Università degli Studi di Padova
Facoltà di Ingegneria

Corso di Laurea Magistrale in
Ingegneria delle Telecomunicazioni

Tesi di laurea magistrale

Sperimentazione su reti mesh wireless

Costruzione della rete e confronto tra protocolli di routing

Candidato:
Francesco Zanella
Matricola 1020452

Relatore:
Andrea Zanella

Anno Accademico 2013–2014

A mio padre, mio punto di riferimento nella carriera e nella vita.

A mia madre, la più dolce e saggia confidente.

Ad Agnese, causa e scopo di tutto, la mia fonte inesauribile di energia ora e sempre.

Indice

1 Protocolli di routing analizzati **1**
1.1 OLSR 2
1.1.1 Descrizione del protocollo 2
1.1.2 Funzionalità aggiuntive 8
1.1.3 Metrica 9
1.1.4 Implementazione 12
1.2 BATMAN 12
1.2.1 Descrizione del protocollo 12
1.2.2 Metrica 16
1.2.3 Funzionalità aggiuntive 17
1.2.4 Implementazione 18

2 Testbed **21**
2.1 Concetti preliminari 21
2.1.1 Mesh network 21
2.1.2 Cognitive network 25
2.1.3 Android 27
2.2 Realizzazione di base del testbed 27
2.3 Hardware 27
2.3.1 ALIX 27
2.3.2 Nexus 7 27
2.4 Struttura del testbed 29
2.4.1 Server di controllo 31
2.5 Software 32
2.5.1 CogNet 33
2.5.2 SSH 33
2.5.3 NTP 34
2.5.4 Iptables 35
2.5.5 Iperf 35
2.5.6 Script Bash 36

3 Esperimenti **43**
3.1 Risultati e commenti 47

4 Conclusioni **59**
4.1 Lavori futuri 59

Elenco delle figure

1.1 Formato del pacchetto di OLSR . . . 3
1.2 Formato del MID-message di OLSR . . . 4
1.3 Formato del HELLO-message di OLSR . . . 5
1.4 Riduzione dei pacchetti di controllo di OLSR grazie al meccanismo degli MPR . . . 7
1.5 Formato del TC-message di OLSR . . . 7
1.6 Formato del HNA-message di OLSR . . . 9
1.7 Formato del pacchetto di BATMAN . . . 13
1.8 Formato dell'OGM di BATMAN . . . 14
1.9 Formato del messaggio HNA di BATMAN . . . 14
1.10 Ottimizzazioni multi-link . . . 18
1.11 Topologie di rete in cui si può utilizzare il *Network Coding*. . . . 18

2.1 Schema generale di una rete mesh. . . . 23
2.2 Esempio applicativo di una rete mesh. . . . 23
2.3 Diverse modalità di comunicazione in una rete *mesh*. . . . 24
2.4 Problema della condivisione della banda in una rete *multihop*. . . . 25
2.5 *Framework* cognitivo di un nodo di una *cognitive network*. . . . 26
2.6 ALIX 3d3. . . . 28
2.7 Nexus 7 e Atheros AR9280. . . . 30
2.8 Mappa del testbed. . . . 30
2.9 Situazione dell'occupazione dei canali Wi-Fi nella zona del testbed. . . 31
2.10 Tabella di *routing* del nodo .131 usando in ogni nodo la potenza di trasmissione massima. . . . 31
2.11 Tabella di *routing* del nodo .131 usando in ogni nodo una potenza di trasmissione ridotta. . . . 32
2.12 Schema completo del testbed. . . . 32
2.13 Output dello script `command_to_mesh` trasmettendo il comando `date`. 41

3.1 Goodput in modalità *routing* statico. . . . 48
3.2 Goodput con protocollo di *routing* BATMAN. . . . 48
3.3 Goodput con protocollo di *routing* OLSR libero. . . . 49
3.4 Goodput con protocollo di *routing* OLSR con rotte bloccate. . . . 49
3.5 Goodput con cambio di *bitrate*. . . . 50
3.6 Confronto tra i goodput medi. . . . 50
3.7 Andamento del goodput istantaneo per la modalità routing statico. . . 51
3.8 Andamento del goodput istantaneo con BATMAN. . . . 52
3.9 Andamento del goodput istantaneo con OLSR libero. . . . 53

3.10 Andamento del goodput istantaneo con OLSR con rotte bloccate. . . . 54
3.11 Analisi di come il cambio di rotta azzera il goodput con BATMAN. . . 56
3.12 Particolare di come il cambio di rotta azzera il goodput con BATMAN. 56
3.13 Analisi di come il cambio di rotta azzera il goodput con OLSR. 57
3.14 Particolare di come il cambio di rotta azzera il goodput con OLSR. . . 57
3.15 Alcuni parametri del TCP per OLSR FREE 2 hop. 58

Elenco delle tabelle

1.1 Valori possibili del campo `Willigness` 5
1.2 Valori proposti dal RFC di OLSR per le costanti 10
1.3 Valori proposti dal RFC di BATMAN per le costanti 16

2.1 caratteristiche di un ALIX 3d3. 28
2.2 Caratteristiche di un Nexus 7. 29
2.3 Potenze di trasmissione imposte ai nodi. 31
2.4 Parametri TCP e MAC osservabili e modificabili da CogNet per diversi dispositivi. 33

Elenco dei codici

2.1 Script `rc.local` di un nodo ALIX. 36
2.2 Script `batctl_script`. 39
2.3 Script `command_to_mesh`. 40
2.4 Script `commandNodes_Iperf`. 41

Sommario

In questa tesi si descriveranno le operazioni necessarie per la messa in opera e la gestione di una rete *cognitive mesh* Wi-Fi. In particolare si descriverà il lavoro pratico svolto per la costruzione della rete *cognitive mesh* Wi-Fi del dipartimento di Ingegneria dell'Informazione dell'Università degli studi di Padova, denominata AWMN (*Android Wireless Mesh Network*). Questa rete è infatti formata da dispositivi fissi con sistema operativo Linux e altri mobili con sistema operativo Android: si è sfruttato il fatto che entrambi si basano su un kernel Linux. Dopo aver descritto le operazioni necessarie per la costruzione della rete, si passerà ad illustrare i primi esperimenti eseguiti sulla stessa, che vertono sulla comparazione di diversi protocolli di *routing* quali OLSR, BATMAN e una modalità con *routing* statico.

Abstract

This thesis illustrates the operations needed to create and manage a cognitive Wi-Fi mesh network. In particular it describes the practical work done to build the cognitive Wi-Fi mesh network created at Department of Information Engineerig of University of Padua, called AWMN (Android Wireless Mesh Network). Indeed, this network is composed by fixed nodes with Linux operative system and by mobile nodes with Android operative system, exploiting the fact that they both lay on a Linux kernel. After this, the thesis illustrates the first experiments done above the AWMN, which focus on the comparation of some routing protocols like OLSR, BATMAN and a static routing mode.

Ringraziamenti

Desidero ringraziare il mio relatore prof. Andrea Zanella che mi ha accompagnato nella mia carriera universitaria sia per la tesi triennale che per questa tesi magistrale.
Ringrazio enormemente il PhD Matteo Danieletto con cui ho lavorato nello sviluppo di questa tesi, per tutti gli insegnamenti e i preziosi consigli.
Ringrazio i miei colleghi telecomunicazionisti che hanno reso più leggeri questi anni di carriera univeritaria.
Ringrazio i miei amici che mi hanno sostenuto nei momenti di sconforto.
Ringrazio la famiglia della mia fidanzata che mi ha accolto a braccia aperte e mi ha incoraggiato.
Ringrazio i miei genitori che hanno sempre creduto in me e mi hanno dato tutto l'amore possibile.
Ringrazio la mia fidanzata Agnese per avermi sorretto e sopportato durante questo duro periodo, a lei è dedicata questa tesi: che sia il punto d'inizio per la nostra vita futura insieme.

Padova, 14 Aprile 2014 F. Z.

Introduzione

In questa tesi si descriveranno le operazioni necessarie per la messa in opera e la gestione di una rete *cognitive mesh* Wi-Fi. In particolare si descriverà il lavoro pratico svolto per la costruzione della rete *cognitive mesh* Wi-Fi del dipartimento di Ingegneria dell'Informazione dell'Università degli studi di Padova, denominata AWMN (*Android Wireless Mesh Network*). Questa tesi descrive nei dettagli tutte le operazioni necessarie alla creazione di base della rete, allo scopo di fungere da *handbook* per l'utilizzo di AWMN.

Nella seconda parte della tesi verranno esposti i primi esperimenti eseguiti sul *testbed* che vertono sulla comparazione di diversi protocolli di *routing* quali OLSR, BATMAN e una modalità con *routing* statico.

È importante sottolineare come i non molti risultati presenti in letteratura su questi argomenti, si basino su simulazioni. Una realizzazione pratica offre sicuramente aspetti e problematiche nuovi che vanno analizzati e superati.

La tesi è strutturata nel seguente modo:

il primo capitolo offre una descrizione dettagliata dei protocolli di *routing* analizzati;

il secondo capitolo descrive il *testbed* e le operazioni necessarie per la creazione della rete *cognitive mesh* Wi-Fi;

Nel terzo capitolo si illustreranno gli esperimenti effettuati, i risultati e la discussione di questi ultimi;

Nel terzo capitolo si trarranno le conclusioni e si indicheranno i possibili lavori futuri correlati a questo.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Capitolo 1

Protocolli di routing analizzati

Un protocollo di *routing* (instradamento) è un protocollo relativo al livello rete del modello ISO-OSI che regola lo scambio di informazioni tra i *router* al fine di costruire delle tabelle di *routing* che permettano il corretto instradamento dei pacchetti verso la giusta destinazione.

Le due tipologie classiche di protocolli di *routing* sono:

***Link State*:** i protocolli *link state* (*routing* basato sullo stato del collegamento) utilizzano il concetto di mappa distribuita, ossia un elenco di tutti i link della rete con relativo costo; tutti i *router* hanno una copia di tale mappa che viene aggiornata in continuazione; ogni nodo periodicamente usa la tecnica del *flooding*, chiamata in questo caso *Link State Broadcast*, per inviare tramite tutti i suoi link diretti un messaggio *Link State Packet* (LSP), contenente tutte le informazioni sui link attivi tra il mittente del messaggio e i suoi vicini; alla ricezione di tale messaggio, ogni *router* aggiorna la propria *routing table* (se il mesaggio porta informazioni nuove) e lo rispedisce (se non l'aveva già fatto con lo stesso messaggio) a tutti i suoi vicini diretti (tranne il mittente); in questo modo tutti i *router* hanno sempre a disposizione il grafo aggiornato dell'intera rete ed il percorso più conveniente si ottiene con un algoritmo *shortest path*, che in genere è l'algoritmo di Dijkstra.

***Distance Vector*:** i protocolli *distance vector* si basano sull'algoritmo di Bellmann-Ford; ogni nodo mantiene un database con le distanze minime tra sé stesso e tutte le possibili destinazioni; a intervalli regolari invia ai nodi adiacenti un *distance vector*, che è un insieme di coppie indirizzo-distanza, chiamate annunci; la distanza è espressa come numero di hop o mediante criteri più generali che tengono conto di velocità, carico e affidabilità dei collegamenti; a partire da tali dati, utilizzando l'algoritmo di Bellman-Ford, il *router* costruisce una tabella che associa ad ogni destinazione conosciuta la distanza che lo separa da essa e il primo passo del percorso calcolato; quando riceve un *distance vector*, un nodo può usare le informazioni contenute per ricalcolare la sua tabella di *routing*; a differenza dei protocolli *link state* in questo caso il messaggio non viene reinoltrato.

I protocolli di *routing* tradizionali vanno modificati e adattati nel caso di reti *ad-hoc* o *mesh* (di cui si parlerà ampiamente in seguito), formate da un insieme di nodi che possono comunicare direttamente l'uno con l'altro, purché siano l'uno nel *range* di trasmissione dell'altro. Le difficoltà più grosse derivano dal fatto che i nodi sono potenzialmente mobili, quindi è necessario predisporre dei metodi per

aggiungere o rimuovere elementi dalla rete in qualsiasi momento, e altri per cambiare dinamicamente i percorsi migliori in base alla situazione. É necessario gestire rotture dei link estremamente più frequenti, e un aggiornamento dei percorsi migliori molto più rapido. In questo caso è possibile una diversa classificazione dei protocolli di *routing* in base al momento in cui avviene l'elaborazione dei cammini:

protocolli proattivi: le rotte vengono calcolate a priori, controllando tutti i possibili percorsi senza sapere se poi verranno effettivamente utilizzati;

protocolli reattivi: le rotte vengono calcolate solo se richieste al momento dell'effettivo instradamento del pacchetto (*on-demand*).

Verranno ora descritti i protocolli di *routing* per reti *mesh* Wi-Fi che si sono testati ed analizzati nel *testbed*.

1.1 OLSR

OLSR (Optimized Link State Routing) è un protocollo per MANET (Mobile Adhoc NETworks) proposto da T. Clausen e P. Jaquet nel RFC 3626 [1] pubblicato nell'Ottobre 2003.

1.1.1 Descrizione del protocollo

OLSR è un protocollo proattivo, cioè che scambia informazioni sull'intera topologia della rete regolarmente, permettendo così di avere immediatamente a disposizione la rotta per un certo nodo quando necessaria. É un ottimizzazione del classico protocollo *Link State*: il concetto chiave è l'uso dei *MultiPoint Relays* (MPR) cioè dei nodi eletti per inoltrare i messaggi in *broadcast* durante il processo di *flooding*. Questa tecnica riduce l'*overhead* rispetto ad un classico *flooding* in cui ogni nodo ritrasmette ogni messaggio. Un MPR è responsabile di comunicare a tutta la rete la rotta per raggiungere i nodi che lo hanno scelto come tale. Viene trasmessa quindi solo una parziale informazione dei *Link State*, che è però progettata per essere sufficiente a fornire rotte ottime.

Il protocollo è particolarmente efficace in reti larghe e dense, ed è idoneo a topologie che cambiano nel tempo. Per adattarsi a reti che cambiano molto rapidamente, OLSR può rendersi più reattivo riducendo l'intervallo massimo di tempo tra due trasmissioni di pacchetti di controllo.

Il protocollo non necessita di una trasmissione affidabile dei pacchetti di controllo: la periodicità con cui vengono inviati rende sostenibile una ragionevole perdita di pacchetti, situazione molto frequente in una rete radio. OLSR non necessita nemmeno di un ordinamento dei pacchetti ricevuti, infatti un *sequence number* presente in ogni pacchetto rende il ricevitore capace di identificare l'informazione più recente. Per questi due motivi i pacchetti di controllo vengono trasmessi su UDP e nessuna modifica alla struttura dei pacchetti IP si rende necessaria.

Per descrivere il protocollo è utile suddividerlo in diverse funzionalità che si andranno ad analizzare, distinguendo le funzionalità principali che permettono la corretta costruzione delle tabelle di *routing* e sono quindi indispensabili per il funzionamento, dalle funzionalità aggiuntive che si rendono utili solo in casi specifici. OLSR definisce alcune funzionalità aggiuntive ma è costruito in modo da poter essere integrato con qualsiasi altro *plug-in* esterno. Si analizzeranno ora le funzionalità principali.

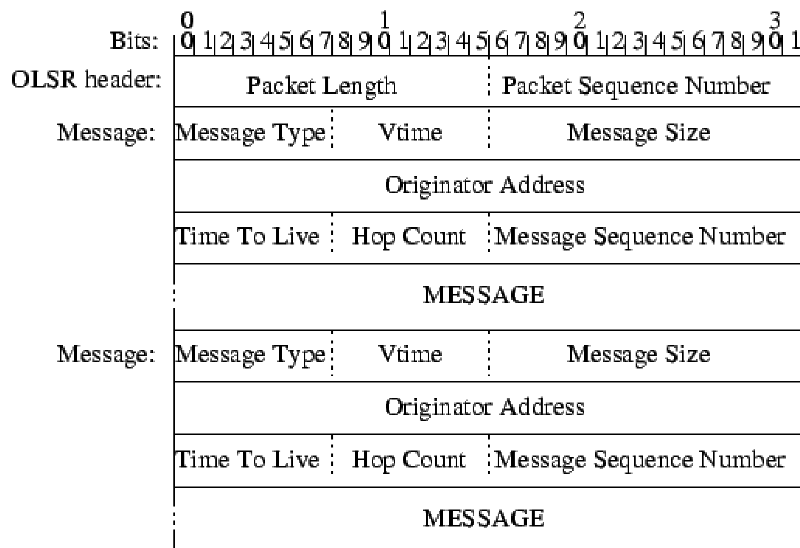


Figura 1.1: Formato del pacchetto di OLSR

Formato del pacchetto

OLSR definisce un formato unificato per tutti i tipi di pacchetti di controllo, che facilita l'eventuale estensibilità senza violare la retro-compatibilità.

In Figura 1.1 si può osservare il formato del pacchetto OLSR. Ogni pacchetto è costituito da un *header* comune e incapsula uno o più messaggi, ognuno costituito da un proprio *header* e dal messaggio vero e proprio. Ogni messaggio apparterrà ad un ben definito "tipo", l'inoltro è concesso anche se un certo nodo non conosce il tipo di messaggio ricevuto. Un messaggio può essere trasmesso in *flooding* all'intera rete oppure si può limitare il diametro (in termini di hop) a partire dal nodo origine tramite il campo `TTL` (Time To Live). I pacchetti o i messaggi duplicati possono essere scartati localmente riconoscendoli tramite i campi `Packet Sequence Number` e `Message Sequence Number`. Un ricevitore può sempre ottenere la distanza da cui ha avuto origine il messaggio leggendo il campo `Hop Count`. Il campo `Vtime` indica per quanto tempo il messaggio deve essere ritenuto valido dopo la ricezione. OLSR definisce alcuni tipi di messaggi essenziali per il funzionamento del protocollo:

HELLO-message: utilizzati per il *link sensing*, la *neighbor detection* e il *MPR signaling*;

TC-message: (*Topology Control*) utilizzati per la trasmissione in *flooding* delle informazioni sui *link state*;

MID-message: (*Multiple Interface Declaration*) utilizzati per dichiarare interfacce multiple su un solo nodo.

OLSR cerca di evitare la naturale sincronizzazione dell'invio dei pacchetti di controllo da parte di nodi vicini per ridurre le collisioni e la conseguente perdita di pacchetti. Per far questo introduce un *jitter*, cioè un intervallo di tempo casuale che un nodo deve attendere prima della trasmissione.

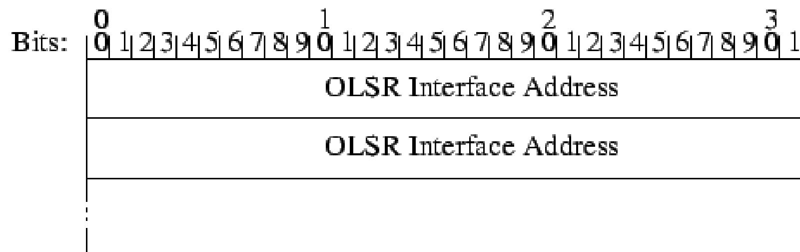


Figura 1.2: Formato del MID-message di OLSR

Patrimonio informativo di un nodo

Ogni nodo deve mantenere aggiornato il proprio patrimonio informativo, contenente i dati che ha raccolto o elaborato riguardanti *link*, nodi vicini e topologia della rete:

link set: insieme dei *link* che collegano ogni sua interfaccia (su cui sta operando OLSR) ad ogni nodo vicino;

neighbor set: insieme dei vicini, cioè i nodi direttamente raggiungibili con 1 hop;

2-hop neighbor set: insieme dei nodi raggiungibili con 2 hop; questa informazione serve per l'elezione del MPR;

MPR set: insieme dei nodi eletti come propri MPR;

MPR selector set: insieme dei nodi che hanno eletto il nodo in analisi come MPR;

topology information base: tutti i dati raccolti tramite i TC-message che vengono utilizzati per il calcolo delle tabelle di *routing*.

Indirizzo principale e interfacce multiple

Per i nodi con una sola interfaccia con OLSR in esecuzione, l'indirizzo principale, cioè quello che identifica il nodo, è ovviamente l'indirizzo di quella interfaccia. Per i nodi con interfacce multiple con OLSR in esecuzione, il nodo deve scegliere un indirizzo principale tra quelli delle sue interfacce e deve comunicarlo all'intera rete attraverso i MID-message. I MID-message vengono trasmessi periodicamente ogni `MID_INTERVAL` in *flooding* tramite il meccanismo degli MPR, e contengono, come mostrato in Figura 1.2, semplicemente l'elenco degli indirizzi delle interfacce aggiuntive di un nodo che devono essere associate all'indirizzo principale (che si può già leggere nel campo `Originator Address`).

Formato e generazione degli HELLO-message

Gli HELLO-message servono a popolare i patrimoni informativi riguardanti i link, i vicini e gli MPR; vengono trasmessi da ogni interfaccia a tutti i vicini diretti (1 hop) e non devono essere mai inoltrati.

Il formato del HELLO-message è illustrato in Figura 1.3. `HTime` è l'intervallo di tempo tra un HELLO-message e il successivo, utilizzato dal nodo che ha generato il messaggio su quella specifica interfaccia; questa informazione viene usata da alcune funzionalità aggiuntive come l'*advanced link sensing*. Un parametro molto interessante nell'ambito di questa tesi e dei lavori futuri sul testbed è `Willingness` cioè l'idoneità di

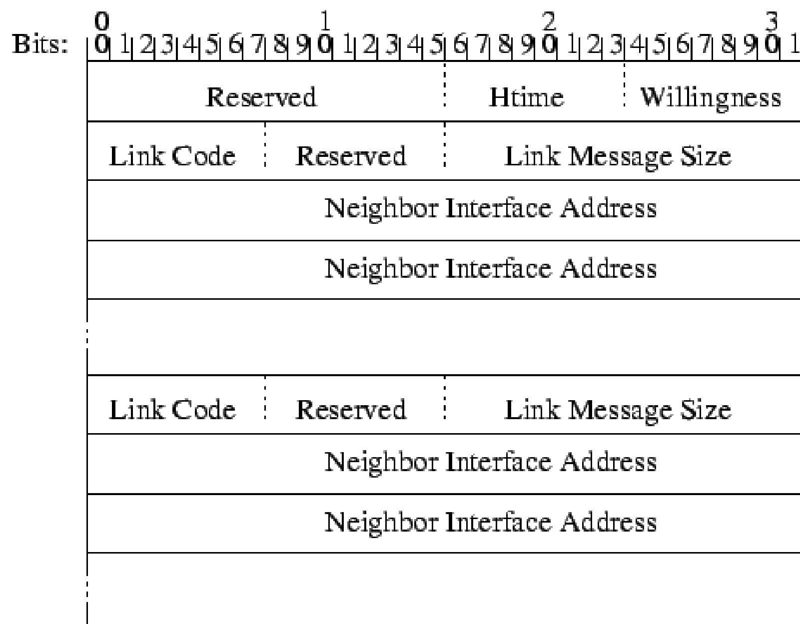


Figura 1.3: Formato del HELLO-message di OLSR

Tabella 1.1: Valori possibili del campo `Willigness`

`Willingness`	valore del campo	significato
`WILL_NEVER`	0	non usare mai come MPR
`WILL_LOW`	1	`Willingness` bassa
`WILL_DEFAULT`	3	`Willingness` media
`WILL_HIGH`	6	`Willingness` alta
`WILL_ALWAYS`	7	usare sempre come MPR

un nodo a inoltrare pacchetti in *flooding*; questo parametro viene utilizzato nell'elezione degli MPR e può essere configurato a priori su un nodo, oppure si potrebbe pensare di cambiarlo *on the fly* ad esempio in base alla mobilità del nodo: se il nodo è molto mobile (come un tablet che può riconoscere il suo stato di mobilità attraverso gli accellerometri che monta a bordo) la *willingness* dovrà essere bassa, se invece un nodo è fisso e può far parte della *backbone* della rete setterà la sua *willingness* ad un valore alto. I valori possibili per il campo `Willingness` sono elencati in Tabella 1.1: si tratta di poche costanti; al fine di cambiarlo *on the fly* in modo automatico, sarebbe utile che i valori spaziassero su tutto l'intervallo possibile $0 \div 255$. Dopo l'*header* comune vengono elencati tutti i `Neighbor Interface Address`, cioè gli indirizzi di tutte le interfacce dei vicini direttamente raggiungibili, raggruppati a seconda del `Link Code` che contiene informazioni generiche sul link; ogni `Neighbor Interface Address` contiene oltre all'indirizzo anche informazioni più specifiche sul tipo di vicino nel campo `Neighbor Type`(ad esempio se è un MPR) e sul tipo di link nel campo `Link Type`(simmetrico, asimmetrico, caduto, ...).

La regola imposta è che ogni nodo e ogni link deve essere citato in un HELLO-message almeno una volta ogni `REFRESH_INTERVAL`, e, per tenere traccia dei cambiamen-

ti veloci della topologia della rete, ogni interfaccia deve trasmette un HELLO-message ogni `HELLO_INTERVAL`.

Link Sensing

Il *link sensing* è la procedura che riempie la parte di patrimonio informativo di un nodo che riguarda i link cioè il *link set*, e per far ciò utilizza il meccanismo dello scambio di HELLO-message. Vengono salvate le informazioni ricavabili dai campi del HELLO-message riguardanti i link (`Link Code` e `Link Type`). Vengono ritenuti validi per il calcolo delle tabelle di *routing* solo i link simmetrici, cioè bidirezionali, su cui è stato sia trasmesso che ricevuto un HELLO-message.

Neighbor Detection

La *Neighbor Detection* è la procedura che riempie la parte di patrimonio informativo di un nodo che riguarda i vicini, e per far ciò utilizza il meccanismo dello scambio di HELLO-message. La *Neighbor Detection* deve occuparsi di creare e mantenere aggiornati il *neighbor set*, il *2-hop neighbor set*, l'*MPR set* e l'*MPR selector set*.

Per creare e mantenere aggiornato il *neighbor set*, la procedura utilizza ovviamente come input il *link set*: un nodo è un vicino se esiste almeno un link tra i due nodi. Il *neighbor set* deve essere aggiornato ogni volta che un link viene creato, modificato o rimosso dal *link set*, quindi alla ricezione di ogni HELLO-message un nodo deve prima aggiornare il *link set* e poi di conseguenza aggiornare il *neighbor set*.

Per creare e mantenere aggiornato il *2-hop neighbor set* si devono processare gli HELLO-message ricevuti, che contengono tutti i nodi con cui può comunicare il nodo che ha trasmesso l'HELLO-message (che dista 1-hop).

Per creare e mantenere aggiornato l'*MPR set* vengono analizzati il *neighbor set* e il *2-hop neighbor set*. Ogni nodo seleziona autonomamente i suoi MPR tra i suoi vicini diretti. La regola per il calcolo dell'*MPR set* è che ogni nodo attraverso i suoi MPR (eletti tra i suoi vicini a 1 hop) deve poter raggiungere ogni suo vicino a 2 hop di distanza. Non è essenziale che l'*MPR set* sia minimo (composto dal minor numero di nodi possibile), ma è indispensabile che ogni nodo nel *2-hop neighbor set* sia raggiungibile. Nel calcolo si deve tener conto della *willingness*. L'*MPR set* deve essere aggiornato ad ogni cambiamento del *neighbor set* o del *2-hop neighbor set*. Come già accennato il meccanismo degli MPR serve a ridurre le ritrasmissioni dei TC-message diminuendo così l'*overhead* sul canale; il beneficio che si ottiene è molto chiarificato dalla Figura 1.4.

Per creare e mantenere aggiornato l'*MPR selector set* si devono processare gli HELLO-message ricevuti e in particolare il campo `Neighbor Type` all'interno del campo `Neighbor Interface Address`: se questo contiene il valore `MPR_NEIGH` allora il nodo origine dell'HELLO-message ha selezionato il nodo che riceve l'HELLO-message come MPR.

Topology Discovery

Il *Link Sensing* e la *Neighbor Detection* offrono ad ogni nodo una lista di vicini; queste informazioni vanno trasmesse in *flooding* a tutta la rete tramite il meccanismo degli MPR per permettere ad ogni nodo di costruirsi la topologia della rete sulla quale calcolare le tabelle di *routing*. Le informazioni da distribuire in *flooding* vengono inserite nei TC-message, il cui formato è illustrato in Figura 1.5.

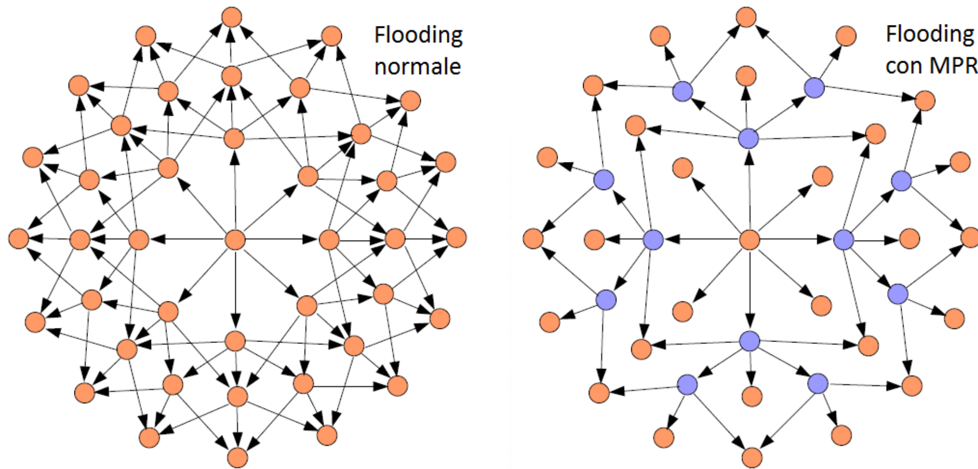


Figura 1.4: Riduzione dei pacchetti di controllo di OLSR grazie al meccanismo degli MPR

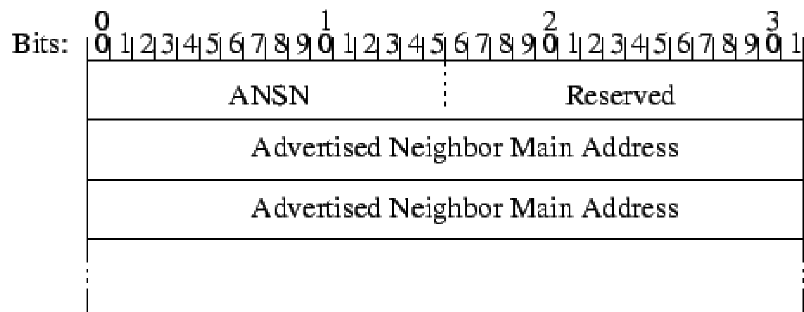


Figura 1.5: Formato del TC-message di OLSR

Il campo `ANSN` cioè *Advertised Neighbor Sequence Number* serve al nodo che riceve il TC-message per poter ordinare temporalmente le informazioni ricevute da tutti i nodi, in modo da riconoscere la più recente. La lista degli `Advertised Neighbor Main Address` è costituita da tutti gli indirizzi principali dei nodi vicini del nodo che ha originato il messaggio.

La lista degli indirizzi può essere anche parziale (ad esempio a causa di limitazioni della dimensione dei pacchetti imposte dalla rete), ma la lista deve essere completata entro un tempo `TC_INTERVAL`. Un nodo può trasmettere TC-message addizionali per incrementare la sua reattività alle cadute dei link. Se viene riscontrata una caduta di un link, deve essere trasmesso un TC-message immediato. I TC-message vengono trasmessi e inoltrati in broadcast dagli MPR, e un nodo che riceve un TC-message deve aggiornare di conseguenza la sua *topology information base*.

Calcolo delle routing table

Ogni nodo mantiene una *routing table* che gli permette di instradare i pacchetti nella direzione migliore per raggiungere il destinatario. Il calcolo del percorso ottimo viene effettuato sulla base del grafo di rete costruito grazie alle informazioni scambiate con gli altri nodi tramite gli HELLO-message, i TC-message e i MID-message. La *routing table* va quindi aggiornata ogni volta che cambiano il *link set*, il *neighbor set* il *2-hop neighbor set*, il *topology set* o le informazioni sulle interfacce multiple.

Nella versione più basilare di OLSR viene utilizzato un algoritmo per il calcolo del percorso che usa come metrica solo il numero di hop per raggiungere la destinazione. Già nella prima stesura del protocollo però, viene aggiunta la possibilità di utilizzare una metrica più complessa che si basa sulla qualità dei link che verrà esaminata tra le funzionalità aggiuntive.

1.1.2 Funzionalità aggiuntive

Si descriveranno ora le funzionalità aggiuntive presentate nel RFC di OLSR, cioè quelle funzionalità da attivare solo in casi specifici. OLSR, e in particolare la sua implementazione OLSRd, è costruito in modo tale da permettere facili modifiche o creazioni di nuove funzionalità.

Interfacce non-OLSR

Alcuni nodi con interfacce multiple potrebbero non utilizzare OLSR su tutte le interfacce; queste interfacce non-OLSR potrebbero essere connesse ad altri nodi non-OLSR o ad altre reti esterne. Per fornire connettività a queste interfacce non-OLSR e a tutto ciò che si interfaccia con esse, il nodo in questione deve trasmettere periodicamente in *flooding* informazioni sulle reti esterne, e lo può fare attraverso un quarto tipo di messaggio: l'HNA-message (Host and Network Association) il cui formato è illustrato in Figura 1.6. Ogni nodo mantiene le informazioni ricevute tramite gli HNA-message nel proprio *association set*. Gli HNA-message si possono considerare una versione più generalizzata dei TC-message che non avevano bisogno di annunciare la *netmask* in quanto annunciavano singoli *host*. Una importante differenza tra TC-message e HNA-message è che le informazioni portate dai primi possono avere effetto di cancellazione o sostituzione di informazioni precedenti, mentre quelle portate dai secondi vengono cancellate solo per scadenza della validità. Un nodo con interfacce non-OLSR deve trasmettere un HNA-message ogni `HNA_interval`.

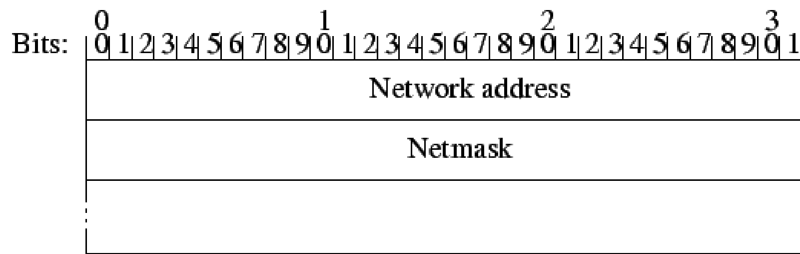


Figura 1.6: Formato del HNA-message di OLSR

Notifiche dal Link Layer

OLSR non è progettato in modo da richiedere o aspettare informazioni specifiche dal *Link Layer* (livello sottostante a quello di *routing* nella pila ISO-OSI che si occupa dei link con i vicini diretti), ma se esso può fornire indicazioni sulla rottura di alcuni link, OLSR ne può tenere conto. Queste informazioni sono usate insieme agli HELLO-message per aggiornare il patrimonio informativo dei vicini e degli MPR.

Ridondanza delle informazioni sulla topologia

Per fornire ridondanza alle informazioni sulla topologia trasmesse da un nodo, quest'ultimo può inserire nei TC-message informazioni di nodi che non sono nel suo *MPR selector set* (i nodi che lo hanno scelto come MPR). La ridondanza è controllata dalla variabile `TC_REDUNDANCY`, in particolare se questa è pari a 0 vengono trasmesse le sole informazioni dei nodi nell'*MPR selector set* (no ridondanza), se pari a 1 vengono trasmesse le informazioni dei nodi nell'*MPR selector set* e di quelli nel *MPR set*, se pari a 2 vengono trasmesse le informazioni di tutti i vicini.

Ridondanza degli MPR

Normalmente l'*MPR set* viene scelto in modo che sia più piccolo possibile per ridurre al massimo l'*overhead* del protocollo sul canale (ma sempre in modo che ogni nodo a 2 hop di distanza sia raggiungibile tramite almeno un MPR). Per assicurare la raggiungibilità di un nodo importante anche in caso di rottura di link, è possibile aumentare il numero di MPR eletti a discapito dell'*overhead*. La variabile `MPR_COVERAGE` indica il numero minimo di MPR tramite cui si deve poter raggiungere un nodo a 2 hop di distanza.

Valori proposti

I valori proposti dal RFC per le costanti più importanti sono elencati in Tabella 1.2.

1.1.3 Metrica

La metrica classica, utilizzata nella versione più basilare di OLSR, prevede che si contino semplicemente gli hop, cioè il numero di link che il pacchetto deve attraversare per arrivare a destinazione. Si può dire quindi che ad ogni link è associato un costo pari a 1, e ogni link che il pacchetto deve attraversare fa crescere la metrica totale di 1. Il percorso ottimo sarà in questo caso quello con la metrica più bassa, cioè quello in cui il pacchetto deve attraversare meno link.

Tabella 1.2: Valori proposti dal RFC di OLSR per le costanti

Costante	valore proposto
HELLO_INTERVAL	2 s
REFRESH_INTERVAL	2 s
TC_INTERVAL	5 s
MID_INTERVAL	TC_INTERVAL
HNA_INTERVAL	TC_INTERVAL

Esistono metriche più evolute che introducono il concetto di qualità del link: ad ogni link sarà associato un valore che rappresenta la sua capacità di trasmettere correttamente il pacchetto, più alto è il valore, più alta sarà la qualità. In questo caso il percorso ottimo sarà quello con la metrica totale più alta. Se la somma delle qualità di due link che formano un percorso verso una destinazione è più alta della qualità del link diretto, il protocollo può decidere di scegliere il percorso più lungo.

Nelle funzionalità aggiuntive presentate nel RFC di OLSR è già presente una sorta di miglioria alla semplice metrica che conta il numero di hop, chiamata *Link Hysteresis*. Un *plug-in* creato successivamente e inserito come default dalla versione 0.4.8 di OLSRd porta ad utilizzare come metrica evoluta l'ETX (Expected Trasmission Count).

Link Hysteresis

La funzionalità opzionale *Link Hysteresis* permette di migliorare la metrica usata nell'algoritmo che calcola il percorso ottimo sul grafo della rete. Introduce il concetto di qualità dei link però questa non è usata come metrica, bensì come fattore che indica se un link debba o meno essere usato; la metrica resta quindi di fatto il numero di hop. Questa funzione è progettata per evitare che un link buono venga scartato troppo presto dopo un *burst* di errori, e che un link transitorio (che compare per scomparire poco dopo causa mobilità ad esempio) venga scelto. La qualità in questo caso è determinata da due fattori: la "bontà" del link in sè, e il tempo consecutivo in cui il link è stato attivo (transitano HELLO-message). Si introduce una sorta di isteresi con due soglie `HYST_THRESHOLD_LOW` minore di `HYST_THRESHOLD_HIGH`: un link per essere ritenuto valido deve superare con la sua qualità `HYST_THRESHOLD_HIGH`, mentre per essere ritenuto non più valido deve scendere ad una qualità inferiore a `HYST_THRESHOLD_LOW`.

Nel RFC è suggerito che la qualità del link può corrispondere ad una misura diretta del SNR (dopo una normalizzazione) se disponibile dal *link layer*, altrimenti viene proposto un algoritmo che effettua la media mobile del *successful transmission rate*:

- se un pacchetto OLSR viene ricevuto correttamente:

$$\texttt{L_link_quality} = (1 - \texttt{HYST_SCALING}) \times \texttt{L_link_quality} + \texttt{HYST_SCALING} \tag{1.1}$$

- se un pacchetto OLSR viene perso:

$$\texttt{L_link_quality} = (1 - \texttt{HYST_SCALING}) \times \texttt{L_link_quality} \tag{1.2}$$

La perdita di pacchetti può essere notificata osservando i *sequence number* mancanti o i lunghi periodi di silenzio (nessun pacchetto ricevuto entro un `HELLO_INTERVAL`).

ETX

Dalla versione 0.4.8, OLSRd offre un'implementazione della metrica ETX. L'RFC usa come metrica il numero di hop per raggiungere la destinazione, anche utilizzando l'opzione *Link Hysteresis*, quindi preferisce in qualsiasi caso un singolo link di bassa qualità a un percorso formato da due link di alta qualità. La metrica ETX risolve questo problema.

Un nodo riceve periodicamente (ogni 2 secondi di default) gli HELLO-message dai suoi vicini, ciò può essere sfruttato per misurare la percentuale di pacchetti persi su un determinato link. In ambito wireless non è rara la perdita di un pacchetto causata dal "cattivo" canale o da una collisione, quindi considerando ad esempio una finestra pari a 10, se si ricevono 7 pacchetti e ne vengono persi 3, si avrà una probabilità di corretta trasmissione (*successful packet trasmission rate*) pari a 0,7: questa è ciò che viene chiamata *Link Quality* (LQ). É anche importante conoscere questa percentuale per la direzione opposta, cioè per i pacchetti trasmessi dal nodo in questione verso il vicino all'altro capo del link in analisi: questa viene chiamata *Neighbor Link Quality* (NLQ).

Una probabilità più interessante è quella di trasmettere correttamente un pacchetto e ricevere indietro correttamente l'ACK cioè la conferma di avvenuta ricezione; questa probabilità è pari a $LQ \times NLQ$. Se il pacchetto o l'ACK vengono persi, il pacchetto va ritrasmesso; il numero medio di ritrasmissioni per il link in questione (in entrambe le direzioni) è pari a $1/(LQ \times NLQ)$ e questa è esattamente la metrica ETX (*Expected Transmission Count*):

$$ETX = \frac{1}{LQ \times NLQ} \tag{1.3}$$

Per un percorso a più hop la metrica totale, cioè la media delle ritrasmissioni totale, è semplicemente la somma degli ETX di ogni link attraversato. Se la somma degli ETX di due link successivi è minore del ETX del link diretto, allora l'algoritmo sceglierà il percorso a 2 hop.

Per applicare questa metrica al protocollo sono necessarie alcune modifiche. Mentre la LQ è calcolata localmente analizzando gli HELLO-message ricevuti o meno, la NLQ deve essere comunicata direttamente dal vicino. Per far ciò è necessario modificare gli HELLO-message e i TC-message in maniera tale che ogni vicino o nodo annunciato porti con sè anche l'informazione della propria LQ che per il nodo che riceve il messaggio diventa la NLQ. I messaggi così modificati vengono chiamati LQ HELLO-message e LQ TC-message e non sono compatibili con le vecchie versioni: tutti i nodi della rete devono utilizzare ETX come metrica.

Il parametro che governa questa funzionalità è il `LINK_QUALITY_LEVEL`: se pari a 0 l'estensione è disabilitata, se pari a 1 l'ETX è utilizzata solo per la selezione degli MPR, se pari a 2 l'ETX è utilizzata come metrica per tutta la rete nel calcolo delle tabelle di routing. Tutti i nodi della rete devono essere settati con il medesimo valore.

Il secondo parametro, forse più interessante per le sperimentazioni, è la `LINK_QUALITY_WIN_SIZE` che determina il numero di pacchetti da tenere in considerazione per il calcolo delle probabilità di successo. Di default la finestra è lunga 10 pacchetti, considerando l'`HELLO_INTERVAL` di default pari a 2 secondi, la finestra temporale risulta di 20 secondi. Questo valore può essere alzato per reti poco mobili o abbassato per aumentare la reattività ai cambiamenti rapidi dovuti alla mobilità.

L'uso della metrica ETX non è compatibile con la *Link Hysteresis* ed è consigliabile aumentare la validità temporale degli HELLO-message in quanto nella versione standard

dopo 3 HELLO-message mancati il link viene considerato rotto, mentre con ETX è preferibile diminuire la qualità del link e considerarlo rotto dopo un tempo pari a `HELLO_INTERVAL` × `LINK_QUALITY_WIN_SIZE`.

1.1.4 Implementazione

OLSRd (OLSR daemon) è un'implementazione *open source* del protocollo OLSR. É molto semplice da utilizzare in quanto basta lanciarlo indicando un file di configurazione dove si possono settare i valori dalle costanti e le funzionalità da abilitare. Dal file di configurazione si può anche abilitare l'uso di un'interfaccia web che mostra in maniera *user-friendly* le tabelle di routing e le principali informazioni di OLSR.

Per il *testbed* è stato modificato leggermente il software per permettere la scrittura su file delle tabelle di routing e delle principali informazioni di OLSR ogni secondo, al fine di processare e analizzare i dati successivamente: basta lanciare il comando con l'opzione `-d 1` per abilitare il log.

1.2 BATMAN

BATMAN (Better Approach To Mobile Ad-hoc Networking) è stato proposto da A. Neuman, C. Aichele, M. Linder e S. Wunderlich in un Internet-Draft [3] dell'Aprile 2008.

1.2.1 Descrizione del protocollo

BATMAN è un semplice e robusto protocollo di routing proattivo per reti *wireless ad-hoc/mesh* mobili, in grado di garantire alta adattabilità e assenza di loop con un basso costo computazionale e di traffico.

BATMAN non è l'evoluzione di qualche protocollo preesistente, bensì un ibrido tra i due classici approcci al routing su MANET: pur essendo proattivo, in quanto scambia periodicamente pacchetti di controllo presentando immediatamente la rotta quando richiesta, esso non calcola la rotta ottima completa sul grafo della rete (pur essendo a conoscenza di ogni nodo) ma calcola solamente il miglior *next hop*, cioè il nodo vicino migliore su cui instradare il pacchetto verso la destinazione.

In ambiente *wireless* le problematiche sono molte (rumore, interferenza, collisioni, congestione,. . .) e fanno sì che i vari link (ritenuti validi da BATMAN solo se bidirezionali) possano presentare diversi livelli di perdita di pacchetti e rate di trasmissione. Le problematiche aumentano se si passa dalla staticità alla mobilità. Queste problematiche affliggono non solo il traffico dati, ma anche i pacchetti di controllo di BATMAN: sono proprio questi che il protocollo analizza per operare la scelta migliore per i *next hop*. Esso analizza la perdita e la rapidità di questi pacchetti di controllo (OGM), non il loro contenuto, quindi gli OGM possono essere pacchetti praticamente vuoti che non pesano molto sull'utilizzo del canale.

Gli OGM (Originator message) vengono trasmessi in broadcast, quindi ad ogni vicino nel range di trasmissione (*single-hop*), da ogni interfaccia BATMAN di ogni nodo (Originator). Ogni vicino che riceve un OGM lo ritrasmette in broadcast operando così un *flooding* in tutta la rete. BATMAN non fa distinzione tra OGM ricevuti da vicini diretti e OGM ricevuti da nodi distanti più di 1 hop. Il numero di OGM ricevuti da un certo nodo attraverso un certo link è l'informazione principale per determinare il link migliore su cui trasmettere dati destinati a quell'Originator. Per far ciò ogni nodo mantiene una *moving window* per ogni Originator nella rete per ogni nodo vicino

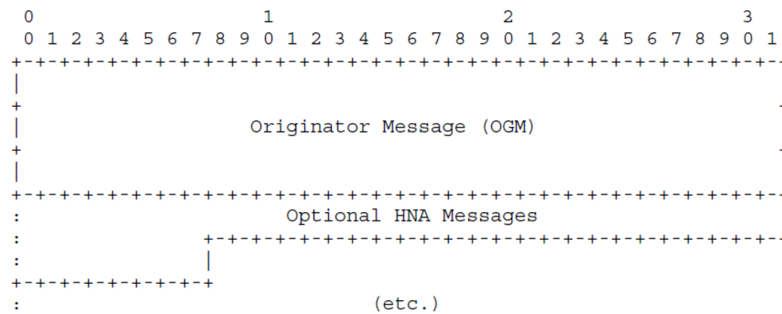


Figura 1.7: Formato del pacchetto di BATMAN

diretto, che può contenere gli ultimi `WINDOW_SIZE` serial number degli OGM provenienti da quell'Originator e ricevuti da quel link: il *next hop* verso quell'Originator è il vicino la cui finestra contiene più serial number.

BATMAN non è stato progettato per reti stabili, ma per contrastare alti livelli di instabilità della topologia e di perdita di pacchetti. Esso cerca di implementare un'idea di "intelligenza collettiva" che si contrappone all'idea del *Link State routing*.

BATMAN considera due funzionalità che si possono definire "aggiuntive" (inseribili al bisogno), come parte integrante del protocollo; esse vengono definite già nell'RFC e integrate nei pacchetti di controllo. Una funzionalità aggiuntiva che BATMAN incorpora direttamente nel protocollo è la possibilità per un Originator di proporsi come *gateway* verso internet, indicando nell'OGM anche la banda disponibile. BATMAN crea un tunnel nella rete mesh per ogni nodo che voglia connettersi con reti esterne assegnandogli una porta costante (importante per un eventuale NAT). BATMAN è in grado di selezionare anche il miglior *gateway* nel caso in cui più di un nodo si proponesse come tale, facendo un *trade-off* tra banda annunciata e numero di hop di distanza. La seconda funzionalità integrata è la possibilità per un Originator di annunciare nodi o reti che lui può raggiungere ma che non partecipano alla rete BATMAN.

Si vedranno ora più in dettaglio le caratteristiche principali di BATMAN per poi passare alle funzionalità aggiuntive. Verranno presentate facendo riferimento all'RFC anche se alcuni cambiamenti importanti sono avvenuti nel corso degli anni, il più grosso dei quali è stato il passaggio dal livello 3 al livello 2. Ogni informazione e procedura resta valida con alcune semplici modifiche, ad esempio quando si parla di indirizzo IP nella versione originale, si dovrà considerare l'indirizzo MAC nella versione a livello 2.

Formato del pacchetto

Il formato di un pacchetto BATMAN è illustrato in Figura 1.7. É costituito da un OGM e da eventuali messaggi HNA (*Host and Network Association*) utilizzati da un Originator per annunciare nodi o reti con cui può comunicare che però non partecipano alla rete BATMAN.

Il formato dell'OGM è mostrato in Figura 1.8. `Version` è la versione di BATMAN utilizzata dall'Originator: se diversa dalla propria, chi riceve il pacchetto lo scarta. `U` è un *flag* che indica se il percorso verso il vicino è unidirezionale o meno, `D` indica invece se il nodo è un vicino diretto. `TTL` è il *Time To Live* che può essere utilizzato per limitare il numero di hop massimi che il pacchetto può effettuare prima di essere scartato. `GWFlags` codifica la banda disponibile in *downstream* e *upstream* nel caso in cui l'Originator si proponesse come *Gateway* verso internet. Un Originator numera ogni

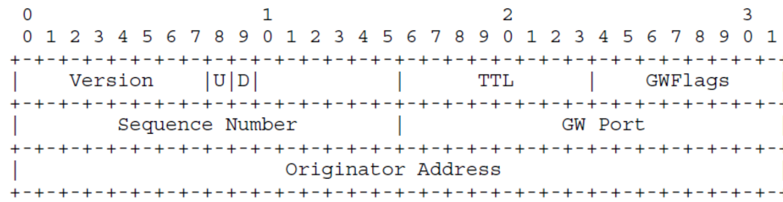


Figura 1.8: Formato dell'OGM di BATMAN

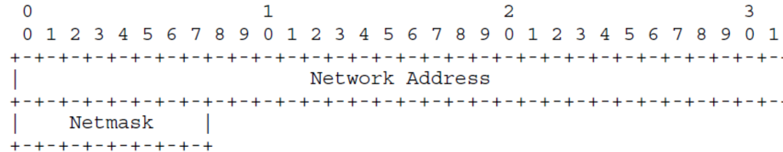


Figura 1.9: Formato del messaggio HNA di BATMAN

nuovo OGM incrementando il `Sequence Number`: questo campo oltre ad evitare doppi *broadcast* dello stesso messaggio nel processo di *flooding* è essenziale per la metrica. `GWPort` è la porta utilizzata per il *tunnelling* verso internet. Infine l' *Originator Address* è l'informazione principale con cui l'Originator si annuncia alla rete.

Il formato di un messaggio HNA è illustrato in Figura 1.9 ed è formato dall'indirizzo IP della rete o dell'*host* annunciato e dalla sua *Netmask*.

Patrimonio informativo di un nodo

Ogni nodo deve mantenere alcune strutture dati per immagazzinare le informazioni che riguardano lui stesso e ogni altro Originator nella rete. Parlando di Originator ci si riferisce ad un nodo nel caso in cui quest'ultimo possieda solo una interfaccia BATMAN, ma nel caso in cui ne possieda più d'una l'Originator è la singola interfaccia.

Ogni nodo deve mantenere aggiornata una *Originator List* cioè la lista di tutti gli Originator conosciuti, cioè quelli dai quali sono stati ricevuti degli OGM durante gli ultimi `PURGE_TIMEOUT` secondi. Per ogni Originator nella lista devono essere mantenute le seguenti informazioni:

- indirizzo IP dell'Originator;
- il *timestamp* dell'ultimo OGM ricevuto dall'Originator;
- il *sequence number* dell'ultimo OGM generato dal nodo di cui si sta analizzando il patrimonio informativo, che ha verificato la bidirezionalità del link verso l'Originator; è utilizzato nella procedura di verifica della bidirezionalità che verrà spiegata in seguito;
- il *sequence number* dell'ultimo OGM ricevuto dall'Originator;
- l'*HNA list*, cioè la lista degli *host* e delle reti che non partecipano alla rete BATMAN ma che l'Originator può raggiungere;
- la capacità o meno dell'Originator di fungere da *gateway* verso internet e gli eventuali parametri;

- la *Neighbor Information List*, cioè la lista dei vicini diretti; per ogni link verso un certo elemento della lista, quindi verso un dato vicino diretto, devono essere mantenute aggiornate le seguenti informazioni e strutture dati :
  - una *sliding window* degli ultimi possibili `WINDOW_SIZE` *sequence number* che memorizza se l'OGM con quel dato *sequence number* è stato o meno ricevuto dal link in analisi; questa struttura serve per il calcolo della metrica, che verrà analizzato in seguito;
  - il numero di *sequence number* nella *sliding window* in cui si sono ricevuti OGM con quel dato *sequence number*; questo costituirà la metrica del percorso verso quell'Originator usando come primo hop il link in questione.

Meccanismo di flooding

Ogni nodo deve, per ogni sua interfaccia BATMAN, trasmettere in *broadcast* un OGM ogni `ORIGINATOR_INTERVAL` secondi. Il primo OGM trasmesso dovrà essere inizializzato con i *flag* `D` e `U` a 0, il `TTL` settato con il valore desiderato tra `TTL_MIN` e `TTL_MAX`, il *sequence number* inizializzato ad un valore casuale nel range disponibile e i campi riguardanti la possibilità di fungere da *gateway* settati adeguatamente. Se il nodo vuole annunciare reti o *host* non BATMAN dovrà aggiungere nel pacchetto oltre all'OGM il messaggio HNA.

Alla ricezione di un OGM il nodo deve effettuare dei controlli e intraprendere delle azioni di conseguenza:

- il pacchetto va scartato se la versione di BATMAN è diversa, se è stato originato dal nodo stesso (se da un altra interfaccia va effettuato un *processing* e poi scartato), se il *flag* `U` (*unidirectional*) è settato a 1;
- il pacchetto va processato (passato al calcolo delle tabelle di *routing*) se è stato ricevuto da un link bidirezionale E contiene un nuovo *sequence number* (non è un duplicato di un OGM già ricevuto);
- il pacchetto va reinoltrato in *broadcast* se è stato ricevuto da un vicino diretto oppure se è stato ricevuto da un link bidirezionale E è stato ricevuto dal miglior link E non è un duplicato.

Prima del reinoltro in *broadcast* dell'OGM alcuni campi del pacchetto vanno cambiati:

- il `TTL` va decrementato di 1 e se raggiunge lo 0 il pacchetto va scartato;
- il *flag* `D` (*direct link*) va posto a 1 se il pacchetto è stato ricevuto da un vicino diretto E, effettuando il reinoltro verso tutti, il pacchetto in questione è quello che deve essere reinoltrato sul medesimo link da cui è stato ricevuto;
- il *flag* `U` (*unidirectional link*) va posto a 1 se il pacchetto è stato ricevuto da un link unidirezionale.

Controllo della bidirezionalità del link

BATMAN considera validi per il *routing* solo i link bidirezionali, quindi deve verificare che un link scoperto verso un vicino sia bidirezionale. Per far ciò un nodo deve tenere in memoria il *sequence number* di un OGM generato da lui stesso e reinoltrato in *broadcast* da un vicino diretto se:

Tabella 1.3: Valori proposti dal RFC di BATMAN per le costanti

`Costante`	valore proposto
`VERSION`	4
`TTL_MIN`	2
`TTL_MAX`	225
`SEQNO_MAX`	65535
`BROADCAST_DELAY_MAX`	100 ms
`ORIGINATOR_INTERVAL`	1000 ms

- l'OGM generato dal nodo stesso viene poi ricevuto attraverso la stessa interfaccia da cui era stato trasmesso;
- il *flag* `D` è stato settato a 1;
- il *sequence number* corrisponde a quello dell'ultimo OGM trasmesso in *broadcast* da quell'interfaccia.

Gateway

Un nodo che abbia accesso a internet può proporsi come *gateway* per la rete BATMAN annunciandosi come tale negli OGM che genera. Nel campo `GWFlags` può inserire la banda che ha a disposizione in *download* e in *upload*.

Se più di un nodo nella rete si è proposto come *gateway*, ogni nodo può calcolare il migliore per lui in base alla qualità del percorso verso il *gateway* e alla banda che offre. Ogni nodo può indipendentemente scegliere la politica che vuole adottare nel *tradeoff* tra qualità del percorso (conta la metrica, influenzata quindi dal numero di hop e dalla qualità dei link) e la banda disponibile. Per un nodo mobile, ad esempio, la priorità sarà la qualità del percorso: non avrebbe senso connettersi ad un nodo che muovendosi è diventato molto distante solo perché offre più banda; la scelta del *gateway* dovrebbe essere molto più reattiva alla mobilità. Per un nodo fisso, invece, ha più senso concentrarsi sulla banda che il *gateway* mette a disposizione, perché una volta scelto probabilmente resterà quello per molto tempo.

Un GW-client, cioè un nodo che deve trasmettere un pacchetto verso internet tramite il *gateway* prescelto, incapsula il pacchetto effettuando un *tunnelling* verso il *gateway*. Il *gateway* decapsula il pacchetto e lo trasmette verso la destinazione originaria. Non è usato il *tunnelling* nella direzione opposta, quindi per i pacchetti che provengono da internet.

Valori proposti

I valori proposti dall'RFC per le costanti più importanti sono elencati in Tabella 1.3.

1.2.2 Metrica

L'informazione chiave usata da BATMAN per calcolare la metrica è il *sequence number* degli OGM. Come già accennato ogni nodo mantiene una *sliding window* per ogni Originator conosciuto, per ogni link che lo collega ad un vicino diretto. Le *sliding window* che riguardano un certo Originator sono formate da `WINDOW_SIZE` posti che rappresentano il *sequence number* dell'ultimo OGM ricevuto dall'Originator e i

`WINDOW_SIZE` − 1 *sequence number* precedenti. Quando un OGM viene ricevuto, viene registrato il link dal quale è arrivato riempiendo il posto del suo *sequence number* nella *sliding window* del link in questione. L'algoritmo confronta le *sliding window* e decide che il *next hop* per raggiungere l'Originator è il nodo all'altro capo del link con la *sliding window* più piena.

Quando viene ricevuto un OGM da un certo Originator con un *sequence number* più recente di qualsiasi altro ricevuto dallo stesso, le *sliding window* legate a quell'Originator devono scorrere in avanti, non considerando più i *sequence number* più vecchi; va quindi ricalcolato il miglior link e di conseguenza il *next hop* da scrivere nelle tabelle di *routing*.

1.2.3 Funzionalità aggiuntive

Translation Table

Il meccanismo delle *Translation Table*, disponibile dalla versione 2013.3.0 di BATMAN-adv (implementazione di BATMAN di cui si discuterà in seguito), permette il miglioramento della gestione dei nodi "non BATMAN" connessi alla rete *mesh*. Una situazione classica prevede una *backbone* di nodi BATMAN che si occupano del *routing* a cui si connettono dei nodi "non-BATMAN": se questi nodi si possono muovere devono potersi agganciare al nodo BATMAN più vicino. Prima dell'introduzione delle *Translation Table* la connessione si bloccava e prima di riprendere doveva aspettare la stabilizzazione delle informazioni scambiate tramite i messaggi HNA. Le *Translation Table* permettono, grazie a un meccanismo di *versioning*, di non far cadere la connessione ai nodi "non-BATMAN" in movimento.

Distributed ARP Table

Le DAT (Distributed ARP Table) sono delle tabelle che permettono il *caching* delle informazioni del protocollo ARP in tutta la rete *mesh*. Se un nodo "non BATMAN" fa una richiesta ARP in *broadcast*, il nodo BATMAN a cui è agganciato controlla la sua DAT e, se la tabella contiene già le informazioni richieste, blocca la richiesta e risponde, evitando così un *broadcast* in tutta la rete *mesh*. Questo fa sì inoltre che il nodo "non BATMAN" riceva una risposta più affidabile e con meno ritardo.

Bridge Loop Avoidance

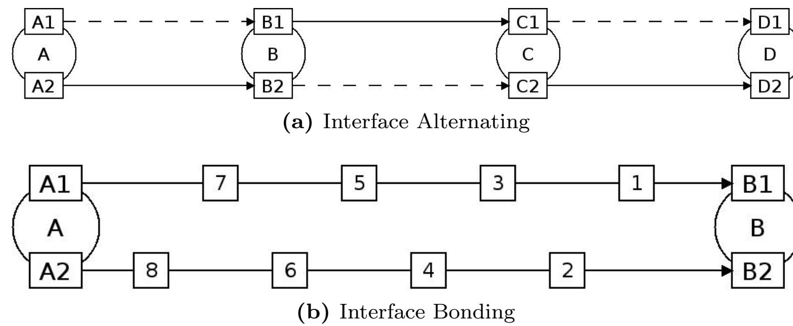
Il *Bridge Loop Avoidance* permette di evitare che si creino *loop* nel *routing* causato da nodi che possiedono più di un interfaccia BATMAN che partecipa alla *mesh*.

Ottimizzazioni multi-link

Per i nodi con più di un'interfaccia BATMAN sono possibili delle ottimizzazioni che incrementano il *throughput* (illustrate in Figura 1.10):

Interface Alternating: BATMAN inoltra pacchetti su un'interfaccia diversa da quella da cui sono stati ricevuti per ridurre l'interferenza;

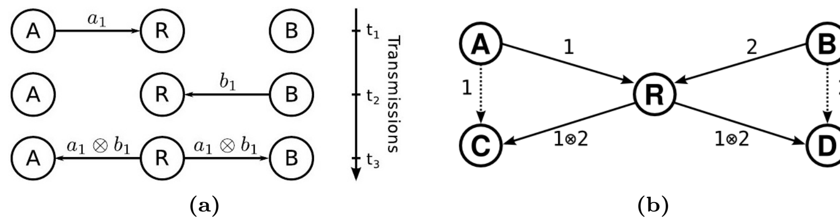
Interface Bonding: se sono disponibili più link di qualità simile verso un vicino, BATMAN può distribuire i pacchetti a lui destinati sui diversi link incrementando il *throughput*.



(a) Interface Alternating

(b) Interface Bonding

Figura 1.10: Ottimizzazioni multi-link



(a) **(b)**

Figura 1.11: Topologie di rete in cui si può utilizzare il *Network Coding*.

AP Isolation

AP Isolation è una funzionalità che permette di evitare che due nodi possano comunicare attraverso un *Access Point* di una rete infrastrutturata.

Network Coding

La funzionalità chiamata *Network Coding* permette ad un nodo *relay* di combinare due pacchetti in uno per risparmiare occupazione del canale. Un nodo ricevente può "decodificare" un pacchetto combinato solo se ne conosce già uno dei due, perciò l'uso di questa funzionalità è limitato a topologie di rete ben definite come ad esempio quelle illustrate in Figura 1.11.

Frammentazione

Dopo la realizzazione del protocollo a livello 2 e dopo l'introduzione delle *Translation Table* è stato scelto di implementare una frammentazione dedicata e ottimizzata per BATMAN.

1.2.4 Implementazione

La prima implementazione di BATMAN lavorava a livello 3: era un demone che comunicava tramite protocollo UDP e scriveva le rotte scoperte nelle *routing table* del kernel. L'ultima versione di BATMAN, denominata BATMAN-adv (*advanced*) è implementata come modulo kernel che lavora a livello 2, cosa che comporta vantaggi e svantaggi. I principali svantaggi sono:

- l'inutilizzabilità di alcuni comandi classici come `ping` e `traceroute` (inconveniente risolto dalla suite di comandi `batctl` di cui si parlerà in seguito);

- il fatto che nelle tabelle di *routing* del kernel tutti i nodi della rete BATMAN appaiono come vicini diretti (per l'ispezione delle tabelle reali viene ancora in aiuto `batctl`);
- l'inefficacia del *firewall* di Linux `iptables` in quanto questo lavora a livello 3 e non riesce a intercettare i pacchetti di BATMAN prima che vengano mandati al modulo (questo problema non è stato risolto).

Le motivazioni che hanno invece avallato questa scelta sono:

- BATMAN può operare direttamente con frame Ethernet risparmiando occupazione del canale;
- si può "montare" sopra a BATMAN-adv qualsiasi protocollo (IPv4, IPv6, DHCP, ...);
- i nodi partecipano alla *mesh* senza che l'interfaccia *wireless* abbia un indirizzo IP;
- è più facile l'integrazione con i nodi "non-BATMAN";
- sono possibili le ottimizzazioni multi-link.

Si può interagire con BATMAN attraverso i file contenuti nella cartella `/batman_adv` che si crea nella cartella di `/sys/net/` di ogni interfaccia in cui si è attivato BATMAN, oppure più comodamente con il comando `batctl` che permette in maniera più *user-friendly* di effettuare settaggi, visualizzare la situazione e usare strumenti altrimenti inutilizzabili (quali ping, traceroute, tcpdump,...). I comandi più utili della suite `batctl` sono:

`batctl o` **:** mostra le tabelle di *routing* di BATMAN, comprese le metriche del miglior *next-hop* attuale e dei potenziali altri *next-hop* verso un Originator;

`batctl p [destination MAC]` **:** effettua il `ping` verso l'indirizzo MAC passato come argomento;

`batctl tr [destination MAC]` **:** effettua il `traceroute` verso l'indirizzo MAC passato come argomento.

Tramite `batctl` è inoltre possibile cambiare i diversi settaggi e abilitare o disabilitare le funzionalità aggiuntive. I parametri più interessanti su cui si può riflettere e sperimentare nel *testbed* sono:

`HOP_PENALTY`**:** indica quanto il nodo non è adatto a fungere da *relay* in un percorso *multi-hop*; di default è impostata a 30, si può alzare se si preferisce non passare per questo nodo (ad esempio si può impostare a 255 se si tratta di un nodo molto mobile) o abbassare se si vuole spingere ad attraversarlo nei percorsi più lunghi;

`ORIGINATOR_INTERVAL`**:** specifica ogni quanti millisecondi un nodo trasmette un OGM; di default è impostato a 1000 (un OGM al secondo), si può alzare in situazioni particolarmente statiche e abbassare in situazioni particolarmente mobili a discapito del traffico sul canale.

Un'altra comoda *feature* di BATMAN-adv è la possibilità di creare un file chiamato `bat-hosts` posizionato in `/etc/` che contiene in ogni riga una coppia "indirizzo_MAC nome_user_friendly": se il file è presente, in tutti i comandi in cui sarebbe necessario scrivere un indirizzo MAC si può scrivere invece il nome assegnato al nodo.

Una particolarità di BATMAN è che abilitandolo su un'interfaccia di rete (ad esempio wlan0) esso crea un'interfaccia virtuale bat0 che utilizzerà per le connessioni. L'indirizzo MAC dei pacchetti resta invece quello dell'interfaccia originale.

Capitolo 2

Testbed

Al Dipartimento di Ingegneria dell'Informazione dell'Università degli studi di Padova è stato realizzato un *testbed* per sperimentare il paradigma del *cognitive networking* su reti *mesh* Wi-Fi (802.11). Gli è stato dato il nome di AWMN [5, 6] (*Android Wireless Mesh Network*) perché sfrutta il fatto che Android, sistema operativo (SO) adottato dalla maggior parte dei dispositivi di nuova generazione come *tablet* e *smartphone*, si appoggia su un kernel Linux opportunamente modificato. Ogni codice creato per calcolatori Linux è quindi potenzialmente eseguibile su dispositivi Android. Questo è importante per avere una rete con codice omogeneo pur essendo costituita sia da dispositivi fissi (calcolatori con kernel Linux) sia da dispositivi mobili (tablet Android). Il *testbed* è stato costruito secondo i seguenti principi:

- i nodi sono dispositivi commerciali relativamente economici comparati con l'*hardware ad-hoc* usato in altri *testbed* su reti *mesh* Wi-Fi;
- il *testbed* deve permettere la raccolta e la modifica in *real time* di tutti i principali parametri dei diversi livelli dello *stack* protocollare per permettere di sperimentare tecniche di *cognitive networking*;
- il *testbed* deve essere facilmente riproducibile da altri gruppi di ricerca.

2.1 Concetti preliminari

Si analizzeranno ora alcuni concetti di base per comprendere la struttura e gli obiettivi del *testbed*.

2.1.1 Mesh network

Le tecnologie radio sviluppate nel corso degli anni sono molto diverse tra loro, ma il modello di rete largamente utilizzato finora è quello delle reti di accesso *wireless* in cui il collegamento senza fili è solo l'ultimo tra il terminale dell'utente e una stazione di accesso, denominata stazione radio base o *access point* (AP) in base alla tecnologia. Il resto della rete dietro la stazione d'accesso è costituito da collegamenti punto-punto quasi sempre cablati che costituiscono uno dei costi più rilevanti dell'intera infrastruttura di rete e il limite principale allo sviluppo della sua capacità di traffico verso sistemi a larghissima banda. L'infrastruttura cablata rappresenta la principale voce di costo negli scenari micro-cellulari necessari per i servizi a larga banda, in molte applicazioni

che richiedono il collegamento di numerosi punti di raccolta informazioni (come nei sistemi di videosorveglianza, monitoraggio ambientale, automazione industriale, ...) e quando lo scenario rende particolarmente complesso posare dei cavi (edifici storici, aree protette, manifestazioni temporanee, ...). Inoltre, il cablaggio richiede tempi di realizzazione normalmente elevati che sono non solo negativi in sé, ma spesso possono creare disagio alle attività che si svolgono nell'area. Per queste ragioni, negli ultimi anni la ricerca scientifica si è concentrata nella ricerca di una soluzione che consenta di sostituire in tutto o in parte l'infrastruttura cablata con una rete anch'essa *wireless* come quella d'accesso. Il *wireless mesh networking*, che si è recentemente affacciato sul mercato, rappresenta il frutto di questo lavoro di ricerca e la nuova frontiera delle reti *wireless*.

La caratteristica principale di questa tecnologia è l'estrema adattabilità alle condizioni operative che consente di creare velocemente la rete e di limitare al minimo le operazioni di gestione degli apparati, apparati che devono essere in grado di configurarsi autonomamente e di reagire velocemente ad ogni cambiamento del sistema (guasti di apparati o collegamenti, modifiche delle condizioni di propagazione, mutate condizioni di traffico, ...). In molti ambiti applicativi il *mesh networking* rappresenta oggi un nuovo paradigma di interconnessione in grado di superare i limiti delle tecnologie *wireless* tradizionali da un lato e dell'approccio IP tradizionale dall'altro.

La banale sostituzione di alcuni collegamenti dell'infrastruttura di rete cablata delle reti *wireless* d'accesso con collegamenti radio punto-punto di per sé non è definibile come *mesh networking*. Non sono propriamente *mesh* neanche le reti Wi-Fi con *Wireless Distribution System* (WDS) costituito da collegamenti radio gestiti a livello 2 con tabelle di inoltro statiche normalmente configurate manualmente, e neppure soluzioni basate su ponti radio (di qualunque tipo) con infrastruttura di rete IP. Le reti *wireless mesh* sono una tecnologia evoluta che si basa sui nuovi protocolli ed algoritmi di controllo che costituiscono il nuovo paradigma di *networking wireless*.

I dispositivi di rete (*mesh router*) devono essere in grado di configurarsi in modo automatico e di adattarsi autonomamente alle condizioni operative. I protocolli di rete devono gestire l'instradamento in modo dinamico scegliendo il miglior percorso in base alle condizioni di propagazione/interferenza e alle frequenze utilizzate. Tutti o alcuni dei dispositivi di rete devono poter svolgere in modo flessibile le funzioni di stazione di accesso *wireless* fornendo copertura ai terminali utente. Uno o più dispositivi devono fungere da *gateway* verso internet ovvero da termine della rete *mesh*.

Lo schema generale di una rete *mesh* è disegnato in Figura 2.1, mentre un esempio di applicazione in un'area ampia come può essere un quartiere o un centro congressi, in cui è necessario gestire diversi servizi (videosorveglianza, VoIP, ...) è illustrato in Figura 2.2.

Viene creata spesso confusione nella differenza tra una rete *mesh* e una rete *ah-hoc*. La differenza sta nel fatto che in una rete *ad-hoc* tutti i nodi sono paritari e devono avere piena capacità di *routing*, in una rete *mesh* invece la *backbone* è costituita da nodi chiamati *mesh router* che sono i soli a dover essere capaci di instradare i pacchetti, i nodi utente sono quindi esonerati da questo compito. La confusione deriva dal fatto che generalmente le reti *mesh* vengono considerate come reti *ad-hoc* in virtù del fatto che le tecnologie e le tecniche sviluppate per le reti *ad-hoc* vengono utilizzate anche nelle reti *mesh*. In realtà è più ragionevole considerare le reti *ad-hoc* come un caso specifico di una rete *mesh* e non il contrario. Infatti, come è stato precedentemente evidenziato, le reti *mesh* beneficiano di una componente di infrastruttura formata dai *mesh router* che garantisce maggiore robustezza alla rete e fa sì che il carico sugli *end point* sia minore, componente non presente nelle reti *ad-hoc*.

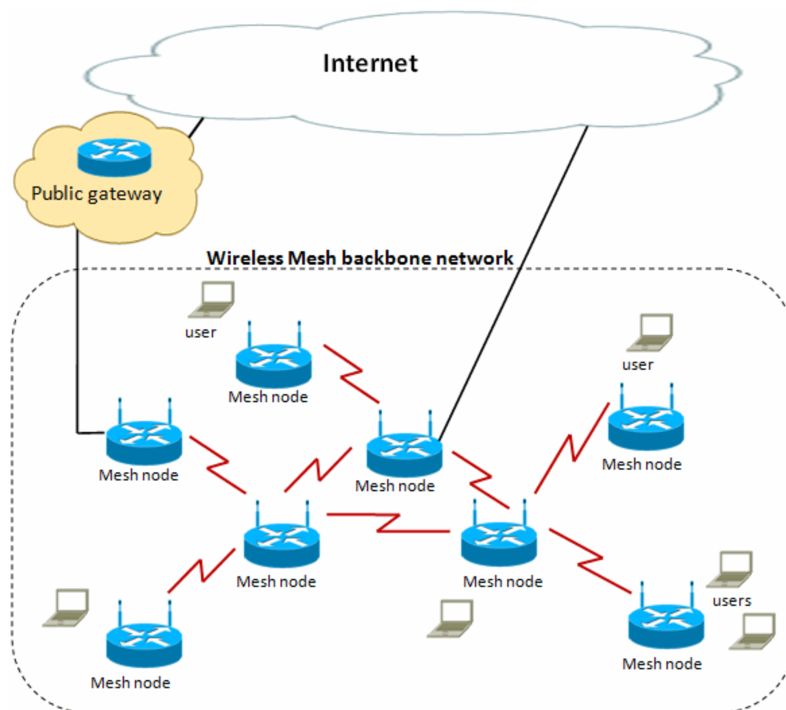


Figura 2.1: Schema generale di una rete mesh.

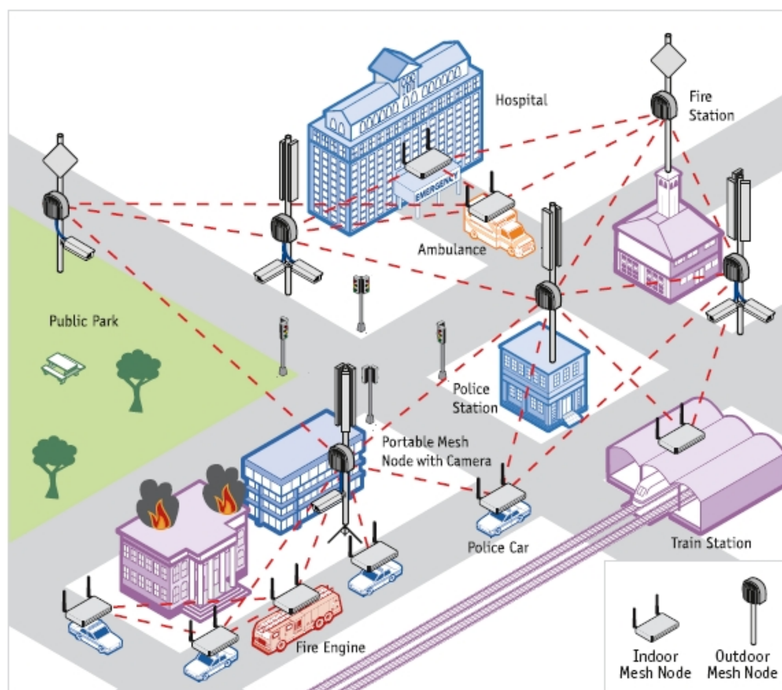


Figura 2.2: Esempio applicativo di una rete mesh.

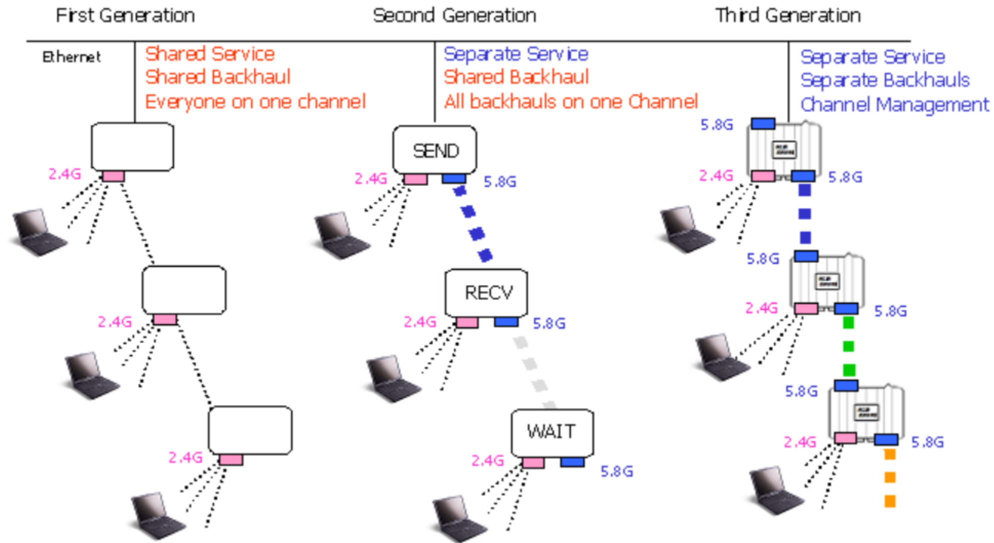


Figura 2.3: Diverse modalità di comunicazione in una rete *mesh*.

Esistono diversi modi per creare la connettività tra i *mesh router* (spesso chiamata *backhaul*) e contemporaneamente fornire connettività ai nodi utente [7]. Le principali modalità sono illustrate in Figura 2.3. Nel primo caso la *backhaul* e la connettività verso i nodi utente vengono create da una stessa interfaccia *wireless* su un unico canale; nel secondo caso le due connettività vengono create da due interfacce radio diverse che lavorano su canali diversi (nel caso della figura addirittura in bande diverse) ma la *backhaul* viene instaurata sullo stesso canale, quindi non può esserci ricezione e trasmissione simultanea; nel terzo caso la *backhaul* viene creata da un gestore dei canali radio che riesce ad evitare l'interferenza permettendo la potenziale ricezione e trasmissione simultanea. Il problema principale che si presenta nella prima e nella seconda situazione è che il *bitrate* della connessione viene condiviso tra i tanti nodi nello stesso raggio di interferenza e ciò comporta l'impossibilità di una ricezione e trasmissione simultanee. Questo fa sì che per ogni hop successivo al primo la banda disponibile venga dimezzata. Ad esempio per un nodo utente che si connette ad un *mesh router* che deve instradare il pacchetto su un percorso a 4 hop la banda risulterà $\frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} = \frac{1}{16}$ della banda originale a disposizione su un canale. Si ha quindi un andamento del tipo $\frac{1}{2^N}$ con N il numero di hop. Questo fenomeno sarà ben visibile nei risultati degli esperimenti essendo il *testbed* creato del primo tipo.

Un altro problema che affligge le reti *mesh* del primo tipo (con una singola interfaccia radio per *mesh router*) è la condivisione della banda. Come in ogni rete Wi-Fi l'accesso al mezzo è regolato dalla DCF (*Distributed Coordination Function*) adattamento per reti radio del classico CSMA (*Carrier Sense Multiple Access*). Un nodo non può trasmettere se sente un'altra trasmissione in atto per non causare collisioni, quindi ogni nodo che voglia trasmettere deve attendere di trovare il canale libero. Se la *backhaul* e la connessione con i nodi utente lavorano sullo stesso canale, tutti per trasmettere dovranno condividere la stessa banda. Se ogni *mesh router* dà connettività a C nodi utente, ad ogni hop la banda viene divisa per $C + 1$ (le connessioni con gli utenti più la connessione della *backhaul*), perché ogni volta la connessione verso il *mesh router* successivo deve ospitare le $C + 1$ connessioni precedenti; l'andamento è quindi $\frac{1}{(C+1)^N}$.

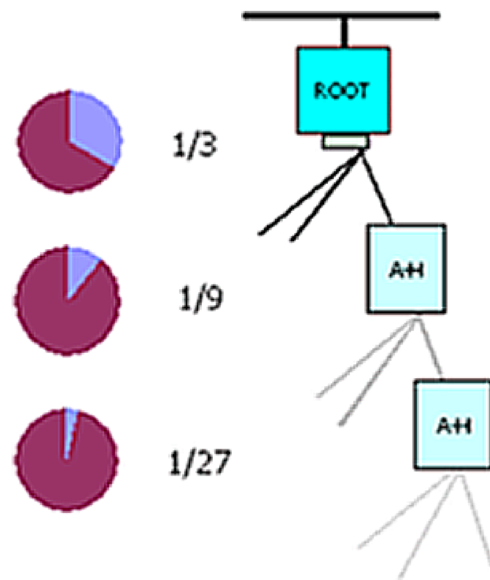


Figura 2.4: Problema della condivisione della banda in una rete *multihop*.

Ipotizzando che ogni *mesh router* dia connettività a 2 nodi utente, la banda sul primo tratto sarà già $\frac{1}{3}$ di quella nominale, e ad ogni hop successivo sarà $\frac{1}{9}$, $\frac{1}{27}$, ... come illustrato in Figura 2.4.

2.1.2 Cognitive network

W. Thomas in [8] definisce una *cognitive network* (CN) come una rete su cui è attivo un processo cognitivo che prende in input l'intera situazione della rete (parametri e misure di tutti i livelli dello stack protocollare), processa queste informazioni e restituisce in output le azioni e le modifiche da attuare agli stessi parametri che ha processato. Il processo cognitivo può imparare dalle modifiche che ha imposto ed usare ciò che ha appreso per le decisioni future. Ogni decisione che il processo prende deve essere votata ad obiettivi *end-to-end*. Per obiettivi *end-to-end* si intende un'ottimizzazione globale di tutta la rete e non di un aspetto singolo.

Il paradigma *cognitive network* è diverso dal *cognitive radio* che interessa solo il livello fisico (parametri come frequenza, canale, forme d'onda, ...), e spesso mira ad ottimizzare l'accesso al canale di dispositivi chiamati "secondari" (AP e dispositivi Wi-Fi privati) che lavorano sulle stesse frequenze di dispositivi usati per le emergenze chiamati "primari"; gli standard 802.11 lavorano infatti su bande di frequenze chiamate ISM (*Industrial, Scientific and Medical*) che permettono l'utilizzo di dispositivi privati a bassa potenza a patto che non sia in atto nello stesso luogo una trasmissione da parte di dispositivi primari.

L'approccio del *cognitive networking* non va confuso con un approccio che si può definire *cognitive layer* che punta all'ottimizzazione di un singolo aspetto; un esempio estremizzato potrebbe essere un protocollo di livello MAC che punta alla minimizzazione della potenza consumata che però potrebbe causare un aumento degli hop nella scelta del percorso, che a sua volta potrebbe causare un aumento del ritardo *end-to-end*

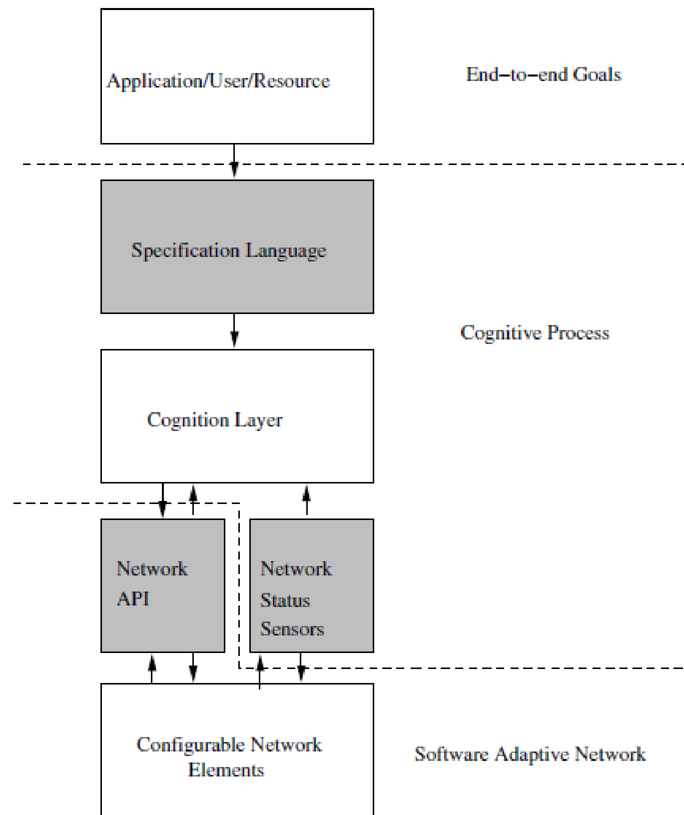


Figura 2.5: *Framework* cognitivo di un nodo di una *cognitive network*.

con un conseguente peggioramento del protocollo di trasporto che aumenterebbe le ritrasmissioni facendo aumentare la potenza globale consumata.

Il *cognitive network* non può nemmeno essere confuso con un semplice approccio *cross-layer* perché mancherebbe l'aspetto cognitivo, cioè l'apprendimento dalle decisioni prese.

Idealmente una CN deve essere in grado di prendere decisioni per il futuro, non solo reattive, in maniera da prevenire problemi prima che si presentino. Infine l'architettura di una CN deve essere flessibile e facilmente espandibile.

Implementare una CN comporta un ovvio aumento della complessità del sistema, in termini di *overhead*, architetture e calcolo computazionale; è quindi necessario che questo aumento di complessità sia giustificata da un adeguato aumento delle prestazioni.

Il *framework* cognitivo di un nodo di una *cognitive network* è illustrato in Figura 2.5. Al livello più alto troviamo gli obiettivi *end-to-end*, che devono essere un insieme di parametri da ottimizzare che riguardano l'intera rete (*routing*, connettività globale, affidabilità della rete, . . .). Il livello intermedio è il processo cognitivo che comprende un interfaccia con gli obiettivi, gli algoritmi di calcolo cognitivo e il sistema che raccoglie i parametri e le misure dei livelli dello *stack* protocollare. Al livello più basso troviamo la rete che deve essere il più possibile modificabile in ogni suo aspetto e ad ogni livello.

2.1.3 Android

Android è il SO più diffuso al mondo per dispositivi portatili come tablet e smartphone. Android è molto flessibile e personalizzabile grazie al suo *Sofware Development Kit* che permette di sviluppare le proprie applicazioni (app). Inoltre è possibile agire sul kernel Linux (disponibile pubblicamente) grazie al *Native Development Kit* (NDK) che permette di compilare codice C/C++.

Android è costituito da tre *layer*: nel livello più basso si trova il kernel Linux che si interfaccia direttamente con l'*hardware*; al livello intermedio troviamo le librerie e la *Dalvik Virtual Machine* che interfaccia il livello kernel con il livello più alto, quello dove risiedono le applicazioni. Il livello kernel in realtà non è direttamente accessibile nei dispositivi commercializzati, per accedervi è necessario acquisire i diritti di *root*.

2.2 Realizzazione di base del testbed

La realizzazione del *testbed* comporta alcuni passaggi preliminari:

1. è necessario abilitare la modalità *ad-hoc* anche nei dispositivi Android; l'unica modalità per la comunicazione diretta tra nodi concessa da Android è chiamata *Wi-Fi Direct* ed è costruita sopra al *transport layer* quindi non è adatta allo scopo del *testbed*; è stato modificato il kernel Android reinserendo la modalità *ad-hoc* nei driver della scheda Wi-Fi e ricompilandolo;

2. una volta abilitata la modalità *ad-hoc* in tutti i dispositivi è necessario creare la rete *mesh* equipaggiandola con un protocollo di *routing*. É qui che si inserisce questa tesi che confronta alcuni protocolli di *routing* ed effettua alcuni esperimenti.

3. è necessario infine creare un meccanismo per estrapolare e modificare i parametri del TCP (*transport Control Protocol*) e del livello MAC (*Media Access Control*). A questo scopo è stato creato il *software* CogNet di cui si parlerà in seguito.

2.3 Hardware

Per rendere il *testbed* flessibile e adatto ad ogni tipo di esperimento si sono utilizzati nodi fissi e mobili. I nodi fissi sono nodi ALIX 3d3 o 3d2 mentre i nodi mobili sono tablet Nexus 7.

2.3.1 ALIX

Gli ALIX sono calcolatori con caratteristiche limitate, elencate in Tabella 2.1, ottimizzati per il *networking*. È possibile osservare un ALIX 3d3 in Figura 2.6.

Gli ALIX montano una scheda di rete *wireless* Atheros 5k 802.11b/g. Ogni ALIX possiede inoltre una scheda *ethernet*.

Il sistema operativo installato sui nodi ALIX è *Voyage Linux*, una derivazione di Debian pensata per dispositivi x86 *embedded* a basso consumo energetico, particolarmente indicata per il *networking* (*routing*, *firewall*, funzionalità *wireless*, ...).

2.3.2 Nexus 7

Nexus 7 è un tablet ideato da Google e prodotto da ASUS, le cui caratteristiche sono elencate in Tabella 2.2. Per poter avere pieno accesso al dispositivo è stato necessario

Tabella 2.1: caratteristiche di un ALIX 3d3.

caratteristica	hardware montato
CPU	500 MHz AMD Geode LX800
DRAM	256 MB DDR DRAM
Storage	CompactFlash socket
Power	DC jack or passive POE, min. 7V to max. 20V
Three LEDs	presenti
Expansion	2 miniPCI slots, LPC bus
Connectivity	1 Ethernet channel (Via VT6105M 10/100)
I/O	DB9 serial port, dual USB, VGA, audio headphone out / microphone in
RTC battery	presente
Board size	100 x 160 mm
Firmware	Award BIOS

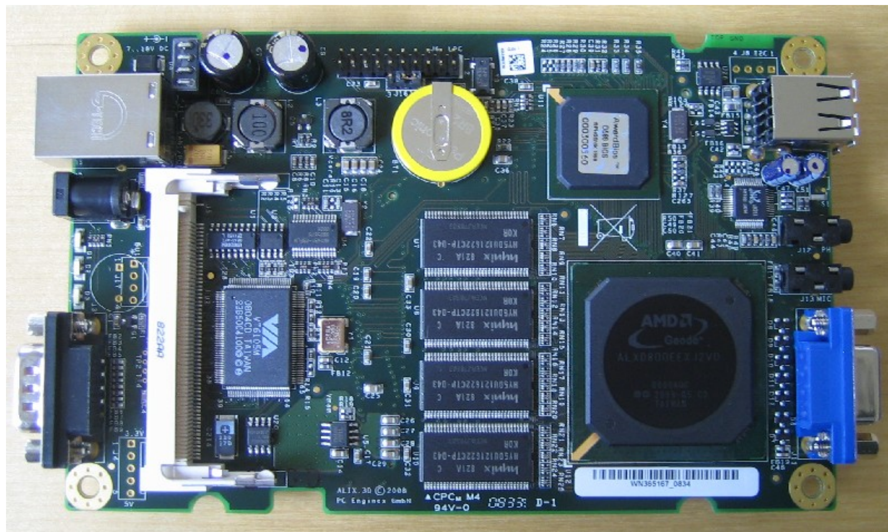


Figura 2.6: ALIX 3d3.

Tabella 2.2: Caratteristiche di un Nexus 7.

caratteristica	hardware montato
CPU	NVIDIA Tegra 3, 1.3 GHz, quad-core, Cortex-A9
RAM	1 GB
Storage	memoria flash 8 GB
Connettività	Wi-Fi 802.11b/g/n (n a 2.4 GHz), Bluetooth, NFC, OTG
Videocamera	frontale, 1.3 MP
Input	TouchScreen, Soft-Keyboard
Schermo	7.0
Risoluzione	1280x800 HD
Dimensioni	198.5x120x10.45 mm
Touchscreen	capacitivo a 10 punti

acquisire i diritti di *root* installando una ROM modificata, la *CyanogenMOD*, derivata da Android Kit Kat 4.4.2.

I Nexus 7 possiedono una scheda *wireless* BCM4330 che necessita del driver Linux `bcmdhd`. Possono essere inoltre equipaggiati con una scheda *wireless* esterna USB (*network adaptor*) Atheros AR9280 che necessita del driver Linux `ath9k_htc`. Avere due schede *wireless* può ampliare di molto le possibilità sperimentali del *testbed*. Il tablet e il *network adaptor* sono illustrati in Figura 2.7.

2.4 Struttura del testbed

Il *testbed* al momento, per quanto riguarda i nodi fissi, è formato da 5 ALIX disposti su un unico piano, come illustrato in Figura 2.8. L'antenna di alcuni ALIX è stata portata tramite un cavo coassiale in una posizione più consona. I nodi (o le antenne) sono stati disposti in modo da formare un percorso in lunghezza per analizzare l'effetto del *multihop* sul TCP. La rete *wireless* scelta è la 192.168.1.0 . Ogni ALIX è inoltre controllabile in ogni momento attraverso la rete cablata Ethernet 10.1.129.0 . Al centro del corridoio tra il nodo .133 e il nodo .138 è presente un *Access Point* (AP) della rete Wi-Fi dipartimentale che genera molto traffico Wi-Fi su canali variabili; uno *screen-shot* della situazione è osservabile in Figura 2.9.

Lasciando la possibilità ai nodi di usare la potenza di trasmissione massima, quasi tutti i nodi sono nel *range* di trasmissione di ogni altro. Ad esempio il nodo .131 riesce a raggiungere tutti gli altri nodi tranne il .134 (il più distante) che raggiunge in 2 hop tramite il .135; questa situazione è ben osservabile nella tabella di *routing* illustrata in Figura 2.10; dalla stessa tabella si osserva anche che i canali di trasmissione sono quasi perfetti, perché la metrica ETX di OLSR è minima: vale 1 per i collegamenti 1 hop e 2 in quelli a 2 hop, tranne per il nodo .135 (il più distante raggiungibile in 1 hop) in cui la metrica è leggermente superiore a 1. È molto interessante osservare anche che, come descritto nella sezione di OLSR, la metrica ETX verso il nodo .134 è la somma dei due valori di ETX dei due hop: si evince quindi che anche il canale tra il nodo .135 e .134 è molto buono avendo ETX pari a 1.

Per riprodurre una situazione reale, in cui i nodi sono più distanti tra loro e quindi devono fare più hop per arrivare a destinazione e i canali di trasmissione non sono

Figura 2.7: Nexus 7 e Atheros AR9280.

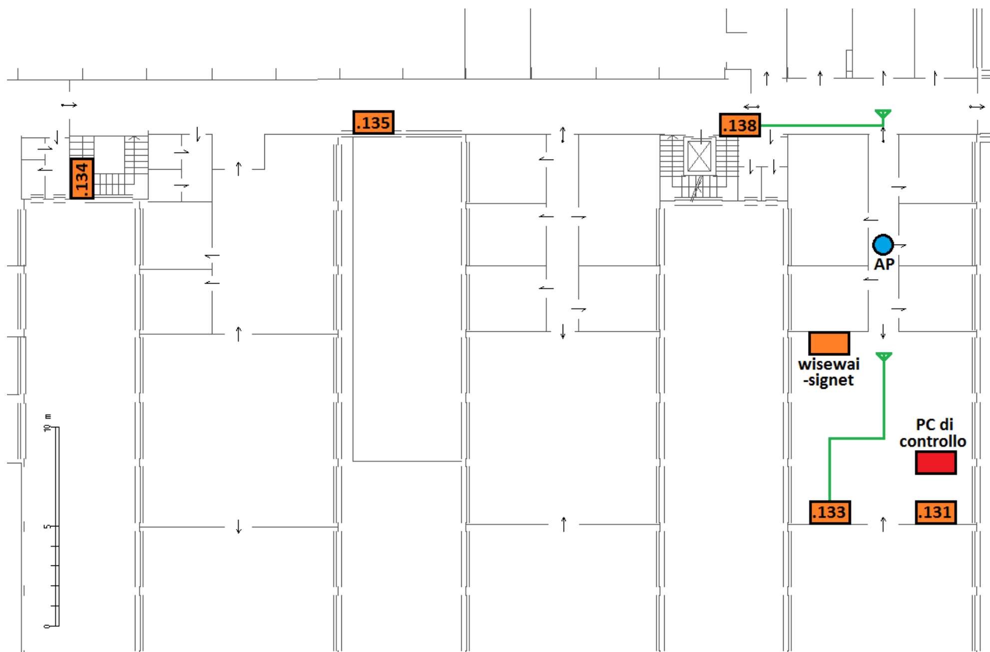


Figura 2.8: Mappa del testbed.

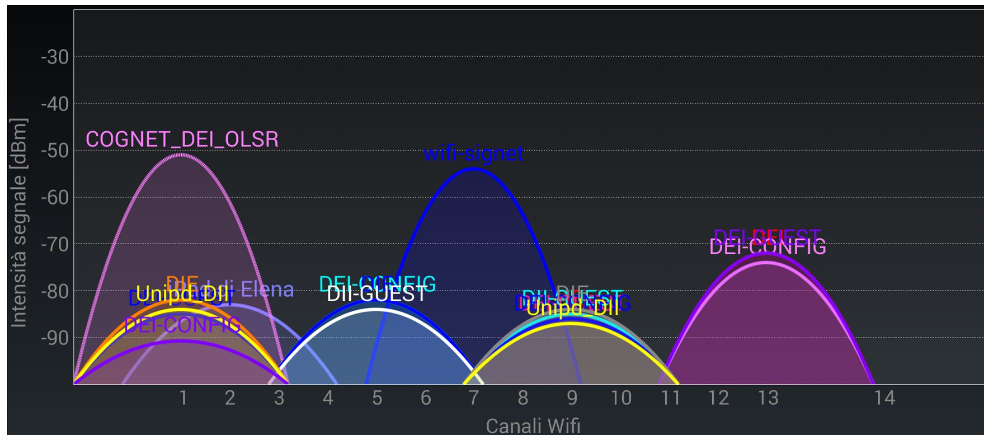


Figura 2.9: Situazione dell'occupazione dei canali Wi-Fi nella zona del testbed.

OLSR Routes in Kernel				
Destination	**Gateway**	**Metric**	**ETX**	**Interface**
192.168.1.133	192.168.1.133	1	1.000	wlan0
192.168.1.134	192.168.1.135	2	2.186	wlan0
192.168.1.135	192.168.1.135	1	1.186	wlan0
192.168.1.138	192.168.1.138	1	1.000	wlan0

Figura 2.10: Tabella di *routing* del nodo .131 usando in ogni nodo la potenza di trasmissione massima.

sempre così favorevoli, si è deciso di abbassare le potenze di trasmissione secondo la Tabella 2.3. Come si può notare, è stata impostata una potenza di trasmissione molto alta nel nodo .133, questo sia per sovrastare le trasmissioni del AP dipartimentale, sia perché spesso i nodi .131 e .138 riuscivano a comunicare in 1 hop; per "suggerire" quindi ai protocolli di *routing* di preferire i 2 hop è necessario che i 2 canali siano estremamente migliori del canale singolo. Dopo le modifiche imposte alle potenze di trasmissione la situazione è quella desiderata, come illustrato in Figura 2.11.

2.4.1 Server di controllo

Oltre ai nodi che formano la rete *mesh*, il *testbed* necessita di due server di controllo: *wisewai-server* e *wisewai-signet*. I nodi ALIX infatti caricano al *boot* il kernel Linux dalla rete tramite PXE (*Preboot Execution Environment*) e in particolare da *wisewai-signet* che è anch'esso un nodo ALIX. Tutta la parte di sistema operativo che sta

Tabella 2.3: Potenze di trasmissione imposte ai nodi.

nodo	potenza di trasmissione
.131	0 dBm
.133	25 dBm
.138	5 dBm
.135	10 dBm
.134	10 dBm

OLSR Routes in Kernel				
Destination	Gateway	Metric	ETX	Interface
192.168.1.133	192.168.1.133	1	1.000	wlan0
192.168.1.134	192.168.1.133	4	4.925	wlan0
192.168.1.135	192.168.1.133	3	3.723	wlan0
192.168.1.138	192.168.1.133	2	2.723	wlan0

Figura 2.11: Tabella di *routing* del nodo .131 usando in ogni nodo una potenza di trasmissione ridotta.

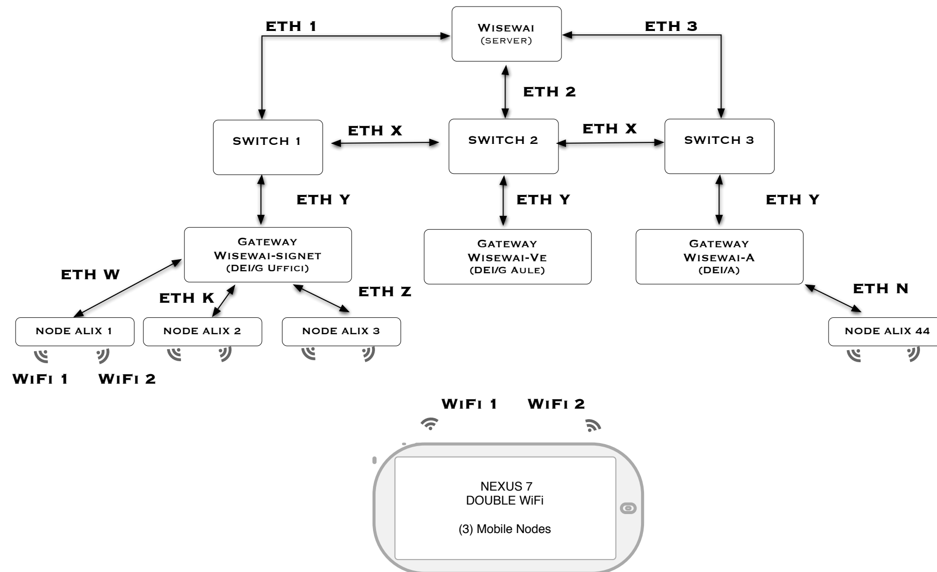


Figura 2.12: Schema completo del testbed.

sopra al kernel (*software*, dati, ...) viene invece caricata e mantenuta sincronizzata con *wisewai-server* tramite *TFTPboot*: nel server è presente una *directory* per ogni ALIX in cui è contenuto l'intero sistema operativo e i dati. In realtà potrebbe fare tutto il lavoro *wisewai-server*, *wisewai-signet* serve solo per segmentare la rete di ALIX disseminata in tutto il dipartimento che è formata da altre due sezioni oltre a quella utilizzata per questa tesi. Uno schema della rete completa è osservabile in Figura 2.12, in cui i 5 ALIX utilizzati fanno parte di quelli che fanno capo a *wisewai-signet* e per ora sono equipaggiati solo con una scheda Wi-Fi. *Wisewai-server* ha inoltre una funzione di controllo su tutti gli ALIX: essendo alimentati da PoE (*Power over Ethernet*), il server può togliere l'alimentazione ad ogni nodo singolarmente tramite il comando `gwreset [numero del nodo senza centinaia]`.

2.5 Software

Ogni nodo necessita di diversi *software* per poter creare e gestire il *testbed*. Verranno ora descritti gli strumenti utilizzati e i *software* creati appositamente.

Tabella 2.4: Parametri TCP e MAC osservabili e modificabili da CogNet per diversi dispositivi.

Observable/Writable	Layer	Parameters	Laptop b43 (bdcm4331)	Alix 3d2 ath5kl (ar5413)	Tab-7 P6210 ath6kl (ar6003)	Nexus 7 bdcmhd (bdcm4330)	USB Adapter ath9k_htc (AR9271)
Observable Writable	TCP	CWND	r/w	r/w	r/w	r/w	r/w
	TCP	SSTHR	r/w	r/w	r/w	r/w	r/w
	TCP	TCP CLAMP	r/w	r/w	r/w	r/w	r/w
	TCP	RTO	r/w	r/w	r/w	r/w	r/w
	MAC	CWmin	r/w	r/w	NO	NO	r/w
	MAC	CWmax	r/w	r/w	NO	NO	r/w
	MAC	AIFS	r/w	r/w	NO	NO	r/w
	MAC	TX power	r/w	r/w	r	r/w	r/w
Observable	TCP	IP address	r	r	r	r	r
	TCP	Port	r	r	r	r	r
	TCP	# Lost packets	r	r	r	r	r
	TCP	# Packets in flight	r	r	r	r	r
	TCP	# Packets outstanding	r	r	r	r	r
	TCP	# Packets acked	r	r	r	r	r
	TCP	# Packets lost	r	r	r	r	r
	TCP	Throughput	r	r	r	r	r
	TCP	RTTvar	r	r	r	r	r
	TCP	SRTT	r	r	r	r	r
	MAC	Transmission Channel	r	r	r	r	r
	MAC	RSSI	r	r	r	r	r
	MAC	Bytes RX	r	r	r	r	r
	MAC	# Frames RX	r	r	r	r	r
	MAC	# Frames RX duplicate	r	r	r	NO	r
	MAC	# Frames RX fragments	r	r	r	NO	r
	MAC	# Frames RX dropped	r	r	NO	NO	r
	MAC	Bytes TX	r	r	r	r	r
	MAC	# Frames TX	r	r	r	r	r
	MAC	# Fragments TX	r	r	r	r	r
	MAC	# Frames TX retry count	r	r	r	NO	r
	MAC	# Frames TX retry failed	r	r	NO	NO	r
	MAC	Inactive station	r	r	NO	NO	r

r = observable, w = writable, NO = to be implemented, TX= transmitted, RX= received, # = number.

2.5.1 CogNet

Il *software* CogNet, creato dal PhD Matteo Danieletto, serve per registrare ed eventualmente modificare quasi tutti i parametri del protocollo TCP e del livello MAC di un nodo con kernel Linux. È composto da due *thread user space* uno che legge o modifica i parametri TCP e uno per il livello MAC. Il *thread* TCP comunica con il modulo kernel CogNet, che modifica leggermente la struttura `tcp_sock`, codice C che implementa TCP in una macchina Linux, per permettere la lettura e la modifica dei parametri. La lettura avviene ogniqualvolta viene ricevuto un ACK TCP. Il *thread user space* che si occupa del livello MAC si interfaccia direttamente con il *driver* della scheda *wireless*; quanti e quali parametri sono modificabili dipende quindi molto dal *driver*. L'elenco dei parametri osservabili e modificabili in diversi dispositivi da CogNet è riassunto in Tabella 2.4.

2.5.2 SSH

SSH (*Secure SHell*) è un protocollo di rete che permette di stabilire una sessione remota cifrata tramite interfaccia a riga di comando con un altro *host* di una rete informatica. È il protocollo che ha sostituito l'analogo, ma insicuro, Telnet.

In ogni macchina Linux è installabile l'implementazione *open source* OpenSSH. Il comando per aprire una sessione SSH è `ssh utente@host`, esiste però una versione che

permette di eseguire sulla macchina remota un singolo comando: `ssh utente@host [comando]`; in questo caso se il comando da eseguire fa partire un processo che non termina immediatamente, torna utile l'opzione `-f` che non fa bloccare la *shell* aspettando la terminazione del programma.

Normalmente l'autenticazione prevede l'inserimento di una *password*; è invece più comodo negli *script* per il controllo remoto di cui si parlerà in seguito, utilizzare l'autenticazione con chiave pubblica e privata. Questo tipo di autenticazione prevede la creazione di una chiave privata dell'utente *client* tramite il comando `ssh key-gen` (se non si vuole inserire nemmeno una *passphrase* durante l'autenticazione, questa va lasciata vuota); il comando crea anche la chiave pubblica corrispondente da inviare al *server* tramite il comando `ssh-copy-id -i  /.ssh/id_rsa.pub utente@server`. Dopo aver controllato che la chiave pubblica del *client* sia stata aggiunta al file `/.ssh/authorized_keys` del *server* e aver controllato che sia abilitata la modalità chiave pubblica/privata nel file di configurazione del *server* `/etc/ssh/sshd_config`, sarà possibile aprire sessioni SSH senza inserire *password*.

Tutti i nodi ALIX del *testbed* sono raggiungibili tramite SSH attraverso la rete cablata *ethernet*.

2.5.3 NTP

NTP (*Network Time Protocol*) è un protocollo per sincronizzare gli orologi dei computer all'interno di una rete a commutazione di pacchetto, quindi con tempi di latenza variabili ed inaffidabili. NTP è un protocollo client-server, il server resta in ascolto sulla porta UDP 123. Il funzionamento si basa sul rilevamento dei tempi di latenza nel transito dei pacchetti sulla rete. Utilizza il tempo coordinato universale ed è quindi indipendente dai fusi orari. Attualmente è in grado di sincronizzare gli orologi dei computer su internet entro un margine di 10 ms e con una accuratezza di almeno 200 μs all'interno di una LAN in condizioni ottimali. I diversi server NTP sono organizzati in una struttura gerarchica a strati, dove lo strato 1 è sincronizzato con una fonte temporale esterna quale un orologio atomico, GPS o un orologio radiocontrollato, lo strato 2 riceve i dati temporali dai server di strato 1, e così via. Un server si sincronizza confrontando il suo orologio con quello di diversi altri server di strato superiore o dello stesso strato. Questo permette di aumentare la precisione e di eliminare eventuali server scorretti. Un server NTP è in grado di stimare e compensare gli errori sistematici dell'orologio *hardware* di sistema, che normalmente è di scarsa qualità.

In una macchina Linux l'implementazione di NTP più comune è il demone `ntpd` che fa riferimento al suo file di configurazione `/etc/ntpd.conf`. `ntpd` funge sia da client che da server: è in grado di sincronizzarsi con server di livello superiore (i server NTP nazionali disponibili in internet) e di aprire un server in ascolto sulla porta UDP 123 per sicronizzare altri computer che ne facessero a lui richiesta. `ntpd` è in grado di stimare e correggere l'errore sistematico del *clock* di sistema, evitando un andamento irregolare del tempo e migliorando la precisione quando il computer non è connesso alla rete.

Nel testbed si è deciso di creare un server NTP su un computer della rete dipartimentale con accesso ad internet, accessibile da tutti gli ALIX. I nodi ALIX non hanno accesso diretto ad internet, quindi si sicronizzeranno facendo richiesta al server creato. L'installazione di un server NTP non è un operazione difficile, basta installare il pacchetto ntp con il comando `apt-get install ntp` e modificare correttamente il file di configurazione inserendo i server NTP italiani. Per la sincronizzazione degli orologi degli ALIX invece, si è scelto di non usare il demone `ntpd` per non appesantire troppo

le mecchine: si è deciso di ricorrere al comando `ntpdate` che esegue la sincronizzazione verso il server (passato come argomento) una volta sola, non costantemente. La sincronizzazione degli orologi negli ALIX tuttavia, non è un'operazione così facile: la batteria tampone di tutti gli ALIX è scarica, quindi ad ogni riavvio la data del SO veniva risettata al 2006 leggendo l'orologio *hardware* (un contatore elettronico che prosegue la sua corsa anche con la macchina spenta usufruendo della batteria tampone). Il problema è che uno *shift* di data di grande entità provoca una grande instabilità nel sistema: molte parti del sistema vanno in *crash*, compreso SSH. Lo *shift* massimo consigliato è 30 minuti, è stato necessario dunque trovare un sistema per settare l'orologio *hardware* ad una data e ora vicine a quelle reali, prima della lettura da parte dell'orologio del SO, in maniera tale che, dopo l'avvio, si possa eseguire la sincronizzazione con il server NTP senza creare instabilità nel sistema. La soluzione adottata è stata quella di inserire in `crontab` (processo in cui si possono inserire delle operazioni da eseguire con una certa cadenza) un comando che va a modificare una riga aggiunta al file `/etc/init.d/hwclock` (uno dei primi script eseguiti al boot) che contiene un comando che modifica l'ora *hardware* impostandola ad un valore passato come argomento: ogni quarto d'ora questa riga viene modificata scrivendo come argomento del comando la data e l'ora attuali. Ad ogni riavvio l'orologio *hardware* sarà impostato ad un'ora distante da quella reale al massimo un quarto d'ora, l'orologio del SO verrà impostato leggendo quell'ora, e sarà quindi possibile eseguire la sincronizzazione col server NTP. Per comodità, tutte queste operazioni sono state eseguite su *wisewai-server*: il *crontab* è quello del server e va a modificare i file `hwclock` di ogni ALIX nelle cartelle che ridondano i loro sistemi operativi.

2.5.4 Iptables

Iptables è un'applicazione *user space* che permette ad un utente *root* di configurare le tabelle messe a disposizione dal *firewall* implementato nel kernel Linux e in particolare di gestire le catene e le regole che esse contengono. Esistono tre catene principali: `INPUT`, `OUTPUT` e `FORWARD`. Tutti i pacchetti generati dal sistema e in uscita da esso passano attraverso la catena `OUTPUT`, tutti i pacchetti in entrata e destinati al sistema stesso passano attraverso la catena `INPUT`, mentre i pacchetti che transitano per il sistema ma sono destinati ad altri sistemi passano dalla catena `FORWARD`. Le catene sono utilizzate per indicare una serie di regole. Un pacchetto viene comparato con ogni regola di una catena, se non ne soddisfa alcuna, ad esso viene applicata la `destination` predefinita per la catena. Per *destination* si intende l'azione da intraprendere sul pacchetto quando questo soddisfa una certa regola; le `destination` più comuni sono `ACCEPT`, `DROP` (scarta silenziosamente il pacchetto), `REJECT` (scarta il pacchetto ma comunica l'evento ai livelli superiori) e `LOG`.

Il comando per definire la `destination` predefinita per una catena è `iptables -P [catena] [destination]`. Per definire invece una regola che scarti tutti i pacchetti provenienti da un certo indirizzo MAC (caso che sarà utile per gli esperimenti) il comando è `iptables -A INPUT -m mac -mac-source [MAC] -j DROP`; l'opzione `-A` serve ad indicare la catena, l'opzione `-j` serve ad indicare la `destination`.

2.5.5 Iperf

Iperf è uno strumento molto usato che misura il *throughput* raggiungibile tra due *host* di una rete, creando uno *stream* di dati TCP o UDP. Permette di variare diversi parametri della connessione per testare ed eventualmente ottimizzare la rete. Il modo

più semplice per effettuare una misura con iperf è lanciare sul server il comando `iperf -s` e sul client il comando `iperf -c [IP del server]`. Esiste anche una versione del comando che effettua una misura che dura un tempo specificato da linea di comando. Questa è la versione che verrà utilizzata negli esperimenti per avere delle medie più vicine alla realtà, essendo state calcolate su esperimenti durati un tempo adeguato a rendere trascurabili i fenomeni isolati.

2.5.6 Script Bash

Per l'inizializzazione e la gestione da remoto di tutta la rete *mesh* sono stati creati diversi script Bash.

Bash (*Bourne Again SHell*) è una *shell* testuale del progetto GNU usata nei sistemi operativi Unix e Unix-like, specialmente in GNU/Linux. Tecnicamente Bash è un clone evoluto della *shell* standard di Unix (`/bin/sh`). Si tratta di un interprete di comandi che permette all'utente di comunicare col sistema operativo attraverso una serie di funzioni predefinite, o di eseguire programmi. Bash mette a disposizione un semplice linguaggio di *scripting* che permette di svolgere i compiti più complessi, non solo raccogliendo in uno script una serie di comandi, ma anche utilizzando variabili, funzioni e strutture di controllo di flusso.

`rc.local` e script di inizializzazione

`rc.local` è l'ultimo script chiamato durante il *boot* del sistema, di default è vuoto ma si può riempire di comandi che devono essere eseguiti all'avvio.

Codice 2.1: Script `rc.local` di un nodo ALIX.

```
1  #!/bin/sh -e
2
3  # rc.local
4
5  # RP=0 --> STATIC routing
6  # RP=1 --> RoutingProtocol=OLSR
7  # RP=2 --> RoutingProtocol=BATMAN
8
9  RP=1
10
11 if [ $RP -eq 0 ]; then
12     sh /etc/init.d/STATIC_init_ALIX.sh
13 elif [ $RP -eq 1 ]; then
14     sh /etc/init.d/launcher_ALIX.sh
15 elif [ $RP -eq 2 ]; then
16     sh /etc/init.d/BATMAN_init_ALIX.sh
17 fi
18
19 ntpdate 147.162.97.20 &
20
21 exit 0
```

Come si può leggere nel Codice 2.1, nel caso dei nodi ALIX da questo script si può scegliere quale protocollo di *routing* far partire modificando la variabile `RP` (*Routing*

Protocol). Cambiando il valore della variabile si sceglie quale script di avvio, contenuto nella cartella `/etc/init.d/`, lanciare. Al termine dello script viene eseguito il comando che esegue la sincronizzazione dell'orologio con il server NTP.

Ogni script di avvio si deve occupare di diverse cose:

1. impostare la modalità *ad-hoc* e il nome della rete sulla scheda *wireless*;

```
ifconfig wlan0 down
iwconfig wlan0 mode ad-hoc
iwconfig wlan0 essid COGNET_DEI_BATMAN
ifconfig wlan0 up
```

2. occuparsi del *routing* facendo partire l'eventuale protocollo;

3. assegnare un indirizzo IP univoco all'interfaccia di rete *wireless* utilizzata, copiando l'ultimo byte dell'indirizzo IP dell'interfaccia *ethernet*; nel caso del protocollo BATMAN l'indirizzo IP va assegnato all'interfaccia virtuale `bat0` e non a `wlan0`;

```
# Assign the ethernet IP address to the variable IP
IP=$(ip addr show dev eth0 | awk '$1=="inet" {print $2}')
# Assing the IP address last byte to the variable LASTDIGIT
LASTDIGIT=$(echo ${IP} | awk 'BEGIN {FS="/"}; {print $1}' |
    awk 'BEGIN {FS="."}; {print $4}' )
# Assign the correct IP address to the wireless interface (
    wlan0 or bat0)
ifconfig wlan0 192.168.1.${LASTDIGIT}
```

4. caricare il modulo CogNet;

```
# Insert COGNET module
sysctl -w net.ipv4.tcp_abc=1
insmod /root/COGNET_TESTBED/KERNEL_MODULE/durip_alix.ko
    checkPrint=0
TCC=net.ipv4.tcp_congestion_control
sysctl $TCC
sysctl -w $TCC=DURIP
mount -t debugfs none /sys/kernel/debug
```

5. far partire il server iperf.

```
# Starting iperf server
iperf -s > /dev/null
```

Per quanto riguarda il *routing* i comandi da eseguire cambiano a seconda della scelta effettuata in `rc.local`. Nel caso si sia scelta la modalità statica lo script deve utilizzare iptables per scartare i pacchetti che arrivano da nodi non vicini e impostare le rotte statiche; ad esempio per il nodo .138 (quello centrale) il codice è:

```
# This node is 10.1.129.138 --> accept .133 and .135
# Reject .131
iptables -A INPUT -m mac --mac-source 00:0D:B9:0D:1E:68 -j DROP
# Reject .134
iptables -A INPUT -m mac --mac-source 00:0D:B9:19:15:90 -j DROP
```

```
6 # Add static routes
7 route add -host 192.168.1.131 gw 192.168.1.133 dev wlan0
8 route add -host 192.168.1.134 gw 192.168.1.135 dev wlan0
```

Il comando che fa partire il protocollo di *routing* OLSR, deve indicare il file di configurazione con l'opzione `-f` e il livello di *logging* con l'opzione `-d`:

```
1 /root/COGNET_TESTBED/olsrdAlix -f /root/COGNET_TESTBED/CONFIG/
    olsrdAlix.conf -d 0 > /dev/null
```

Nel caso del protocollo BATMAN, è necessario caricare il modulo e comunicargli le interfacce su cui abilitare il protocollo:

```
1 # Insert modules
2 insmod lib/modules/3.8.11/kernel/lib/libcrc32c.ko
3 insmod lib/modules/3.8.11/kernel/net/batman-adv/batman-adv.ko
4 # Enable BATMAN on wlan0 and call the virtual interface bat0
5 echo bat0 > /sys/class/net/wlan0/batman_adv/mesh_iface
```

`iptables_script`

`iptables_script` è uno script personalizzato per ogni ALIX che imposta delle regole con iptables in modo da scartare tutti i pacchetti che non provengono da nodi vicini (controllando l'indirizzo MAC). Questo è utile per forzare il percorso che un pacchetto deve seguire anche nel caso in cui sia attivo un protocollo di routing. Le regole imposte sono le stesse dello script di inizializzazione del *routing* statico: in quel caso i nodi partono già con queste regole imposte, questo script è utile nel caso in cui si sia fatto partire un protocollo di *routing*.

`batctl_script` e `batctl_tr_script`

`batctl_script` e `batctl_tr_script` sono due script che servono a salvare su file rispettivamente le tabelle di *routing* del protocollo BATMAN e il *traceroute* verso una certa destinazione. Come già spiegato, la scelta di implementare BATMAN a livello 2 rende inutile l'utilizzo dei comandi `route` e `traceroute`, è necessario utilizzare i rispettivi comandi `batctl o` e `batctl tr [MAC destinazione]`. I due script eseguono i rispettivi comandi periodicamente, il periodo di campionamento è scelto tramite il primo argomento. Il secondo argomento permette di specificare il nome del file di output. Lo script è essenzialmente un ciclo *while* che per decidere quando reiterare e quando uscire dal ciclo, legge il contenuto di un file che lo script stesso crea: `batctl_script_state` (o `batctl_tr_script_state`). Lo stesso file può essere letto per sapere se lo script sia in esecuzione o meno. Per bloccare lo script basta scrivere "stop" nel file di controllo. `batctl_script`, ripulito della parte di verifica degli argomenti e di qualche riga non essenziale è leggibile in Codice 2.2. Il codice di `batctl_tr_script` è pressoché identico, tranne per il fatto che esegue il comando `batctl tr` e che necessita di un terzo argomento che indica l'indirizzo MAC del destinatario di *traceroute*.

`command_to_mesh` e script di controllo remoto

Per non dover entrare in ogni nodo ALIX singolarmente e replicare in ognuno di essi le stesse operazioni, impiegando così più di mezz'ora per la preparazione di un

Codice 2.2: Script `batctl_script`.

```
#!/bin/bash

# This script continuously calls "batctl o" every (ARG 1) s and
    saves the routing table in file (ARG 2).

touch /root/COGNET_TESTBED/batctl_script_RES/$2
echo > /root/COGNET_TESTBED/batctl_script_RES/$2

touch /root/COGNET_TESTBED/batctl_script_state
echo "go" >  /root/COGNET_TESTBED/batctl_script_state

while [ "$(cat /root/COGNET_TESTBED/batctl_script_state)" == "go"
     ]
do
   echo "#" >> /root/COGNET_TESTBED/batctl_script_RES/$2
   date >> /root/COGNET_TESTBED/batctl_script_RES/$2
   batctl o >> /root/COGNET_TESTBED/batctl_script_RES/$2
   sleep $1
done
echo "------------ END -----------" >> /root/COGNET_TESTBED/
    batctl_script_RES/$2

exit 0
```

esperimento, si è reso necessario creare degli script per il controllo remoto. Dopo la creazione di questi script è stato possibile avviarne uno da un singolo PC di controllo che abbia accesso a tutti i nodi, affinché l'operazione venga eseguita su tutti gli ALIX della rete *mesh*. Lo script più generico possibile è `command_to_mesh` che fa eseguire un singolo comando (dato come argomento) ad ogni nodo della rete. Esso fa uso della versione di SSH che trasmette un singolo comando, come si può leggere in Codice 2.3 (ripulito del controllo degli argomenti). Lo script esegue il comando su un nodo alla volta per permettere la visualizzazione dell'eventuale output. Un esempio di output dello script è mostrato in Figura 2.13, nel quale era stato inviato il comando `date`; come si può osservare non viene richiesta la *password* SSH grazie alla modalità chiave pubblica/privata, fatta eccezione per il nodo .131 in cui questa modalità non è stata attivata a titolo esemplificativo. Altri utili comandi che si possono inviare comodamente tramite `command_to_mesh` sono:

`reboot` per riavviare tutti i nodi (ad esempio dopo aver cambiato la variabile `RP` in tutti gli `rc.local` per far partire un diverso protocollo di *routing*);

`iwconfig` per poter visualizzare in un unico posto le situazioni delle schede *wireless* di tutti i nodi (potenza, modalità, ssid, . . .);

`ntpdate 147.162.97.20` per far sincronizzare tutti i nodi col server NTP;

`/root/COGNET_TESTBED/iptables_script.sh` per eseguire `iptables_script` su ogni ALIX, quindi limitare la libertà dei protocolli di *routing*.

Codice 2.3: Script `command_to_mesh`.

```
#!/bin/bash

# This script send a command (ARG 1) to all the ALIX in the mesh
    network

echo "10.1.129.131: executing '$1' ..."
ssh -f root@10.1.129.131 "$1" &&
sleep 2
echo "10.1.129.133: executing '$1' ..."
ssh -f root@10.1.129.133 "$1" &&
sleep 2
echo "10.1.129.138: executing '$1' ..."
ssh -f root@10.1.129.138 "$1" &&
sleep 2
echo "10.1.129.135: executing '$1' ..."
ssh -f root@10.1.129.135 "$1" &&
sleep 2
echo "10.1.129.134: executing '$1' ..."
ssh -f root@10.1.129.134 "$1" &&
sleep 2

echo "All commands executed."

exit 0
```

Per trasmettere comandi in cui sono presenti degli spazi, è opportuno racchiudere tutta la stringa con degli apici `' '`. In realtà è possibile anche trasmettere più di un comando da eseguire uno dopo l'altro su ogni nodo, separandoli con `&&`.

Sono stati creati altri script per il controllo remoto, più specifici e che richiedono più argomenti:

`command_batctl_script` fa partire o fermare in ogni nodo `batctl_script`; accetta come argomenti `start` o `stop`, nel caso di `start` sono necessari altri 2 argomenti: il periodo di campionamento e il nome del file di output;

`command_COGNET` fa partire e inizializza il software CogNet in ogni nodo; accetta come argomenti `start` o `stop`, nel caso di `start` sono necessari altri 4 argomenti: numero di porta, periodo di campionamento del livello MAC (in ms), nome della cartella di output, indirizzo IP della rete;

`command_OLSR` fa partire OLSR in ogni nodo; accetta come argomenti `start` o `stop`, nel caso di `start` è necessario un ulteriore argomento: il livello di log (1 se si vogliono creare i file di log, 0 altrimenti);

`txpower_script` imposta le corrette potenze di trasmissione nelle schede *wireless* di ogni nodo;

`scriptIperf` e `commandNodes_Iperf`

`scriptIperf` è uno script che fa partire una misurazione con `iperf` della durata di un tempo impostato da argomento e la ripete un certo numero di volte, attendendo un

```
zanellaf@pcsensori09: ~/Codici
zanellaf@pcsensori09:~/Codici$ ./command_to_mesh.sh 'date'
10.1.129.131: executing 'date' ...
root@10.1.129.131's password:
Fri Apr  4 13:13:03 CEST 2014
10.1.129.133: executing 'date' ...
Fri Apr  4 13:13:15 CEST 2014
10.1.129.138: executing 'date' ...
Fri Apr  4 13:13:28 CEST 2014
10.1.129.135: executing 'date' ...
Fri Apr  4 13:13:40 CEST 2014
10.1.129.134: executing 'date' ...
Fri Apr  4 13:13:53 CEST 2014
All commands executed.
zanellaf@pcsensori09:~/Codici$
```

Figura 2.13: Output dello script `command_to_mesh` trasmettendo il comando `date`.

Codice 2.4: Script `commandNodes_Iperf`.

```
1 #!/bin/bash
2
3 # scriptIperf arguments: numberOfInterations, port, fileIP,
    nameFileBandwidth, sleepingTimeExperiment, waitTimeThread,
    IPiperf, iperfDuration, intervalTime.
4
5 sysctl -w net.ipv4.tcp_abc=1
6 ./scriptIperf.sh 5 4000 fileIPAlix STATIC 5 10 192.168.1.134 600
    10
7
8 exit 0
```

dato periodo di tempo tra una misura e l'altra per lasciar decadere tutte le ritrasmissioni dei pacchetti della misura precedente. Prima di ogni misura, lo script comunica al software CogNet (precedentemente avviato e inizializzato con `command_COGNET`) di avviare la registrazione dei parametri TCP e MAC. Lo script salva tutti i risultati in diversi file (uno per misura). `scriptIperf` richiede parecchi argomenti nel seguente ordine: numero di iterazioni, porta, file contenente tutti gli indirizzi IP dei nodi, prefisso dei file di output, ??, ??, IP destinazione di iperf, durata di iperf, tempo di attesa tra una misura e l'altra.

Per evitare di scrivere ogni volta tutti gli argomenti di `scriptIperf` è stato creato `commandNodes_Iperf` che chiama semplicemente `scriptIperf` ma permette, modificandolo, di cambiare solo alcuni argomenti che interessano l'esperimento, senza dover ricordare tutti gli altri, come si può osservare in Codice 2.4.

`save_res`

`save_res` è uno script che va lanciato su *wisewai-server* che raccoglie tutti i dati dell'esperimento appena concluso da tutti gli ALIX e li sposta in un unico posto organizzato in una struttura di *directory* adeguata. Lo script raccoglie i file di output di CogNet, i file di output di iperf e gli eventuali file di OLSR (file di log) o BATMAN (file di output di `batctl_script` e `batctl_tr_script`). Richiede i seguenti argomenti:

il nome scelto per la cartella di output in `command_COGNET`, il prefisso dei file di output scelto in `commandNodes_Iperf`, il suffisso che identifica l'esperimento e un eventuale stringa "OLSR" o "BATMAN" che indica allo script se è necessario prelevare anche i dati dei due protocolli di *routing*.

Capitolo 3

Esperimenti

Sono stati eseguiti diversi esperimenti sul *testbed* composto dai soli nodi fissi, per confrontare i diversi protocolli di *routing*. Oltre agli esperimenti con OLSR e BATMAN in azione sulla rete *mesh*, sono stati eseguiti esperimenti sulla rete costruita con rotte statiche per poter avere una base con cui poi confrontare l'*overhead* di traffico introdotto dai protocolli.

Gli esperimenti constano in 5 trasmissioni eseguite con `iperf` della durata di 10 minuti l'una. Le trasmissioni partono sempre dal nodo .131 e hanno come destinazione uno degli altri nodi: per ognuna delle situazioni analizzate si sono eseguite trasmissioni verso il nodo .134 su un percorso potenzialmente da 4 hop, verso il nodo .135 su un percorso potenzialmente da 3 hop, verso il nodo .138 su un percorso potenzialmente da 2 hop e verso il nodo .133 su un percorso da 1 hop.

Le situazioni analizzate sono:

- *routing* statico (STATIC);
- protocollo OLSR in evoluzione libera (OLSR_FREE);
- protocollo OLSR con rotte imposte (OLSR_BLKD);
- protocollo BATMAN in evoluzione libera (BATMAN);

Nel *routing* statico e in OLSR con rotte bloccate è stato usato iptables per scartare tutti i pacchetti che non provengono da nodi vicini (controllando l'indirizzo MAC): in questo modo è stato imposto ai pacchetti il percorso da seguire facendo in modo che effettuino tutti gli hop possibili; ad esempio un pacchetto di una trasmissione dal nodo .131 al nodo .134 farà sicuramente tutti e 4 gli hop senza saltare alcun nodo. Se in qualche momento uno dei link fosse rotto, il pacchetto non arriverebbe a destinazione finché il link non ritornasse attivo. Questo tipo di esperimenti è stato eseguito nell'ottica di misurare l'*overhead* di traffico introdotto dai protocolli; purtroppo per il protocollo BATMAN non è stato possibile effettuare queste misure per colpa della già discussa inefficacia di iptables.

Negli altri due casi invece, i protocolli di *routing* vengono lasciati liberi: in qualche momento il confronto delle metriche potrebbe preferire un percorso con un numero di hop minore di quello "nominale" verso una certa destinazione. Ad esempio un pacchetto di una trasmissione dal nodo .131 al nodo .134 potrebbe non fare 4 hop ma 3 saltando uno dei nodi intermedi; questo succede quando la metrica di un link che salta un nodo, pur essendo questo link più lungo (e quindi che attenua di più la potenza), risulta

migliore della combinazione delle due metriche dei 2 link più corti. Si deve pensare che qualsiasi protocollo di *routing*, anche adottando una metrica che non sia il semplice numero di hop, in qualche modo predilige percorsi più corti (in termini di numero di hop); ad esempio con la metrica ETX di OLSR, la combinazione delle ETX di due link successivi è la somma delle stesse, quindi non sarà così facile che la somma di due ETX sia minore di una singola ETX; questo succederà solo quando l'ETX del link diretto (1 hop) sarà molto alta, quindi con un link pessimo (ad esempio se il link è molto lungo e quindi la potenza viene molto attenuata). L'effetto più negativo di questi cambi di *path* (che avvengono spontaneamente nel tempo) è che il TCP viene momentaneamente bloccato, riducendo di il *throughput* complessivo. Gli esperimenti effettuati puntano ad analizzare queste situazioni confrontando i diversi protocolli.

La procedura per iniziare un esperimento in modalità *routing* statico prevede, dopo aver modificato il file `rc.local` di ogni ALIX scegliendo il valore 0 per la variabile `RP`, l'esecuzione da un PC di controllo e da una *directory* in cui siano presenti tutti gli script di controllo remoto, dei seguenti comandi:

```
1  # riavviare i nodi in modalita' routing statico:
2  ./command_to_mesh.sh "reboot"
3  # sincronizzare gli orologi tramite il server NTP:
4  ./command_to_mesh.sh "ntpdate 147.162.97.20"
5  # impostare le potenze:
6  ./txpower_script.sh
7  # avviare il software CogNet:
8  ./command_COGNET.sh start 4000 500 TEST_STATIC 192.168.1.0
9  # DAL NODO .131 in /root/MANAGER:
10 # controllare che in "fileIPAlix" siano presenti tutti gli
       indirizzi IP dei nodi della rete;
11 # modificare commandNode_Iperf.sh impostando la destinazione di
       iperf e il prefisso "STATIC":
12 vi commandNode_Iperf.sh
13 # avviare l'esperimento:
14 ./commandNode_Iperf.sh
15
16 # Al termine dell'esperimento:
17 # fermare CogNet:
18 ./command_COGNET.sh stop
19 # SU wisewai-server: salvare i risultati impostando un suffisso
       che identifichi l'esperimento:
20 ./save_res.sh TEST_STATIC STATIC [X]_HOP
```

La procedura per iniziare invece un esperimento con il protocollo OLSR prevede, dopo aver modificato il file `rc.local` di ogni ALIX scegliendo il valore 1 per la variabile `RP`, l'esecuzione dei seguenti comandi:

```
1  # riavviare i nodi con OLSR:
2  ./command_to_mesh.sh "reboot"
3  # bloccare il processo OLSR iniziato all'avvio senza log
       specifici:
4  ./command_OLSR.sh stop
5  # rimuovere i pochi file di log creati all'avvio da OLSR:
6  ./command_to_mesh.sh "rm -rf /mnt/local/log/OLSRD/*"
7  # sincronizzare gli orologi tramite il server NTP:
8  ./command_to_mesh.sh "ntpdate 147.162.97.20"
9  # settare le potenze:
```

```
10 ./txpower_script.sh
11 # avviare OLSR con log:
12 ./command_OLSR.sh start 1
13 # avviare il software CogNet:
14 ./command_COGNET.sh start 4000 500 TEST_OLSR 192.168.1.0
15 # DAL NODO .131 in /root/MANAGER:
16 # controllare che in "fileIPAlix" siano presenti tutti gli
       indirizzi IP dei nodi della rete;
17 # modificare commandNode_Iperf.sh impostando la destinazione di
       iperf e il prefisso "OLSR":
18 vi commandNode_Iperf.sh
19 # avviare l'esperimento:
20 ./commandNode_Iperf.sh
21
22 # Al termine dell'esperimento:
23 # fermare CogNet (per il rilascio dei file):
24 ./command_COGNET.sh stop
25 # fermare OLSR (per il rilascio dei file):
26 ./command_OLSR.sh stop
27 # SU wisewai-server: salvare i risultati impostando un suffisso
       che identifichi l'esperimento:
28 ./save_res.sh TEST_OLSR OLSR [X]_HOP OLSR
```

La procedura per iniziare infine un esperimento con il protocollo BATMAN prevede, dopo aver modificato il file `rc.local` di ogni ALIX scegliendo il valore 2 per la variabile `RP`, l'esecuzione dei seguenti comandi:

```
1 # riavviare i nodi con BATMAN:
2 ./command_to_mesh.sh "reboot"
3 # sincronizzare gli orologi tramite il server NTP:
4 ./command_to_mesh.sh "ntpdate 147.162.97.20"
5 # settare le potenze:
6 ./txpower_script.sh
7 # avviare il software CogNet:
8 ./command_COGNET.sh start 4000 500 TEST_OLSR 192.168.1.0
9 # avviare in ogni ALIX batctl_script per registrare le tabelle di
       routing:
10 ./command_batctl_script.sh start 1 TEST_BATMAN_[X]_HOP.txt
11 # DAL NODO .131 in /root/COGNET_TESTBED:
12 # avviare  batctl_tr_script per registrare il traceroute verso la
       destinazione scelta:
13 ./batctl_tr_script.sh ALIX_13[X] 10 TEST_BATMAN_[X]_HOP_tr.txt
14 # in /root/MANAGER:
15 # controllare che in "fileIPAlix" siano presenti tutti gli
       indirizzi IP dei nodi della rete;
16 # modificare commandNode_Iperf.sh impostando la destinazione di
       iperf e il prefisso "BATMAN":
17 vi commandNode_Iperf.sh
18 # avviare l'esperimento:
19 ./commandNode_Iperf.sh
20
21 # Al termine dell'esperimento:
22 # sul nodo .131 fermare batctl_tr_script (per il rilascio del
       file):
23 echo "stop" > /root/COGNET_TESTBED/batctl_tr_script_state
```

```
24 # Dal PC di controllo:
25 # fermare batctl_script (per il rilascio dei file):
26 ./command_batctl_script.sh stop
27 # fermare CogNet (per il rilascio dei file):
28 ./command_COGNET.sh stop
29
30 # SU wisewai-server: salvare i risultati impostando un suffisso
     che identifichi l'esperimento:
31 ./save_res.sh TEST_BATMAN BATMAN [X]_HOP
```

3.1 Risultati e commenti

La prima serie di grafici (Figura 3.1, Figura 3.2, Figura 3.3 e Figura 3.4) mostra il *goodput* medio di ognuno dei 5 esperimenti effettuati per ogni numero di hop e per ogni modalità di *routing*. Si può notare come ad ogni aumento di hop il *goodput* venga più che dimezzato, questo è dovuto ai numerosi fattori che si aggiungono al problema dell'*half duplex*, e che si cercherà di analizzare. Si può inoltre osservare che in più di un'occasione negli esperimenti a 1 hop si hanno misure molto più basse delle altre: si è osservato che probabilmente ciò sia dovuto ad altre trasmissioni Wi-Fi in atto sul canale utilizzato, che fanno abbassare il *bitrate* nominale scelto dalla scheda di rete; questa deduzione è fatta sulla base di alcuni risultati ottenuti, come quello di Figura 3.5. Sono questi tipi di problematiche che una simulazione non può evidenziare.

Le medie sui 5 esperimenti si possono confrontare nel grafico di Figura 3.6. È necessario sottolineare il fatto che le medie delle misure effettuate a 1 e 2 hop si possono considerare veritiere, mentre per quelle a 3 e 4 hop entrano in gioco molti fattori, che si cercherà di analizzare in seguito, che renderebbero necessari esperimenti su un tempo molto più lungo per avere medie vicine alla realtà. Si può osservare, come ci si aspettava, che la modalità con *routing* statico offre il miglior *goodput*, questo perché non c'è l'*overhead* dei pacchetti di controllo dei protocolli sul canale e perché non ci sono cambi di rotta. In ordine di migliori performance in termini di *goodput*, dopo il *routing* statico si ha il protocollo BATMAN, le sue buone performance sono dovute al fatto che, nonostante trasmetta pacchetti di controllo una volta al secondo come OLSR, questi sono molto piccoli e non occupano quindi per molto tempo il canale; come sarà ben visibile nella seconda serie di grafici BATMAN è affetto da una modesta frequenza di cambi di rotta. Infine si hanno le due versioni di OLSR: la versione libera raggiunge un *goodput* leggermente superiore perché nei momenti in cui uno dei canali radio è in brutte condizioni il protocollo può riuscire a trovare un altra strada per il pacchetto (saltando un hop); questo non è concesso alla versione "bloccata", che però, come si osserverà nei grafici successivi, presenta alcuni lati positivi.

La seconda serie di grafici (Figura 3.7, Figura 3.8, Figura 3.9 e Figura 3.10) illustra il *goodput* istantaneo misurato in ACK al secondo (differisce da quello in Mbps per un semplice fattore di scala). Per quanto riguarda il *routing* statico (Figura 3.7) si può osservare come il *goodput* resti pressoché costante per gli esperimenti a 1 e 2 hop, mentre inizi ad avere forti irregolarità per i 3 e 4 hop evidenziando come il TCP sia inadatto a trasmissioni *multihop*, per i motivi che si discuteranno in seguito. Nei grafici del protocollo BATMAN (Figura 3.8) nell'esperimento a 2 hop si iniziano ad osservare i "buchi" del *goodput* causati dai cambi di rotta; negli esperimenti a 3 e 4 hop il *goodput* è molto irregolare a causa dei cambi di rotta (buchi più stretti) e dei problemi del TCP (buchi più larghi). OLSR dimostra di essere molto più soggetto ai cambi di rotta di BATMAN mostrando grafici estremamente irregolari, che diventano pressoché nulli negli esperimenti a 4 hop. Ciò dimostra anche come OLSR faccia molta più fatica a considerare una rotta stabile e quindi a immetterla nelle tabelle di *routing* del nodo, perché intervalli con *goodput* pari a zero così estesi non possono essere imputati al solo TCP, ma anche alla mancanza di una rotta su cui inoltrare i pacchetti. È molto interessante osservare come OLSR BLKD a 2 hop (Figura 3.10) presenti molti meno buchi rispetto alla versione FREE (Figura 3.9) non essendoci cambi di rotta.

L'analisi dei buchi causati dai cambi di rotta in BATMAN è presentata nei grafici di Figura 3.11 e Figura 3.12. Per questo tipo di analisi gli esperimenti più adatti sono quelli a 2 hop, nei quali non sono ancora evidenti i problemi del TCP. Nei grafici vengono tracciate le metriche di BATMAN per i link dal nodo .131 al .133 e dal nodo

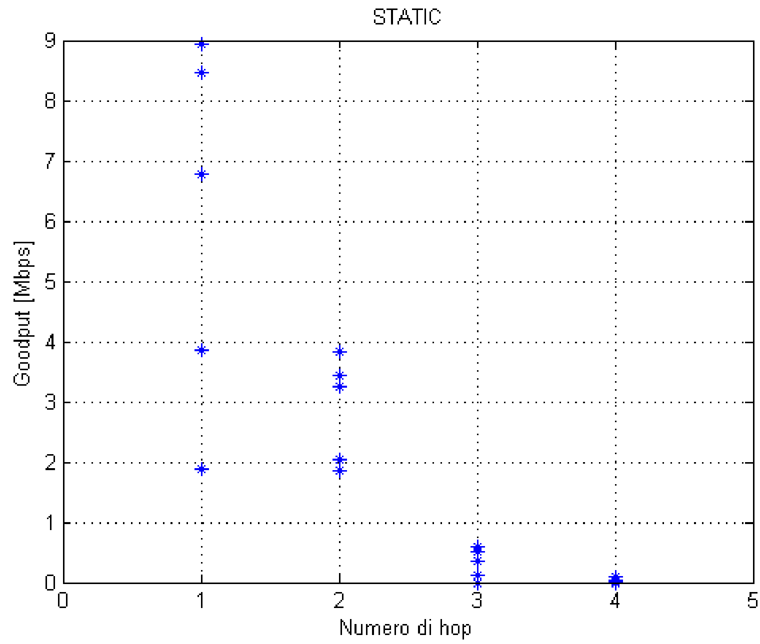


Figura 3.1: Goodput in modalità *routing* statico.

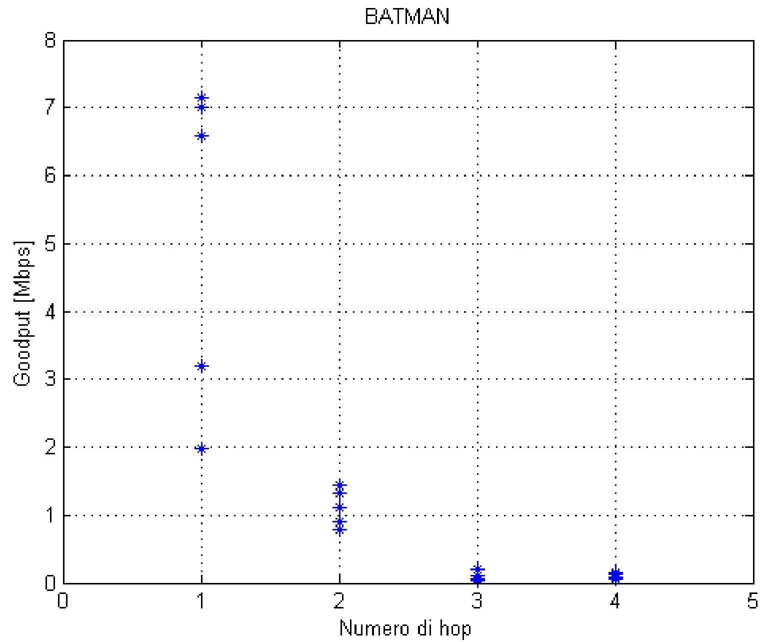


Figura 3.2: Goodput con protocollo di *routing* BATMAN.

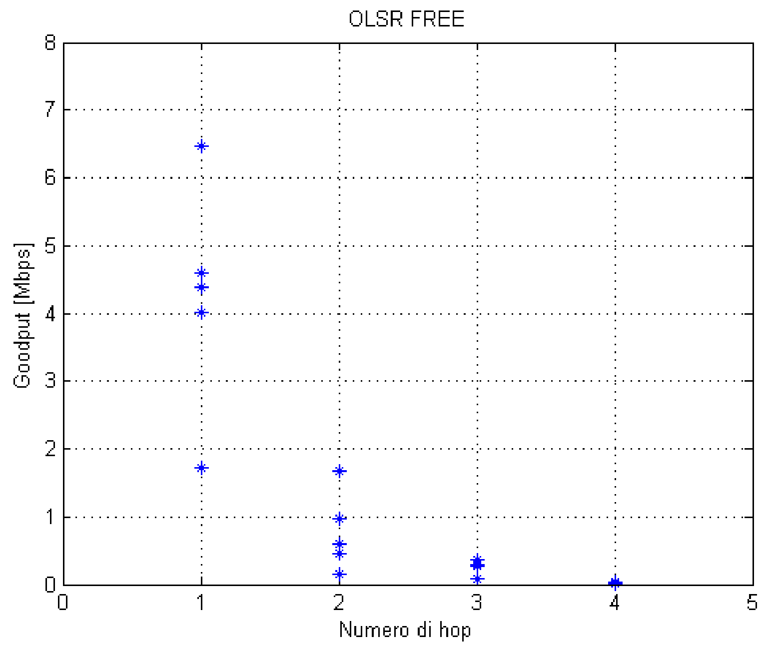


Figura 3.3: Goodput con protocollo di *routing* OLSR libero.

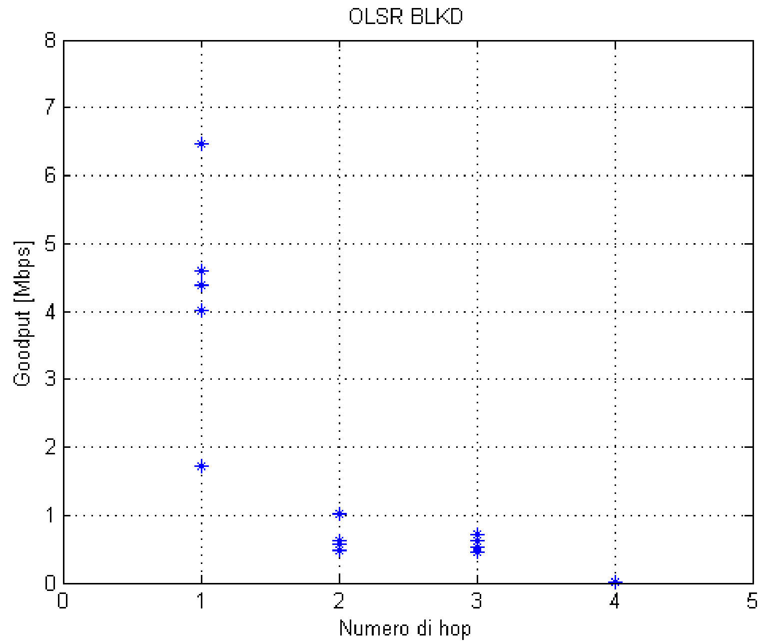


Figura 3.4: Goodput con protocollo di *routing* OLSR con rotte bloccate.

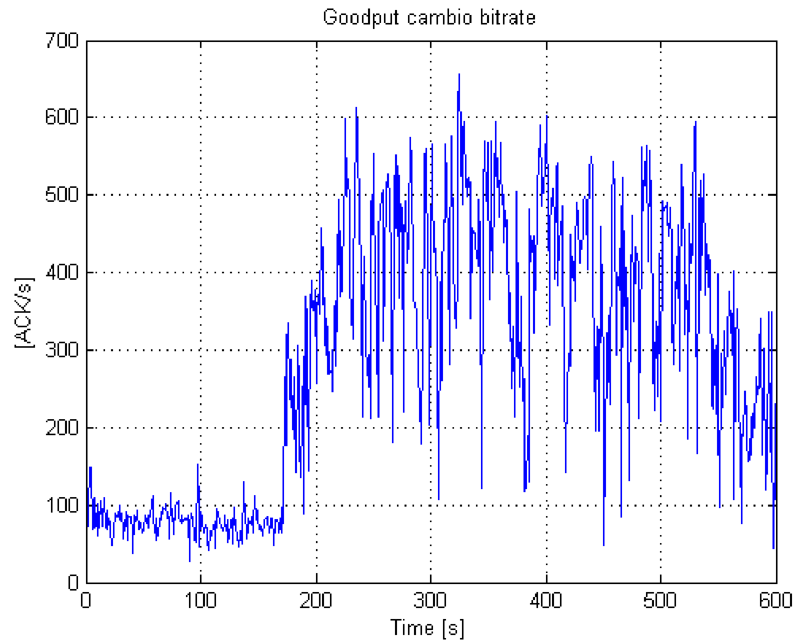


Figura 3.5: Goodput con cambio di *bitrate*.

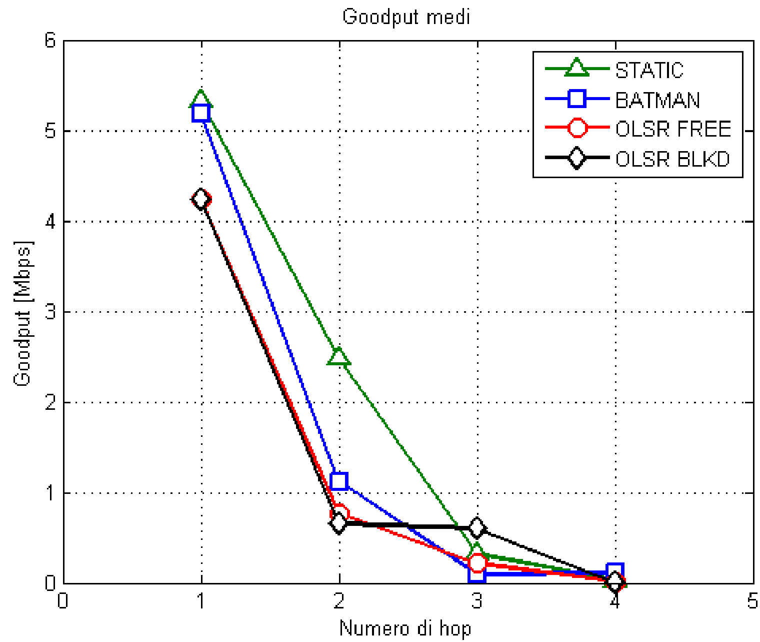


Figura 3.6: Confronto tra i goodput medi.

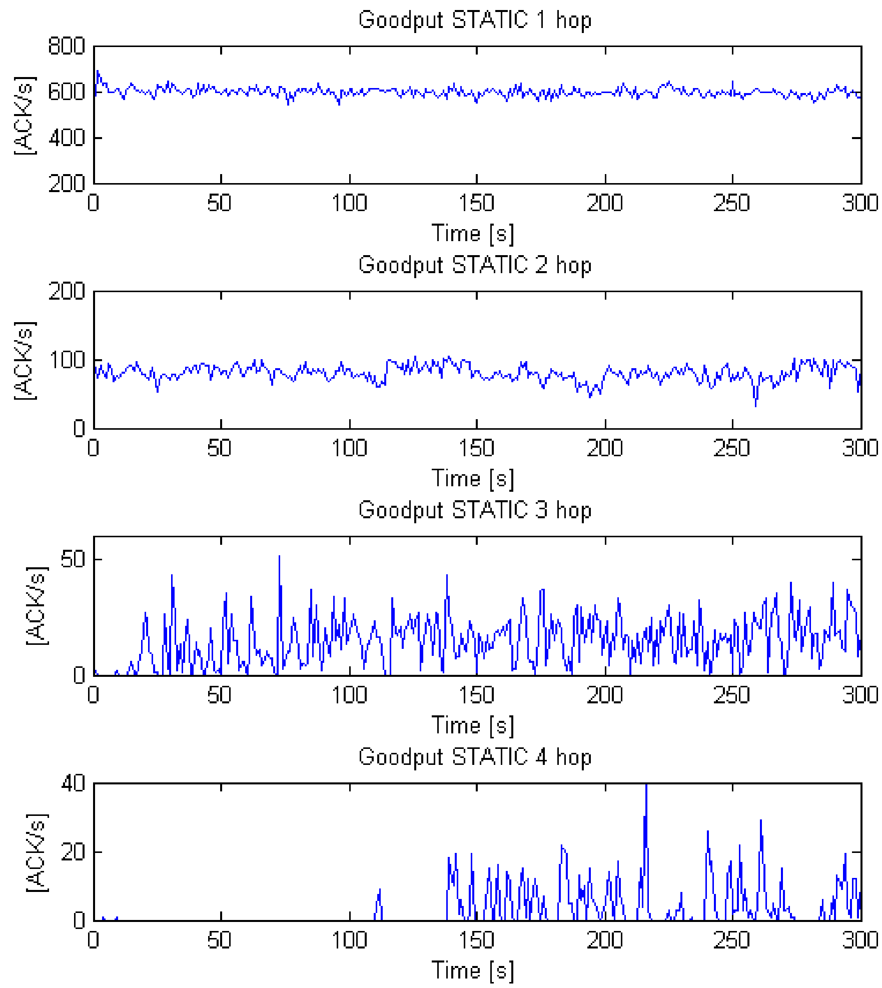


Figura 3.7: Andamento del goodput istantaneo per la modalità routing statico.

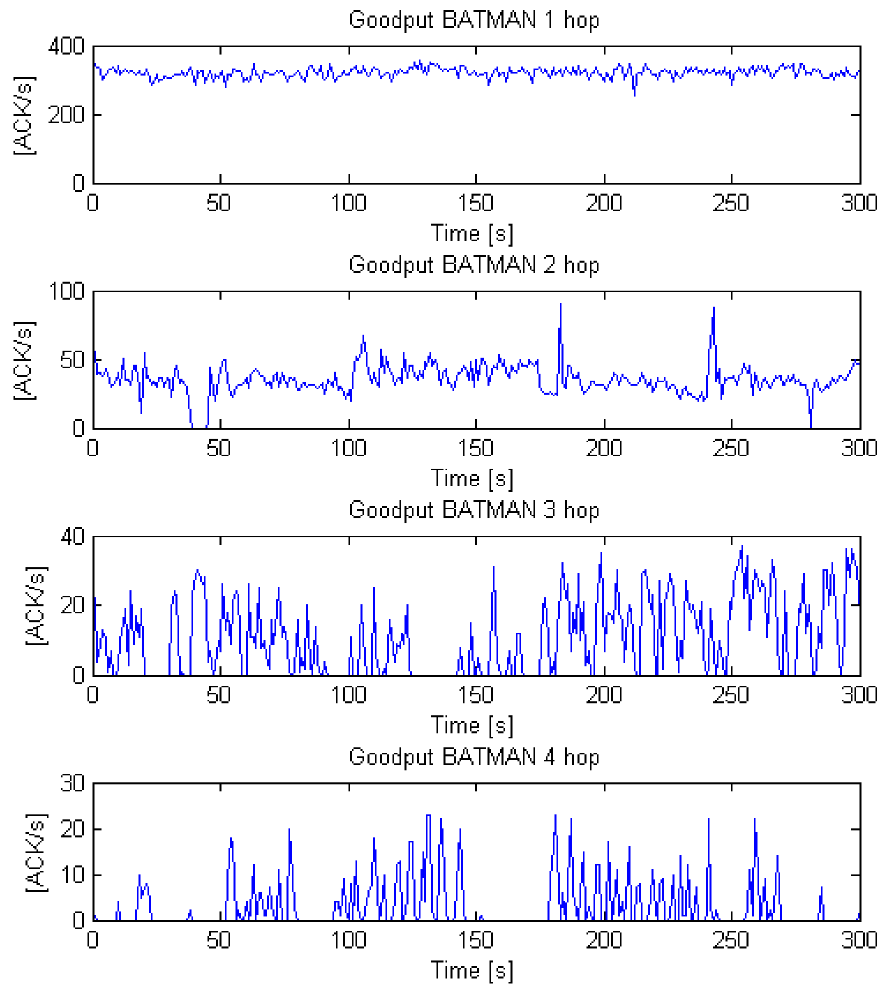


Figura 3.8: Andamento del goodput istantaneo con BATMAN.

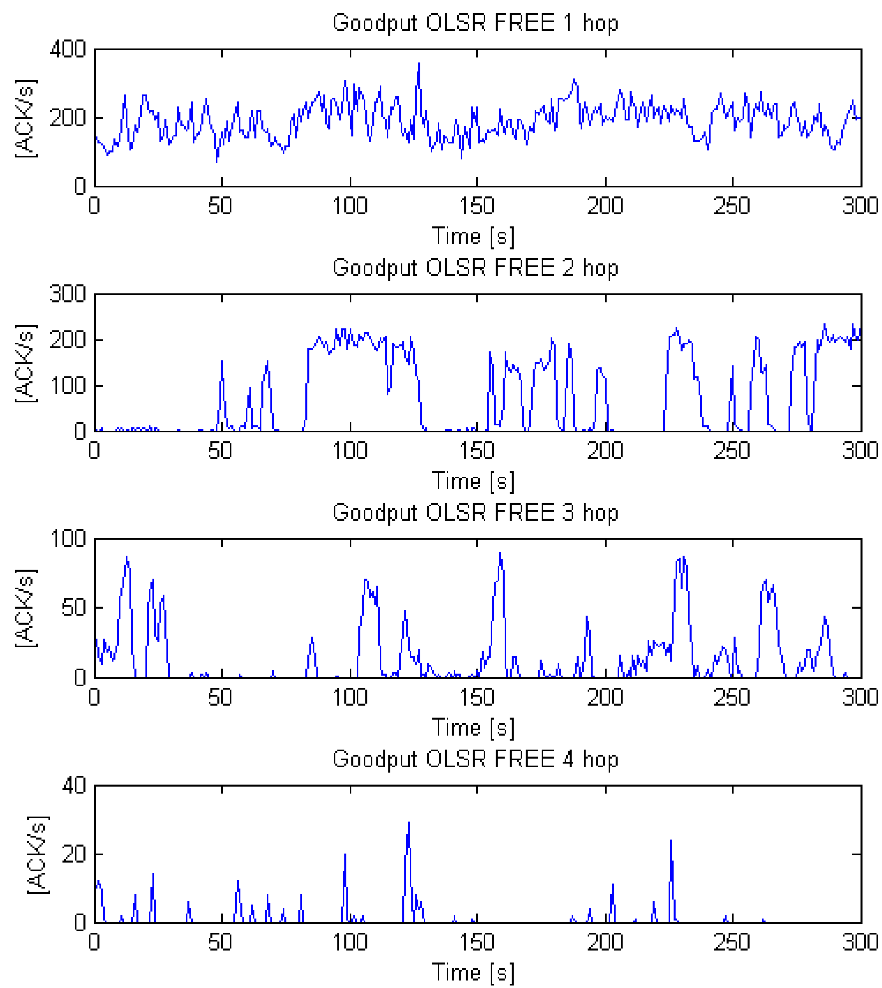


Figura 3.9: Andamento del goodput istantaneo con OLSR libero.

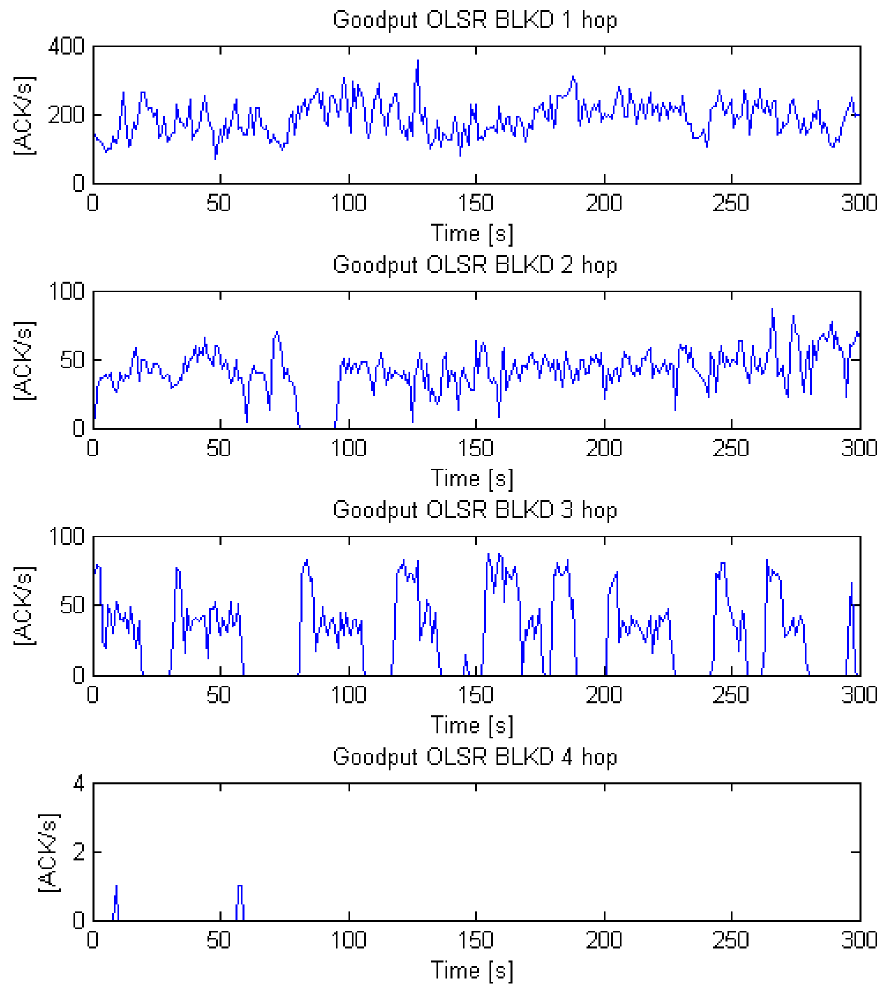


Figura 3.10: Andamento del goodput istantaneo con OLSR con rotte bloccate.

.131 al .138 (si ricorda che per BATMAN la metrica vincente è quella più alta); viene inoltre tracciato una valore logico che indica quando il *next hop* del nodo .131 sia o meno il .133 (il *nest hop* "nominale"), in caso contrario il pacchetto salterà il nodo .133 per arrivare direttamente al .138. Quest'ultima traccia aiuta molto ad identificare quando è avvenuto un cambio di rotta, ed è estremamente evidente, soprattutto nel particolare di (Figura 3.12), come questa comporti il fatidico "buco" del *goodput*. Si può anche osservare come il buco avvenga in situazioni in cui le metriche sono molto vicine tra loro e si superino a vicenda molte volte in poco tempo. In Figura 3.11 si può osservare un secondo cambio di rotta verso la fine della traccia: anche qui è presente un crollo del *goodput*; quest'ultimo non è perfettamente allineato per via di una scala temporale non precisa nella misura degli ACK ricevuti in un secondo, causata da un clock di sistema del TCP nel kernel Linux non perfetto, che necessita di continue correzioni da parte del sistema operativo. Il fenomeno è stato osservato sperimentalmente ed è un altro esempio di fenomeno non visibile in una simulazione.

Con OLSR lo studio è stato più difficile per via dell'estrema instabilità delle metriche. In Figura 3.13 e in Figura 3.14 vengono tracciate le metriche ETX del link diretto dal nodo .131 al nodo .138 e del percorso a 2 hop .133 → .133 → .138 (si ricorda che il valore di ETX vincente è il più basso). In Figura 3.13 si può notare come quasi ogni volta che le due metriche si incrociano, il *gootput* crolli a zero, mentre quando una metrica resta la più bassa per un bel periodo di tempo, il *goodput* resti alto. Tutto ciò è ancora più evidente nel particolare di Figura 3.14.

I buchi nel *goodput* causati dai cambi di rotta imposti dai protocolli di *routing* potrebbero essere spiegati in questo modo: partendo da una situazione in cui il *throughput* è stabile e i canali buoni, si avrà un certo *Round Trip Time* (RTT) pressoché costante; il cambio di rotta non intacca direttamente il TCP perché esso lavora *end-to-end* ma possono verificarsi delle latenze a livello IP, che per qualche secondo non conosce la rotta verso cui instradare i pacchetti. Ciò potrebbe comportare la scadenza dell'RTO che si trovava ad un valore basso e tutto ciò che ne consegue: la *Congestion Window* (CWND) viene riportata a 1 (quindi il *goodput* ne risente) e la *Slow Start Threshold* dimezzata.

Come già accennato il TCP presenta grossi problemi nelle trasmissioni radio *multihop* [9] : il meccanismo di ritrasmissione assume che i pacchetti vengano persi a causa di una congestione, di conseguenza raddoppia il *Retransmission Time Out* (RTO) per far defluire il traffico. Questo comportamento in una rete radio è estremamente controproducente in quanto la maggioranza delle volte che un pacchetto viene perso, la causa è un *link failure* o un canale sfavorevole. Questo evento porta ad un aumento esagerato dell'RTO, il traffico non riprende a defluire in quanto se un link è rotto resta tale per un tempo non indifferente e quindi si arriverà alla scadenza dell'RTO e a tutto ciò che questo comporta: la CWND viene riportata a 1 e la *Slow Start Threshold* dimezzata; la conseguenza più grave è però il lungo periodo di tempo di inattività fino alla scadenza dell'RTO (i buchi più grossi visibili nei grafici); paradossalmente sarebbe più conveniente che l'RTO scadesse subito. Questi fenomeni sono ben visibili nei grafici di Figura 3.15: si può notare che ogni periodo in cui il *goodput* resta alto può essere associato ad una salita dalla CWND; questi periodi sono però divisi da dei buchi che durano esattamente quanto l'RTO che era salito molto supponendo una congestione. Si presti attenzione che la scala temporale del *goodput* (il primo grafico) è il tempo reale in secondi, mentre quella degli altri 3 grafici è dettata dall'arrivo degli ACK (momento in cui tutti i parametri di TCP vengono aggiornati), ed è per questo che non mostra i momenti in cui il *goodput* è zero.

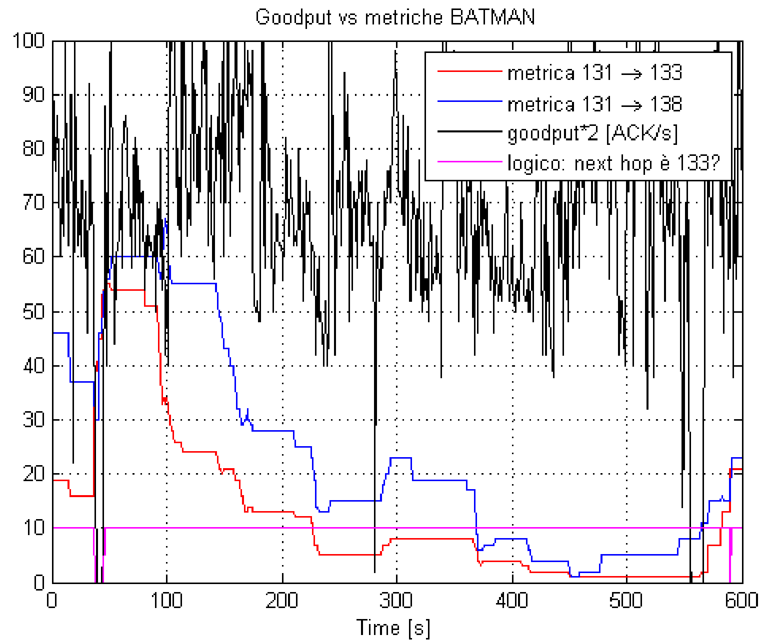


Figura 3.11: Analisi di come il cambio di rotta azzera il goodput con BATMAN.

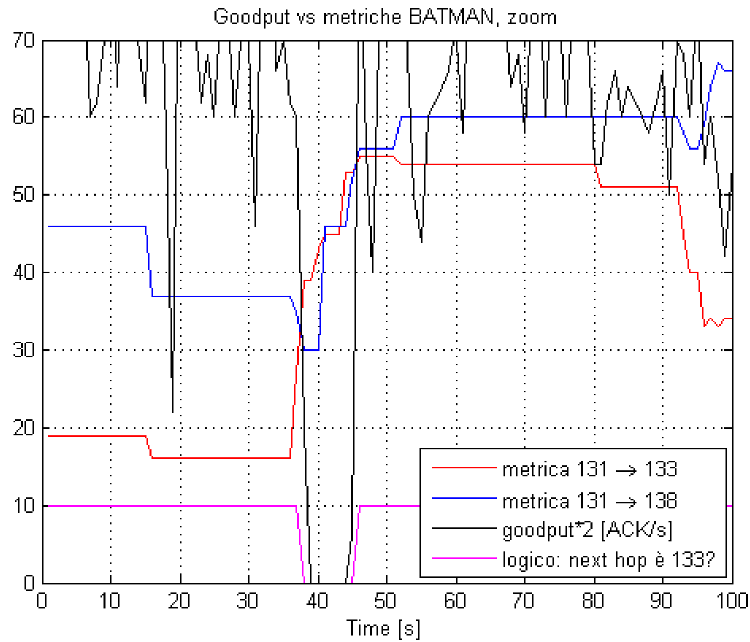


Figura 3.12: Particolare di come il cambio di rotta azzera il goodput con BATMAN.

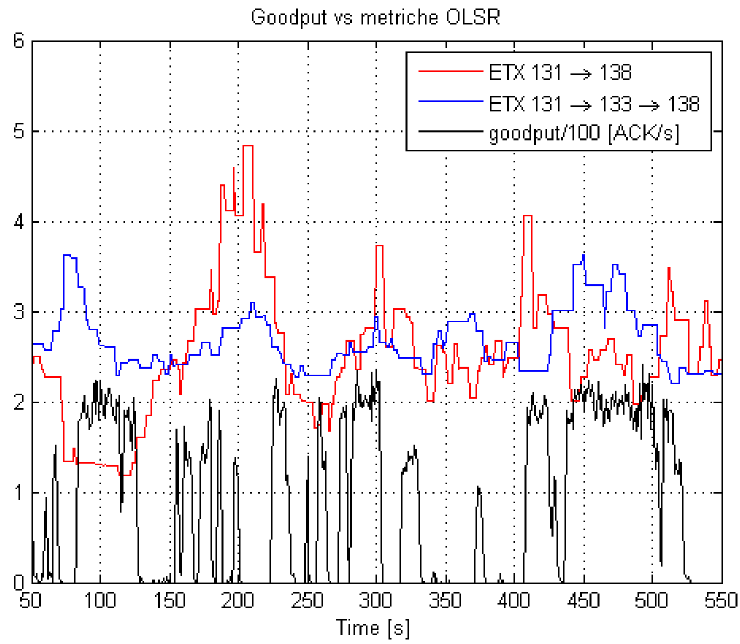


Figura 3.13: Analisi di come il cambio di rotta azzera il goodput con OLSR.

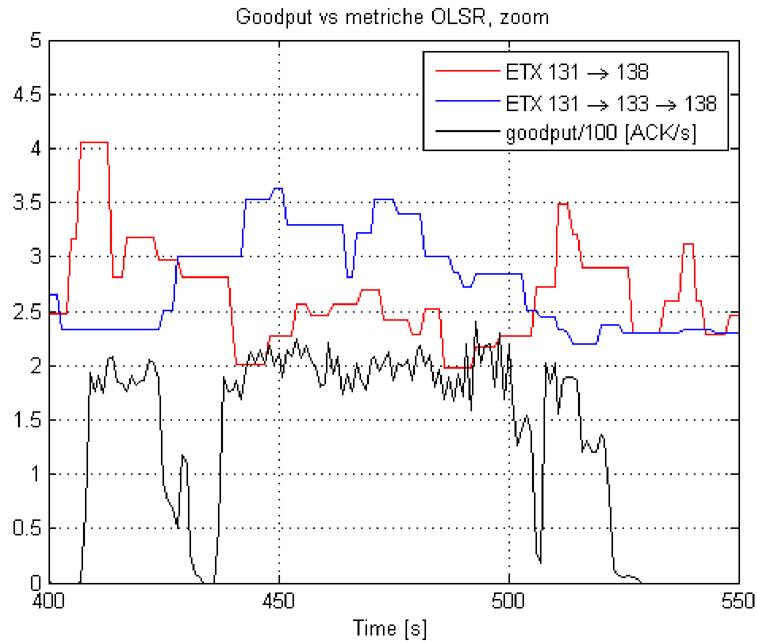


Figura 3.14: Particolare di come il cambio di rotta azzera il goodput con OLSR.

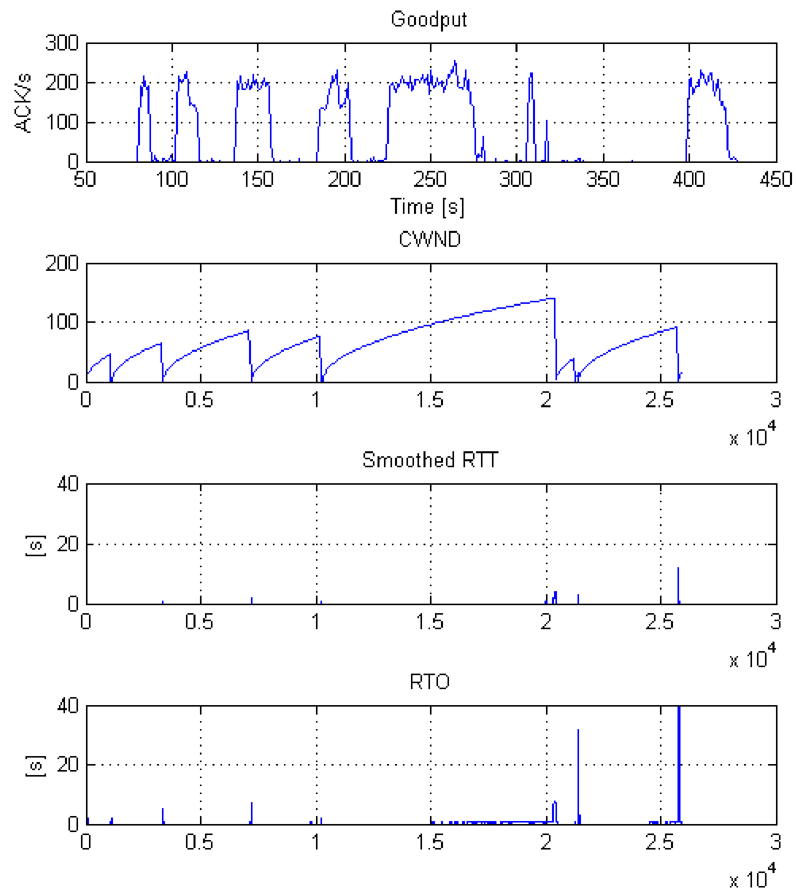


Figura 3.15: Alcuni parametri del TCP per OLSR FREE 2 hop.

Capitolo 4

Conclusioni

In questa tesi sono state spiegate tutte le operazioni da effettuare per la creazione e la corretta messa in opera di un *testbed* per una rete Wi-Fi *mesh* cognitiva. Si sono utilizzati dispositivi commerciali a basso costo in maniera tale che ogni gruppo di ricerca possa essere in grado di riprodurlo. Si è utilizzato solo *software opensource*.

Si sono analizzate le *performance* in una situazione reale dei due protocolli di *routing* analizzati e di una modalità che fa uso di *routing* statico. È risultato evidente che in una topologia del tipo del *testbed* (nodi disposti in lunghezza), l'utilizzo di un qualsiasi protocollo di *routing* appesantisce il traffico e soprattutto crea dei fenomeni in cui dei cambi di rotta non necessari azzerano per qualche secondo il *throughput*. L'uso di questi protocolli resta però indispensabile nei casi in cui uno qualsiasi dei link cada, in questo caso i protocolli permettono una importante funzione di auto-riparazione della rete. L'utilizzo dei protocolli di *routing* resta indispensabile nel caso in cui si introducano nodi mobili, sia che siano nodi *mesh* (che fanno parte della *backbone*), sia che siano nodi utente. La soluzione migliore sarebbe un algoritmo di *cognitive network* che inibisca gli spontanei cambi di rotta inutili per i percorsi che non ne hanno bisogno, ma che li riattivino solo nel momento in cui un link si rompa. L'algoritmo potrebbe analizzare la situazione e agire sui parametri dei protocolli. Si potrebbe pensare altrimenti ad un più semplice meccanismo *cross-layer* che notifichi al TCP i cambi di rotta imposti dai protocolli, in maniera tale che il tempo necessario per ristabilire la rotta non degradi la connessione. Si è osservato che nella situazione analizzata il protocollo BATMAN ha *performance* migliori rispetto a OLSR per via dei suoi pacchetti più piccoli che occupano meno il canale e soprattutto per la sua metrica che risulta essere meno soggetta ai frequenti cambi di rotta inutili.

Si è verificata inoltre nella pratica la ben nota inadeguatezza del TCP su trasmissioni *multihop*.

4.1 Lavori futuri

Molti sono i lavori che si potrebbero effettuare sul *testbed*, sia per quanto riguarda l'espansione e l'evoluzione dello stesso, sia per quanto riguarda gli esperimenti effettuabili su di esso.

I lavori da fare nell'immediato possono essere l'introduzione dei nodi mobili (sia mesh che utente) nelle misure di *performance* dei protocolli, e l'introduzione di altri protocolli di *routing* per reti *mesh*/*ad-hoc*, ad esempio il protocollo BABEL.

Successivamente si potrebbe pensare all'implementazione e sperimentazione di qualche algoritmo di *cognitive network*. In letteratura non si trova ancora nulla di questo tipo. Un esempio potrebbe essere quello proposto, che evita i cambi di rotta in situazioni dove questi peggiorano solo le cose.

Si potrebbe pensare anche ad una reimplementazione del TCP che limiti i suoi problemi nelle trasmissioni *multihop* e ad una prova pratica della stessa sul *testbed*.

Bibliografia

[1] T. Clausen, P. Jacqueti, *RFC 3626 - Optimized Link State Routing Protocol (OLSR)*, October 2003, http://www.ietf.org/rfc/rfc3626.txt.pdf.

[2] Sito ufficiale di OLSR, http://www.olsr.org/.

[3] A. Neumann, C. Aichele, M. Lindner, S. Wunderlich, *Better Approach To Mobile Ad-hoc Networking (BATMAN)*, April 2008, http://tools.ietf.org/pdf/draft-wunderlich-openmesh-manet-routing-00.pdf.

[4] Sito ufficiale di BATMAN, http://www.open-mesh.org/.

[5] M. Danieletto, G. Quer, R. R. Rao, M. Zorzi, *A cognitive Networking Testbed on Android OS Devices*.

[6] M. Danieletto, G. Quer, R. R. Rao, M. Zorzi, *On the Exploitation of the Android OS for the Design of a Wireless Mesh Network Testbed* http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6735760.

[7] Sito Mesh Dynamics, http://www.meshdynamics.com/performance-analysis.html.

[8] R. W. Thomas, L. A. DaSilva, Allen B. MacKenzie, *Cognitive Networks* http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1542652.

[9] M. Mezzavilla, G. Quer, M. Zorzi, *On the Effects of Cognitive Mobility Prediction in Wireless Multi-hop Ad Hoc Networks*.