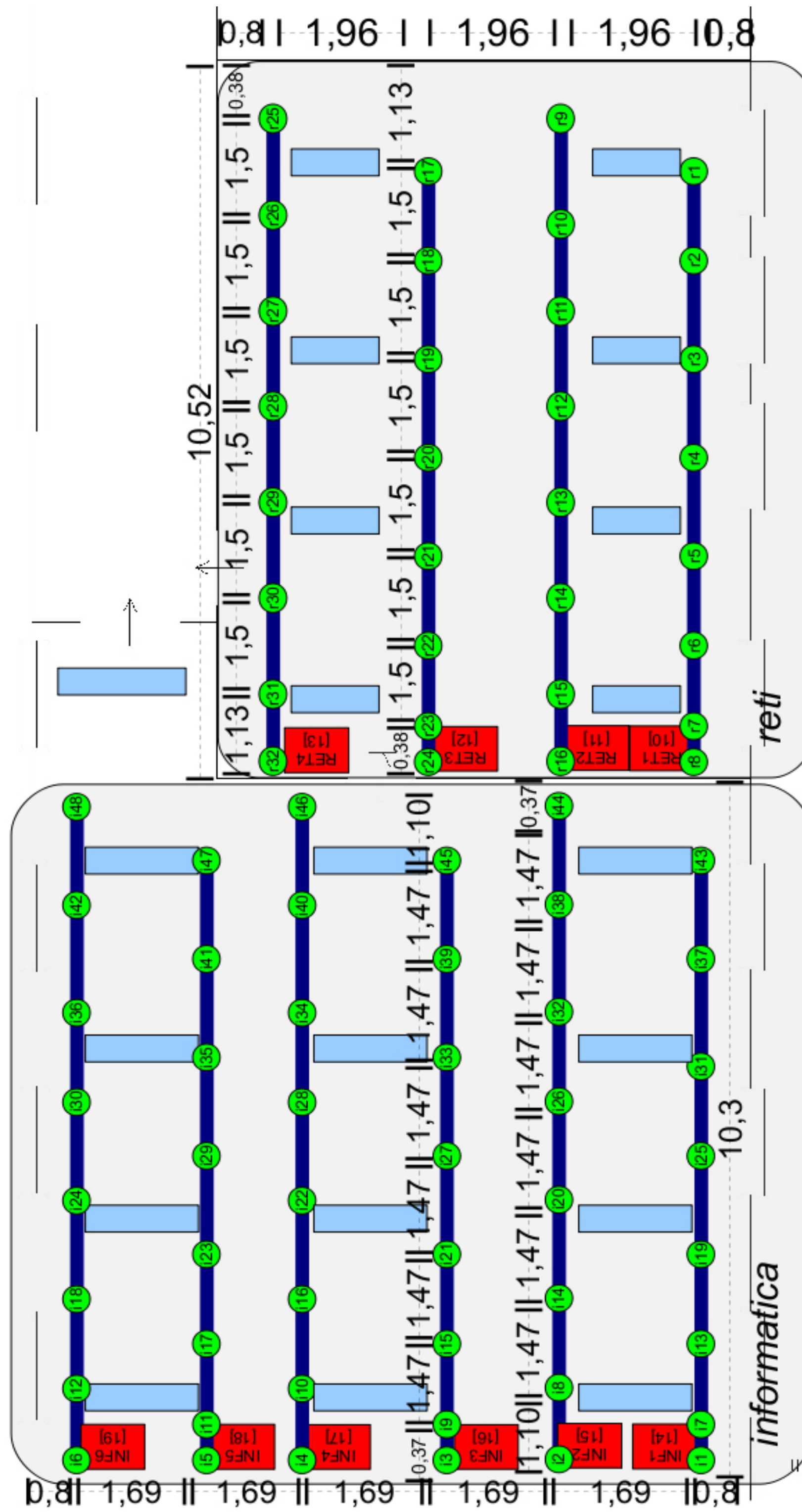


Figura A.3: Gruppi: RETI,INFORMATICA

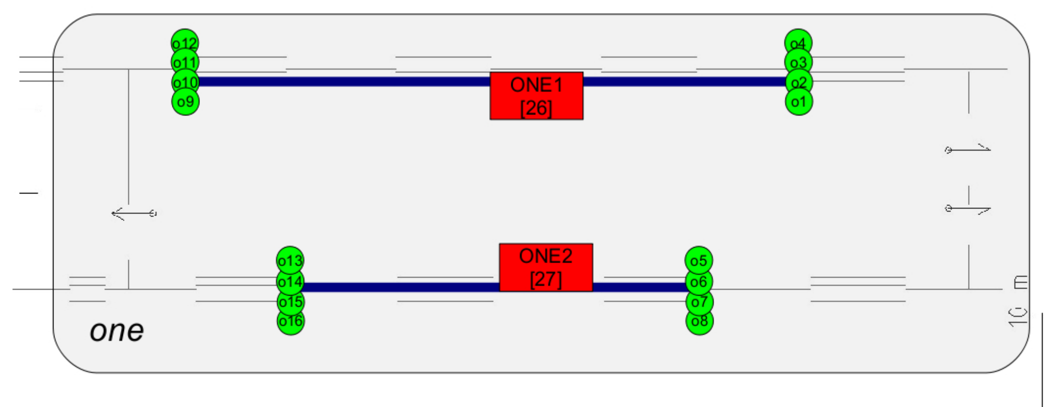


Figura A.4: Gruppo: ONE

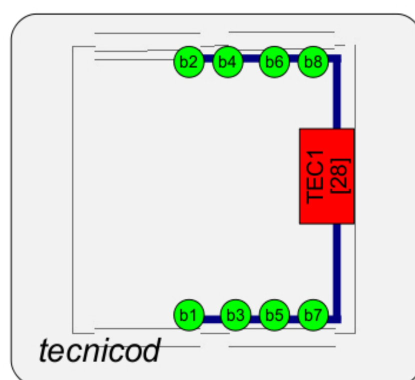


Figura A.5: Gruppo: TECNICOD

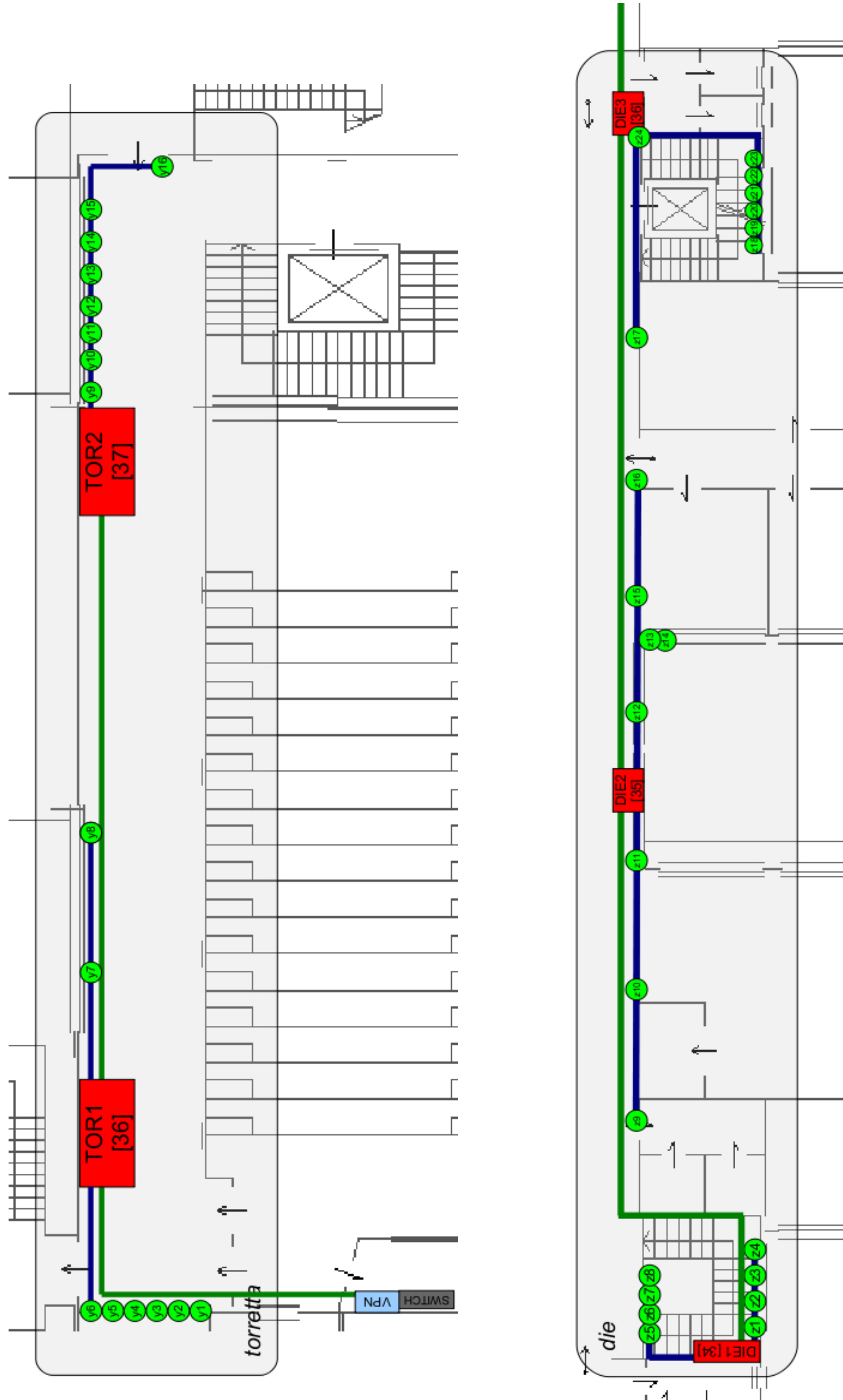


Figura A.6: Gruppi: TORRETTA,DIE

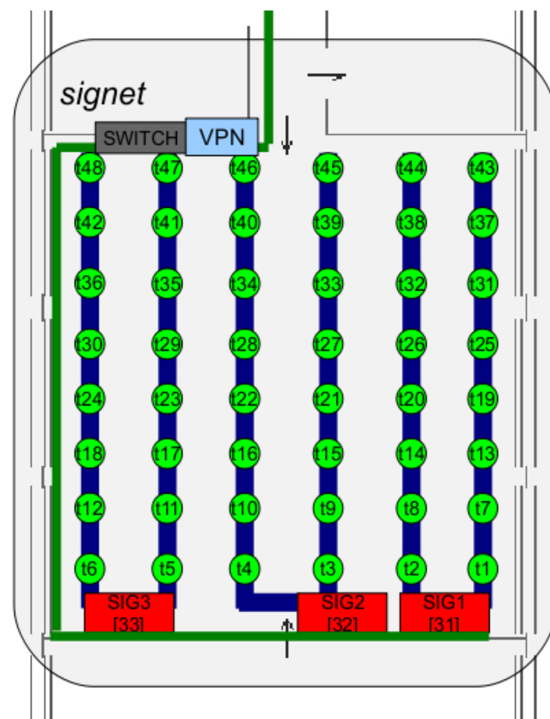


Figura A.7: Gruppo: SIGNET

Appendice B

Appendice

bunch	sensors
navlab	n1, n2, n3, n4, n5, n6, n7, n8, n9, n10
mmicro	mm1, mm2, mm3, mm4, mm5, mm6
micro	m1, m2, m3, m4, m5, m6, m7, m8, m9, m10
corridoio	c1, c2, c3, c4, c5, c6, c7, c8, c9, c10
dottorandi	d1, d2, d3, d4
scale	s1, s2, s3, s4
potenza	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15, p16, p17, 18, p19, p20, p21, p22, p23, p24
luxor	i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16
elettronica	e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13, e14, e15, e16
reti	r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, 18, r19, r20, r21, r22, r23, r24, r25
informatica	i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16, i17, i18, i19, i20, i21, i22, i23, i24, i25, i26, i27, i28, i29, i30, i31, i32, i33, i34, i35, i36, i37, i38, i39, i40, i41, i42, i43, i44, i45, i46, i47, i48
one	o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14, o15, o16
tecnicod	b1, b2, b3, b4, b5, b6, b7, b8
torretta	y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11, y12, y13, y14, y15, y16
die	z1, z2, z3, z4, z5, z6, z7, z8, z9, z10, z11, z12, z13, z14, z15, z16, z17, 18, z19, z20, z21, z22, z23, z24
signet	t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23, t24, t25, t26, t27, t28, t29, t30, t31, t32, t33, t34, t35, t36, t37, t38, t39, t40, t41, t42, t43, t44, t45, t46, t47, t48

Tabella B.1: Associazione dei nodi con il bunch di appartenenza

Bibliografia

[1] http://www.xbow.com/Products/productdetails.aspx?sid=252

[2] http://www.alix-board.de

[3] http://zachshelby.org/

[4] http://code.google.com/webtoolkit

[5] http://tomcat.apache.org/

[6] http://docs.tinyos.net/index.php/Main_Page

[7] http://en.wikipedia.org/wiki/IPv4

[8] http://en.wikipedia.org/wiki/Ipv6

[9] http://en.wikipedia.org/wiki/Mobile_ad-hoc_network

[10] Khaleel Ur Rahman Khan, A Venugopal Reddy, Rafi U Zaman, K. Aditya Reddy, T Sri Harsha "An Efficient DSDV Routing Protocol for Wireless Mobile Ad Hoc Networks and its Performance Comparison", Second UKSIM European Symposium on Computer Modeling and Simulation.

[11] Zhan Huawei, Zhou Yun, "Comparison and Analysis AODV and OLSR Routing Protocols in Ad Hoc Network", 2008.

[12] A. S. M. Ashique Mahmood, Mohammad Shahid Hossain, Faisal Aman Aziz, "Comparative performance analysis of DSR and AODV protocol based on Mobility factor",Proceedings of 11th International Conference on Computer and Information Technology (ICCIT 2008) 25-27 December, 2008, Khulna, Bangladesh.

[13] Vincent D. Park, M. Scott Corson, "A Performance Comparison of the Temporally-Ordered Routing Algorithm and Ideal Link-State Routing".

[14] Takashi Hamma, Takashi Katoh, Bhed Bahadur Bista, Toyoo Takata "An Efficient ZHLS Routing Protocol for Mobile Ad Hoc Networks", Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA'06).

[15] George Lukachan and Miguel A. Labrador, "SELAR: Scalable Energy-Efficient Location Aided Routing Protocol for Wireless Sensor Networks", Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04).

[16] JAMAL N. AL-KARAKI, AHMED E. KAMAL, "ROUTING TECHNIQUES IN WIRELESS SENSOR NETWORKS: A SURVEY", IEEE Wireless Communications • December 2004.

[17] Charles E. Perkins, Elizabeth M. Royer, Samir R. Das e Mahesh K. Marina, "Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks", IEEE Personal Communications • February 2001.