Ringraziamenti

Un ringraziamento particolare va alla mia famiglia che mi ha sostenuto in questi anni permettendomi di raggiungere questo obiettivo.
Un sentito ringraziamento va a Dario che mi ha seguito in questo lavoro di tesi con pazienza e professionalità.
Ringrazio inoltre il prof. Antonio Corradi per la grande disponibilità e attenzione dimostrata nei miei confronti.
Ringrazio anche Francesca che mi è stata vicina in questi ultimi anni e che mi ha sopportato così a lungo, e tutti i miei amici insieme ai quali ho trascorso questi meravigliosi anni di università.

Indice

Introduzione

La crescente proliferazione di dispositivi portatili che possono fruire di connettività wireless, i recenti sviluppi delle tecnologie wireless e l'emergenza delle Mobile Ad-Hoc NETwork (MANET), aprono un nuovo scenario. Nel nuovo scenario degli utenti richiedono la possibilità di beneficiare di servizi innovativi che consentano di collaborare ovunque essi si trovino, in qualunque momento e quando sono in movimento. Esempi di servizi abilitati dalle tecnologie MANET sono costruiti da applicazioni per il file sharing fra utenti mobili, dal coordinamento di veicoli, da scenari di protezione civile.
Le caratteristiche delle MANET sollevano molti problemi nello sviluppo di servizi collaborativi. La topologia della rete nn è determinabile a priori rendendo impossibili assunzioni sulla corrente disponibilità on-line delle diverse entità interagenti. Disconnessioni, partizioni e merge di rete sono eventi comuni che causano transitori nella collaborazione fra partner nuovi e precedentemente conosciuti.
Un aspetto molto importante nello sviluppo di servizi avanzati in scenari MANET è costituito dall'insieme di quei protocolli che consentono la disseminazione delle informazioni all'interno della rete.
Protocolli di comunicazione di questo tipo sono adottati per esempio, per il discovery di risorse o servizi all'interno di una rete, o per la comunicazione multipoint.
Recentemente la ricerca ha individuato diverse soluzioni adatte alla disseminazione di informazioni in scenari ad-hoc. In particolare, diversi studi a riguardo hanno introdotto protocolli di Gossip che promuovevano approcci probabilistici per la diffusione delle informazioni all'interno di una rete. In questo lavoro di tesi approfondiremo lo studio sui protocolli appartenenti a questa famiglia, analizzando in particolare le prestazioni fornite da un protocollo Gossip(p,0) in uno scenario di rete Ad-hoc e mettendole a confronto con quelle fornite dai generici protocolli di flooding.
La tesi è organizzata come segue: nel Capitolo 1 vengono introdotte le reti wireless ed in particolare vengono analizzate varie tipologie di rete come BAN, PAN, LAN e WAN e i loro possibili scenari applicativi. Inoltre vengono illustrati due principali standard per le reti wireless e cioè IEEE 802.11 e Bluetooth, e mostrati i vari modelli architetturali di reti. Nel Capitolo 2 viene analizzato uno di questi modelli, in particolare il modello di rete Ad Hoc e descritti i principali protocolli di routing unicast per questa particolare tipologia di rete.

Nel Capitolo 3 vengono analizzate le problematiche relative al routing di tipo multicast nelle reti Ad Hoc ed effettuata una panoramica sui principali protocolli.Nel Capitolo 4 focalizziamo la nostra attenzione su una tipologia di protocolli di comunicazione in particolare, cioè il Gossip. Ne verrà spiegato il principio di funzionamento e saranno illustrati i principali protocolli di Gossip. Il Capitolo 5 mostra l'architettura del simulatore di reti Ns2 ed un breve tutorial per spiegarne il funzionamento. Il Capitolo 6 mostra infine i risultati delle simulazioni effettuate.
Seguono le conclusioni.

Cap. 1
Reti Wireless

1.1 Introduzione alle reti Wireless

La crescente diffusione dei personal computer e la conseguente evoluzione tecnologica, al fine di ricercare sempre una maggiore flessibilità, associata alle esigenze di mobilità, ha portato ad una crescente diffusione degli standard per reti di tipo wireless. Il più grande vantaggio di queste tipologie di reti è senz'altro la mobilità, consentendo agli utenti connessi con i loro dispositivi di spostarsi continuamente rimanendo comunque collegati alla rete, sfruttando anche le reti cablate preesistenti.

Un altro fattore a favore delle reti wireless è senz'altro la loro flessibilità, sia in termini di scalabilità sia in termini di usabilità. Le reti Wireless utilizzano un certo numero di stazioni base al fine di connettere queste reti a quelle preesistenti. Questa infrastruttura comunque resta qualitativamente la stessa anche nel caso si dovessero connettere un grandissimo numero di utenti. Siccome infatti per poter fornire un servizio di questo tipo sono necessarie un antenna e una stazione base, una volta costituita l'infrastruttura aggiungere un potenziale utente non desta particolari problemi. Inoltre questa nuova tecnologia offre il grandissimo vantaggio di interconnettere dispositivi eterogenei situati in reti eterogenee. Infatti oltre a consentire la comunicazione fra le diverse reti wireless, caratterizzate dalla variazione di diversi parametri come banda e raggio di trasmissione, consentono la comunicazione anche con reti fisse, rendendo lo sviluppo di questa tecnologia ancora più importante data la grande varietà di tipologie di utilizzo, come per esempio a scopi ludici, personali e militari.

[Moh03] Possiamo dare una classificazione di queste tipologie di rete in base alla loro copertura in termini di maggior distanza di trasmissione, in quattro classi principali come mostrato in Figura 1.0 :

- Body Area Network (BAN)
- Personal Area Network (PAN)
- Local Area Network (LAN)
- Wide Area Network (WAN)

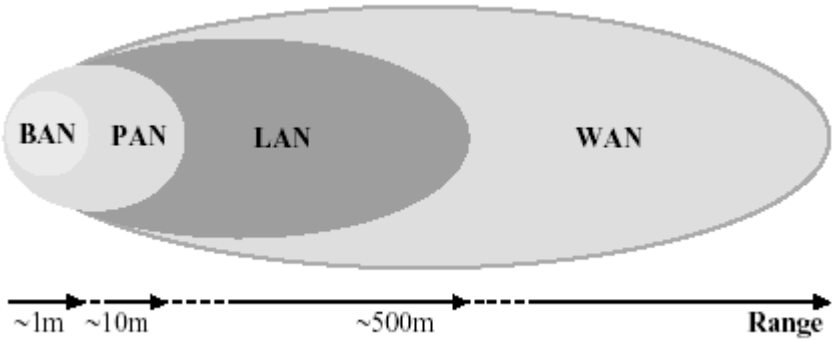

Figura 1.0 Tassonomia delle Reti Ad-Hoc

1.1.1 Body Area Network

Una rete BAN (Body Area Network) è fortemente correlata con i dispositivi elettronici indossabili. Essa nasce per l'esigenza di interconnettere dispositivi eterogenei all'interno del raggio di comunicazione della stessa, che approssimativamente corrisponde all'altezza di una persona, cioè circa 1-2 metri, come per esempio microfoni, cuffie, cellulari, lettori Mp3.
Le reti BAN devono garantire la possibilità di consentire il networking fra più reti ed inoltre devono far si che all'utente sia completamente trasparente il fatto di aggiungere o rimuovere un dispositivo BAN da una rete; si parla in tali casi di capacità di autoconfigurazione. Uno dei primi prototipi di rete BAN è stato quello sviluppato da T.G. Zimmerman, che propone un modello che genera un flusso di dati che viaggiano attraverso tutto il corpo sfruttandolo come un normale canale di comunicazione con una banda dell'ordine dei 400 Kbit/s.

1.1.2 Personal Area Network

Una rete PAN (Personal Area Network) risponde all'esigenza di consentire il networking di dispositivi mobili in possesso degli utenti quali PDA, laptop, cellulari. Tipicamente queste reti possiedono un raggio di trasmissione dell'ordine della decina di 10 metri ed una banda di trasmissione dell'ordine dei Mbps. Una rete PAN particolarmente rilevante è Bluetooth. La rete Bluetooth è emersa come uno standard de facto e fornisce connettività per un numero di dispositivi con raggio di circa 10 m e banda dell'ordine dei Mbps

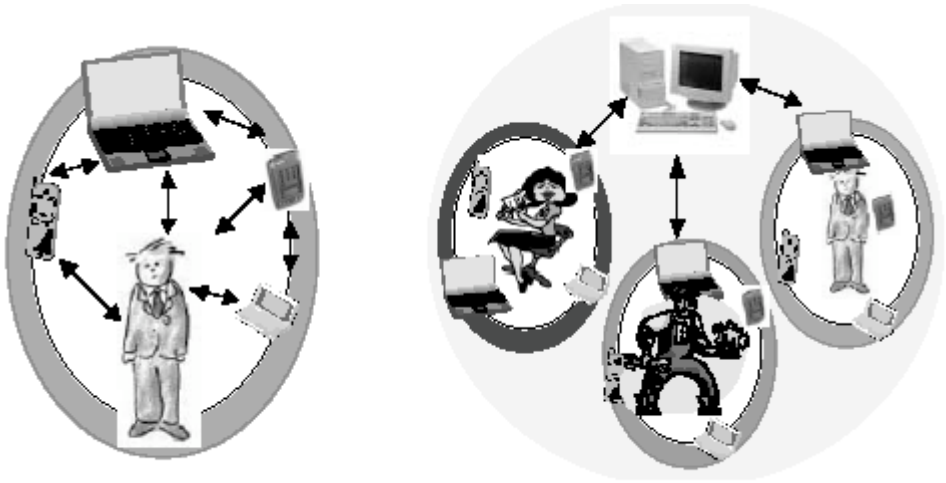

Figura 1.1 Relazione tra una BAN (fig. a) e una PAN (fig. b)

1.1.3 Wireless Local Area Network

Una rete WLAN risponde all'esigenza di fornire un supporto analogo a quello delle LAN in ambienti in cui la mobilità dei terminali rende problematica l'adozione di soluzioni cablate o di scenari in cui la rete deve essere usata per un tempo limitato (es. un concerto). Per fare questo una WLAN dovrebbe essere progettata al fine di rispettare alcuni ulteriori parametri e caratteristiche specifiche dell' ambiente wireless come la sicurezza, consumo di energia, mobilità e occupazione limitata di banda.
Negli ultimi anni l'uso della tecnologia wireless nell'ambiente LAN è divenuto sempre più diffuso. Una WLAN (Wireless LAN) ha un raggio di comunicazione compreso tra i 100 e i 500 metri ed una banda di trasmissione dell'ordine delle decine di Mbps . Uno standard rilevante per le reti WLAN è l' IEEE 802.11, il quale opera nella banda dei 2,4 Ghz con una bit rate dell'ordine della decina di Mbps.

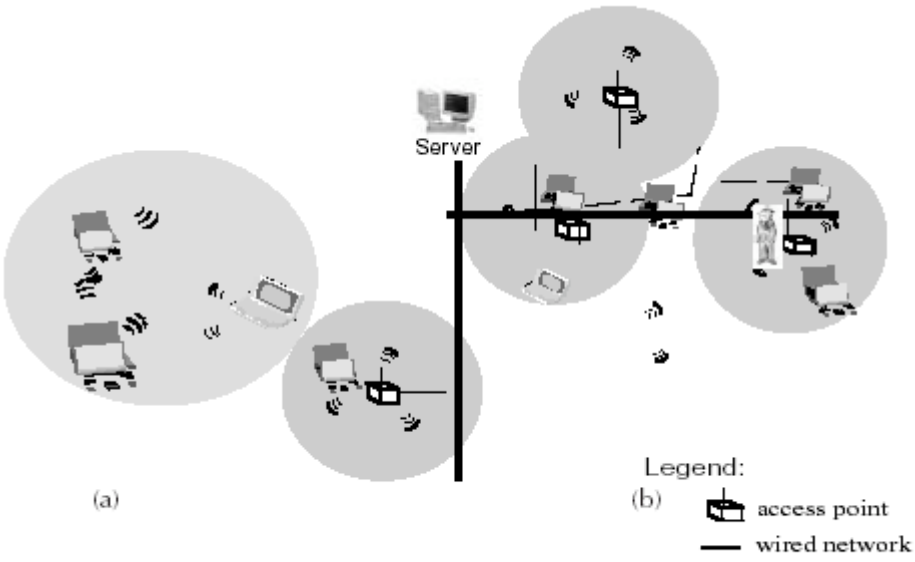

Figura 1.3 Configurazioni di una WLAN : a) rete Ad-Hoc
b) Access Point

1.1.4 Wireless Wide Area Network

Le WWAN sono reti wireless ad estensione geografica. Esse consentono infatti l'interconnessione di dispositivi situati anche a grande distanza, anche se la velocità di trasmissione è decisamente ridotta e varia tra i 5Kbps e i 28 Kbps.

1.2 Tecnologie per le reti Wireless

Affinché una tecnologia di rete abbia successo è necessaria la presenza di standard che ne permettano un corretto uso e diffusione. Due tecnologie wireless particolarmente diffuse sono :

- Lo standard IEEE 802.11 per le reti WAN
- Bluetooth per le reti BAN e PAN

1.2.1 Architettura e Protocolli del IEEE 802.11

Lo standard IEEE 802.11 fornisce uno strato MAC ed uno strato fisico per le reti WLAN (Figura 1.4). Lo strato MAC fornisce ai suoi utenti due tipi di accessi allo strato fisico :

- Un controllo dell'accesso basato sul contenuto
- Un controllo dell'accesso libero

Il metodo di accesso base del protocollo MAC del IEEE 802.11 è il DCF (Distributed Coordination Function), il quale è un protocollo di tipo CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance).

Oltre al DCF l'IEEE 802.11 incorpora al suo interno anche un metodo alternativo di accesso al mezzo, cioè il PCF (Point Coordination Function). Quest'ultimo agisce come un classico sistema di polling, dove un coordinatore fornisce i diritti di trasmissione ad una stazione alla volta.

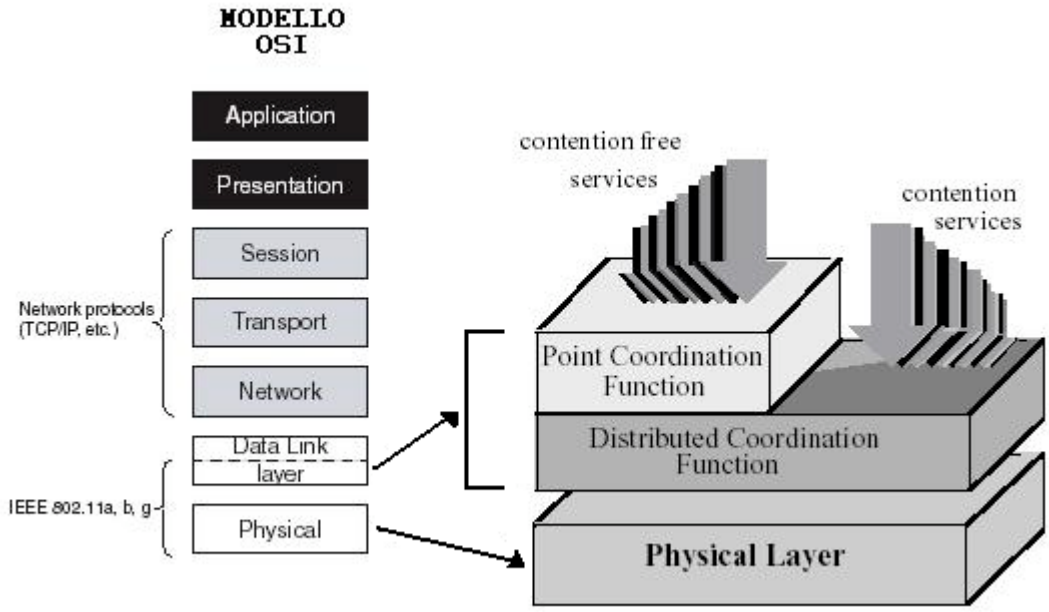

Figura 1.4 Architettura del IEEE 802.11

1.2.1.1 Il metodo di accesso DCF

L'accesso DCF consente ad una stazione che deve trasmettere un determinato messaggio di ottenere l'accesso al canale evitando cosi delle dannose collisioni, infatti la stazione controlla il canale per verificare se un'altra stazione sta trasmettendo, e se il mezzo viene trovato in Idle per un tempo maggiore di una soglia detta Distributed InterFrame Space (DIFS), questa continua con la trasmissione dei suoi dati (Figura 1.5).
In particolare quello che viene spedito inizialmente è la lunghezza presunta della trasmissione, per cui ogni stazione memorizza questa informazione in una variabile locale NAV (Network Allocation Vector) che conterrà quindi il periodo di tempo per cui il canale rimarrà occupato.(Figura 1.5 a).Questo è molto utile poiché evita alla stazione di stare sempre in ascolto sul canale e quindi potrebbe risultare importante in termini di risparmio energetico.

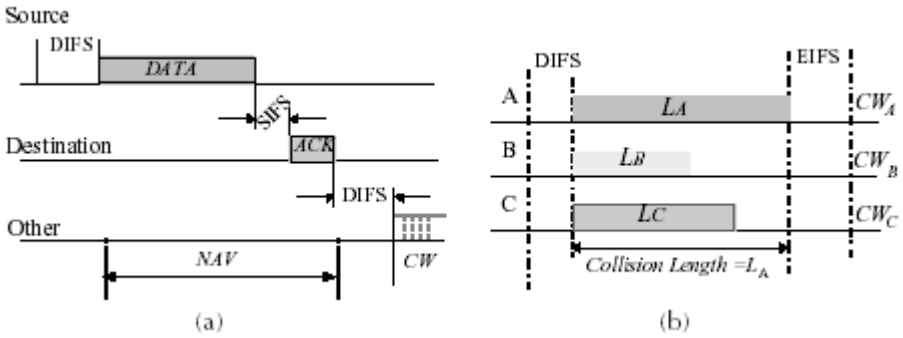

Figura 1.5 (a) Trasmissione a buon fine (b) Collisione

Poiché il protocollo CSMA/CA non supporta la capacità delle stazioni di accorgersi delle collisioni dovute alle proprie trasmissioni, il ricevente di un certo pacchetto attende al massimo un intervallo di tempo chiamato Short InterFrame Space (SIFS). Durante questo intervallo se il pacchetto viene ricevuto, inizia immediatamente la trasmissione di un frame **ACK** . Se l'ACK non viene ricevuto si presume che il frame sia stato perso e si provvede quindi a rischedulare la trasmissione del pacchetto. Al fine di scoprire eventuali errori di trasmissione viene adottato un algoritmo **CRC** che va ad analizzare il pacchetto appena ricevuto.
Se la stazione ricevente non ritrasmette l'ACK poiché c'è stato un errore di trasmissione oppure una collisione, il canale deve rimanere in uno stato di **Idle** per almeno un intervallo di tempo pari al **EIFS** (Extended InterFrame Space) prima che la stazione che ha trasmesso il pacchetto riattivi l'algoritmo di backoff per rischedulare la trasmissione. Inoltre quando una stazione con un pacchetto pronto per la trasmissione trova il canale occupato ritarda lo start fino alla fine della trasmissione corrente e a questo punto inizializza un timer per rischedulare il suo tentativo.

1.2.1.2 Il metodo RTS/CTS

La progettazione di reti WLAN basata su protocolli come il CSMA/CA è complicata dalla presenza dei terminali nascosti (Figura 1.6). Questo rende la rilevazione della portante un approccio inefficace al rilevamento delle collisioni, poiché una stazione ascoltando il canale può sentirlo libero, mentre un'altra stazione sta trasmettendo, si verifica inevitabilmente una collisione che nessun rilevamento di portante può risolvere. Il fenomeno delle stazioni nascoste si può verificare sia nelle reti basate sulle infrastrutture sia nelle reti Ad-Hoc, anche se in queste ultime si presenta assai più di frequente poiché non c'è alcun tipo di coordinazione tra le stazioni partecipanti alla rete.

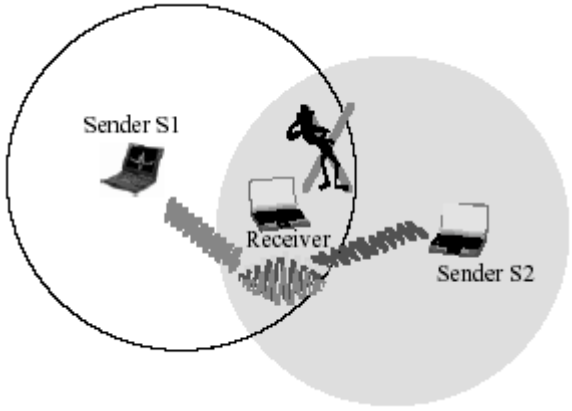

Figura 1.6 Il problema delle stazioni nascoste

Per ovviare a questo problema il meccanismo base di accesso al mezzo del IEEE 802.11 è stato esteso con un metodo di rilevazione virtuale della portante chiamato Request To Send / Clear To Send (RTS/CTS). Ora dopo che è stato guadagnato l'accesso al canale e prima di iniziare la trasmissione di un pacchetto dati viene spedito un piccolo frame di controllo chiamato RTS a tutte le future stazioni che riceveranno i dati preannunciando l'imminente trasmissione. A questo punto la stazione ricevente replica con un piccolo frame di controllo CTS per indicare la disponibilità a ricevere il pacchetto. Nel contenuto di questi frame è inserita anche la presunta lunghezza della trasmissione. Perciò, tutte le stazioni nel raggio di almeno una delle due tra trasmettitore e ricevitore, sa per quanto tempo il canale rimarrà in uso per quella trasmissione (Figura 1.7).

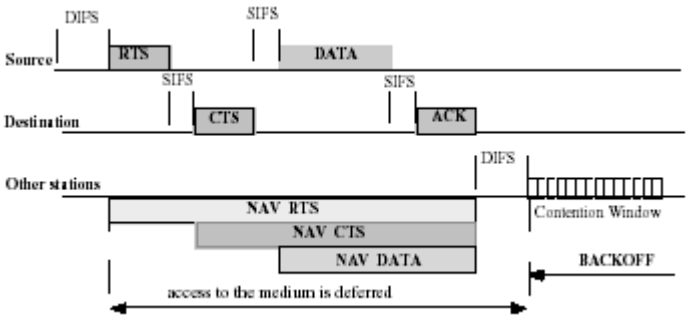

Figura 1.7 Il funzionamento del meccanismo RTS/CTS

1.2.2 La Tecnologia Bluetooth

La tecnologia Bluetooth è uno standard de facto per quanto riguarda il basso costo e il corto raggio dei collegamenti radio tra dispositivi

portatili, laptops e cellulari. Essa opera nella banda dei 2,4 Ghz e una sua unità, integrata in un microchip permette la trasmissione in reti wireless Ad-Hoc di voce e dati in ambienti stazionari e mobili. Per questo motivo si può immaginare come ben presto ogni dispositivo elettronico sarà dotato di un chip che permetta l'utilizzo di queste funzionalità.

1.2.2.1 Architettura di una rete Bluetooth

In una rete Bluetooth, una stazione ha il ruolo di master e tutte le altre stazioni sono slave, e proprio il master decide quale degli slave accederà al canale. L'unità fondamentale di una rete Bluetooth è definita Piconet, e ne fanno parte quelle stazioni che condividono lo stesso canale, cioè sono sincronizzate dallo stesso master. Una Piconet contiene un master e fino a sette slave contemporaneamente e ha una bit rate di 1Mb/sec.
Nella figura 1.8 possiamo capire meglio questi concetti costruendo un esempio in cui abbiamo due Piconet parzialmente sovrapposte dove indichiamo con "S" le stazioni slave ,con "M" le stazioni master e con "P" le stazioni che sono sincronizzate con un master ma che non partecipano ad alcun scambio di dati. Piconet indipendenti che si sovrappongono parzialmente possono formare una Scatternet, la quale si viene a creare quando una o più stazioni fanno parte contemporaneamente di due o più Piconet .
Il protocollo Bluetooth contiene a sua volta numerosi protocolli, di cui sfrutta le funzionalità come il Bluetooth radio, Baseband, LMP (Link Manager Protocol), L2CAP (Logical Link Control and Adaptation Protocol) e SDP (Service Discovery Protocol) .
Il protocollo Bluetooth radio fornisce il collegamento fisico fra i dispositivi Bluetooth, mentre lo strato Baseband fornisce un servizio di trasporto attraverso i link fisici. I servizi L2CAP sono invece usati solamente per la trasmissione dei dati.

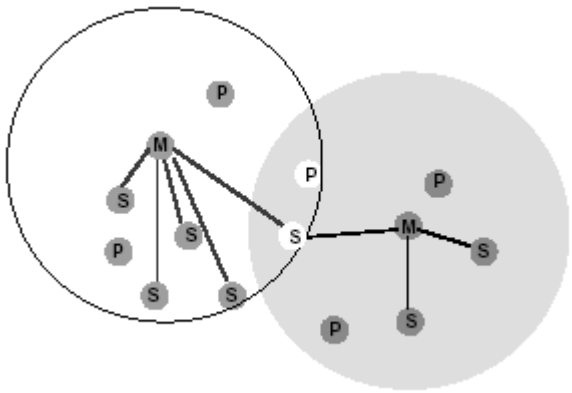

Figura 1.8 Due Piconet parzialmente sovrapposte

Come già detto il protocollo Bluetooth opera nella banda dei 2,4 Ghz dove sono definiti ben 79 canali di frequenza radio spaziati l'uno dall'altro di 1Mhz.Inoltre lo strato radio utilizza una tecnica di trasmissione detta **FHSS** (Frequency Hopping Spread Spectrum).La sequenza di salto è una sequenza pseudocasuale di lunghezza pari a 79 hop ed è unica per ogni Piconet.

1.2.2.2 Trasmissione Dati Bluetooth

Tra dispositivi Bluetooth possono essere stabiliti due tipi di collegamenti:

- Syncronous Connection Oriented (SCO)
- Asyncronous Connection Less (ACL)

Il primo tipo di collegamento è di tipo punto-punto tra un master e uno specifico slave ed è usato per la trasmissione di dati sensibili al ritardo e cioè principalmente audio. La bit-rate di un link di tipo SCO è 64 Kb/sec ed è impostata per riservare due slot di tempo ravvicinati per la trasmissione master-slave e per l'immediata risposta slave-master.

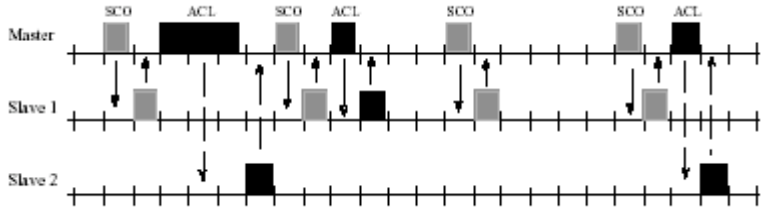

Figura 1.9 Esempio di Trasmissione in una Piconet

Il secondo tipo di collegamento invece è una connessione tra il master e tutti gli slave appartenenti alla Piconet.Un canale ACL supporta la trasmissione punto-multipunto tra il master e gli slave.
Come già precedentemente definito il canale è gestito secondo uno schema a polling, nel quale il master decide qual è il solo slave ad accedere al canale mandando un semplice pacchetto.
Per i link di tipo SCO il master effettua un continuo polling sul rispettivo slave, mentre per i link ACL il polling è asincrono. Nella figura 1.9 possiamo vedere un tipico pattern di trasmissione in una Piconet dove sono presenti un master e due slave. Il primo slave possiede collegamenti verso il master sia di tipo SCO sia di tipo ACL mentre il secondo slave possiede solamente un link di tipo ACL verso il master. In questo esempio sui link di tipo SCO viene periodicamente fatto un polling da parte del master ogni sei slot di tempo, mentre sui link di tipo ACL viene fatto un polling asincrono. Nella figura 1.9 possiamo vedere come il master spedisca un pacchetto che occupa più slot, al secondo slave, che replica con un pacchetto che occupa un solo slot di tempo poiché gli slot successivi sono occupati dai link di tipo SCO.

1.2.3 Modelli per le reti Wireless

Le reti Wireless mobili si possono suddividere in 3 modelli principali :

- Reti cellulari
- Reti cellulari virtuali (VCN)
- Reti Ad-Hoc

Ovviamente ogni modello ha i suoi vantaggi. Nel modello relativo alle reti cellulari per esempio, certamente il grande vantaggio è la semplicità del processo di gestione della mobilità, mentre nelle reti Ad-Hoc è ottimale il bilanciamento del traffico di rete. [Bal] Nelle sezioni successive forniremo una rapida descrizione di questi tre modelli.

1.2.3.1 Reti Cellulari

Un modello a rete cellulare permette di coprire una certa area che a sua volta può essere divisa in ulteriori celle più piccole ed anche sovrapposte. All'interno di ogni cella possono essere distinte due tipologie di entità; una stazione base fissa (BS) e degli host mobili (MH). Ogni stazione base può comunicare con le altre attraverso una rete cablata. Gli host mobili invece si possono muovere da una cella all'altra e possono comunicare con gli altri nodi solamente attraverso la stazione base all'interno della cella in cui si trovano, stabilendo con essa un link di tipo wireless che ne permetta lo scambio di informazioni. Se l'host mobile destinato a ricevere il messaggio si trova all'interno della stessa cella, sarà la stessa stazione base ad inoltrare il messaggio lungo un altro link wireless, mentre se si trova all'interno di una cella diversa, dovrà spedire il messaggio attraverso la rete cablata che la collega alla stazione base della cella dove si trova l'host destinatario.

1.2.3.2 Reti Cellulari Virtuali

Un modello VCN (Virtual Cellular Network) richiama il funzionamento del modello a rete cellulare, poiché anche in questo caso la rete è suddivisa in celle al cui interno sono presenti stazione base ed host mobili. La differenza molto importante sta nel fatto che in questo modello anche la stazione base è mobile e quindi anche i link fra le diverse stazioni di ogni cella sono di tipo wireless. Queste continuano ad ogni modo a coordinare la comunicazione fra gli host mobili che si trovano ad una certa distanza. Comunque le stazioni base non sono fisse ma variano in base ad algoritmi distribuiti implementati per questo modello.

1.2.3.3 Reti Ad-Hoc

Nelle reti Ad-Hoc non ci sono distinzioni nella tipologia dei nodi. Infatti sono tutti mobili e si coordinano tra loro al fine di prendere delle decisioni. Tutti gli host mobili sono collegati attraverso link di tipo wireless e sono liberi di spaziare in qualunque punto all'interno della rete. A causa della mobilità di ciascun dispositivo, il subset dei nodi che compongono il vicinato di ciascun nodo varia col tempo, cosi come la topologia della rete.

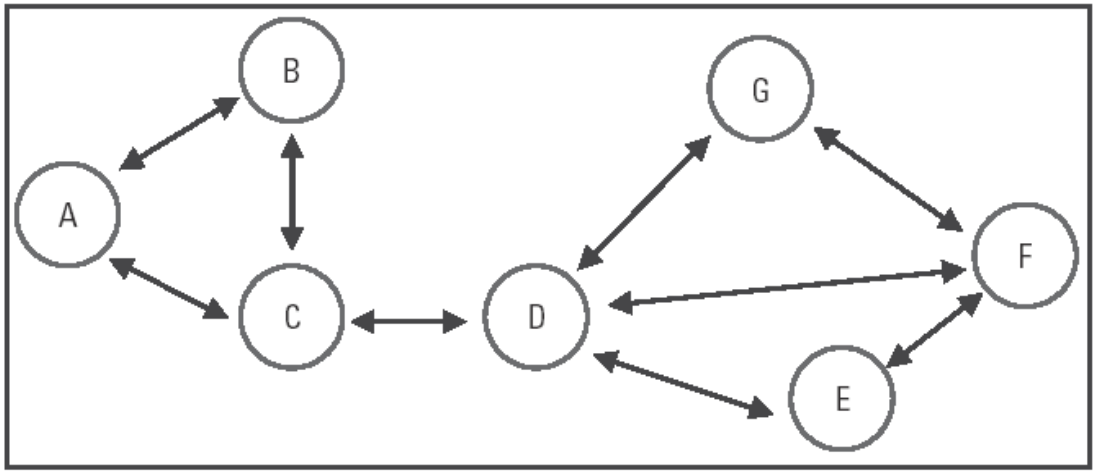

Figura 1.10 Struttura base di una rete Ad-Hoc

Cap. 2

Le Mobile Ad Hoc Networks (MANET)

2.1 Introduzione alle MANET

Una rete mobile Ad-Hoc (MANET) rappresenta un sistema di nodi mobili connessi da link wireless capaci di auto costituire liberamente e dinamicamente architetture di rete arbitrarie e temporanee consentendo ai dispositivi posseduti dagli utenti di interagire tra loro senza alcuna infrastruttura di rete preesistente. Le Manet sono caratterizzate da topologie di rete dinamiche, infatti a causa dell'impossibilità di prevedere gli spostamenti dei nodi e quindi dei rapidi cambiamenti degli scenari di rete, le informazioni possedute dai nodi e gli stati dei link che li collegano diventano velocemente obsoleti. Di conseguenza è necessario un continuo scambio di dati tra i nodi della rete al fine di mantenere aggiornate le informazioni di rete. Le comunicazioni che avvengono tra i vari dispositivi sono multihop; di conseguenza ogni nodo può trovarsi nella condizione di trasmettere un pacchetto, riceverlo, oppure inoltrarlo verso un'altra destinazione. In questo modo i messaggi spediti da un nodo trasmettitore possono raggiungere una destinazione grazie al passaggio attraverso numerosi nodi (hop) che ne hanno semplicemente fatto il "forward". Inoltre i link che collegano i vari dispositivi possono essere asimmetrici, cioè le comunicazioni tra due nodi qualsiasi di una rete possono differire a seconda di chi è il nodo che trasmette il messaggio; per esempio se un nodo *n* è nel range di trasmissione di un altro nodo *m*, l'inverso può non essere vero. Questo deriva dal fatto che essendo i dispositivi comunicanti eterogenei, il loro raggio di trasmissione può essere diverso e quindi una stazione che ha raggio maggiore può riuscire a trasmettere ad una seconda stazione, mentre se quest'ultima ha un raggio di trasmissione minore vi è la possibilità che il secondo dispositivo non riceva alcun pacchetto spedito dall'altro. Le trasmissioni fra le varie stazioni inoltre sono vincolate dall'energia dei nodi, infatti i nodi mobili nelle reti ad-Hoc fanno pieno affidamento sul tempo di vita della loro batteria e quindi è molto importante lo sviluppo di criteri per il risparmio e l'ottimizzazione dell'energia consumata nelle operazioni di trasmissione. Al contrario delle reti cellulari dove vi erano le stazioni base che coordinavano le trasmissioni tra i vari dispositivi, nelle reti Ad-Hoc le operazioni sono decentralizzate, cioè vista l'assenza di infrastrutture di rete preesistenti non vi è la necessita di controller centralizzati. Inoltre rispetto alle reti cablate, la banda a disposizione dei dispositivi wireless è limitata, cioè i link wireless hanno una minor capacità, riuscendo quindi a trasportare una quantita di dati minore per unità di tempo, e il throughput delle comunicazioni è sempre minore

della frequenza radio massima di trasmissione, a causa dell'accesso multiplo o di interferenze nella trasmissione.
Queste caratteristiche permettono di creare una serie di indici per la progettazione di nuovi protocolli di comunicazione che estendono quelli adottati per lo sviluppo di protocolli per le reti cablate.

2.2 Architettura delle reti MANET

Il modello più semplice di architettura per una rete Ad-Hoc è senz'altro il modello piatto (flat). [Ami03] Questa architettura è caratterizzata dal fatto che tutti i nodi della rete si trovano allo stesso livello di comunicazione e non ci sono legami gerarchici fra le varie stazioni. Inoltre i nodi hanno le stesse responsabilità e tutti partecipano nell'instradamento dei pacchetti. Il routing in questo tipo di architettura è ottimale poiché la rete tende a bilanciare maggiormente il traffico attraverso più nodi. Quello che manca invece è un processo di gestione della mobilità, che però andrebbe ad aumentare l'overhead all'interno della rete dovuto alla propagazione dei messaggi di controllo attraverso la rete stessa.

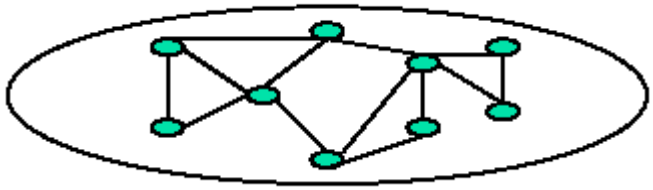

Figura 2.1 Modello flat di una rete Ad-Hoc

Un modello di architettura più complesso è quello gerarchico. Qui i nodi della rete non operano più tutti allo stesso livello ed inoltre sono suddivisi in vari gruppi chiamati *cluster*. All'interno di ogni cluster si possono individuare tre entità gerarchiche; il nodo semplice, il *cluster head* cioè il leader del gruppo, e il nodo gateway che si occupa di far comunicare cluster differenti. Il leader si occupa di prendere le decisioni di controllo per conto degli altri nodi e gestisce il routing del traffico tra due nodi dello stesso cluster. Inoltre il cluster head di ogni singolo gruppo cambia con il tempo, in risposta al movimento dei nodi all'interno della rete. L'obiettivo che ha spinto a suddividere la rete in cluster è quello di ottimizzare l'uso delle risorse al fine di riuscire a ridurre i conflitti sul canale e formare dei percorsi di instradamento che permettano di ridurre le dimensioni della rete. Nella Figura 2.2 possiamo vedere un esempio di "clustering" dove i gateway vengono identificati con la lettera G, i leader dei singoli gruppi con la lettera C, e i nodi semplici con la lettera N.

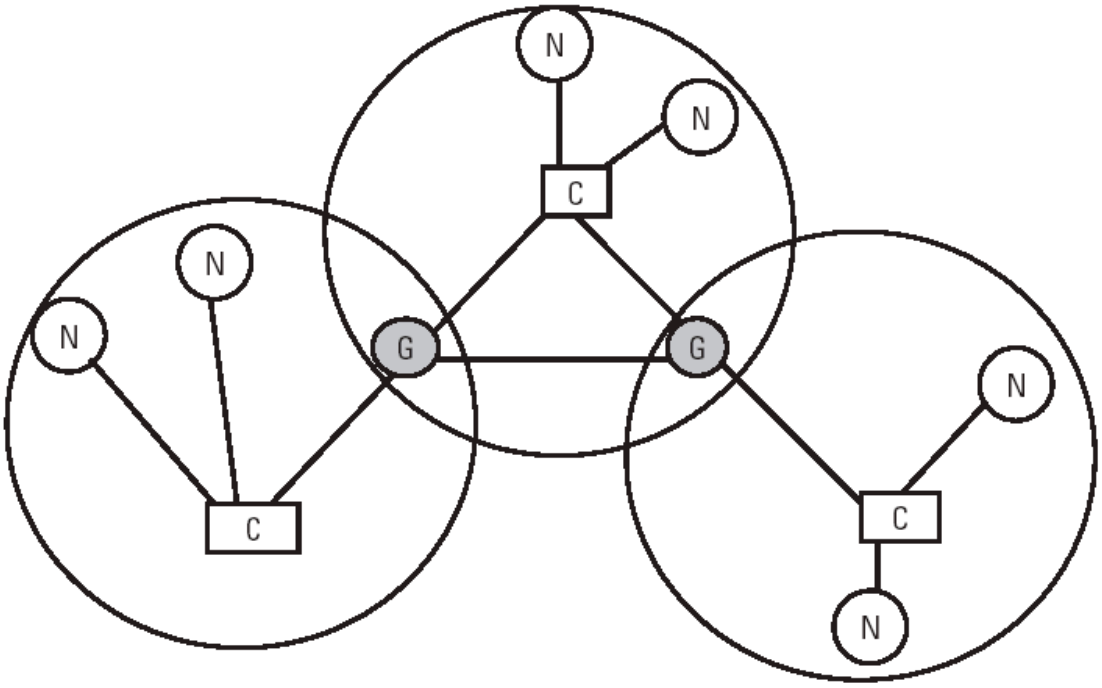

Figura 2.2 Architettura a cluster di una rete Ad-Hoc

Il principale vantaggio dell'architettura gerarchica è senz'altro la semplicità del processo di gestione della mobilità. Infatti al fine di limitare l'impatto che un aggiornamento delle informazioni della rete dei singoli nodi può avere, si è scelto di mantenere un informazione completa dei percorsi di routing relativi solamente ai singoli *cluster*. La comunicazione tra *cluster* diversi avviene mantenendo all'interno dei leader una tabella contenente le informazioni di tutti i nodi facenti parte del *cluster*. In questo modo per determinare dove si trova un determinato nodo vengono fatte delle query a tutti i leader dei gruppi che devono rispedire la risposta alla sorgente della query stessa. Naturalmente questo tipo di architettura deve prevedere dei protocolli particolari per poter scegliere di volta in volta i cluster head e i gateway di ciascun cluster L'inconveniente in un modello di questo tipo è la possibile riduzione del throughput a causa del fastidioso problema del "collo di bottiglia" dovuto al passaggio obbligato da parte di tutti i nodi del cluster, ed una minor robustezza del sistema dovuta alla possibilità di guasto dei nodi chiave come cluster head o gateway.

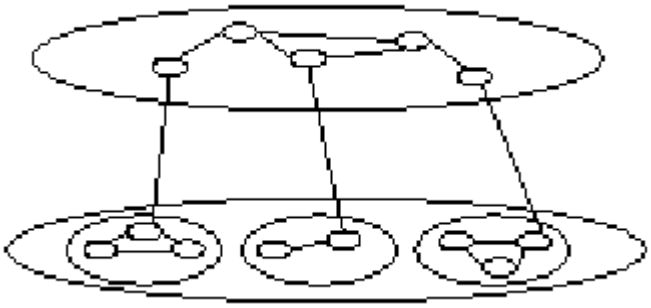

Figura 2.3 Architettura Gerarchica di una rete Ad-Hoc

2.3 Protocolli Di Routing Unicast nelle reti Ad-Hoc

Una prima tipologia di protocolli di routing unicast è quella derivante dall'uso di tecniche particolari mirate ad importare il modello dei protocolli di routing convenzionali usati nelle reti cablate, per le reti Ad-Hoc. [Moh03] Questi protocolli vengono denominati proattivi. L'uso di questo termine fa riferimento all'abilità del protocollo di calcolare tutti i possibili percorsi di rete indipendentemente dal loro effettivo utilizzo. Questo è molto importante poiché permette di evitare ritardi nella trasmissione dovuti ai calcoli dei percorsi da seguire per raggiungere la destinazione desiderata. Un approccio differente è stato invece seguito nello sviluppo di un'altra tipologia di protocolli unicast, cioè quello di calcolare un percorso di rete solamente quando è strettamente necessario per la trasmissione dei dati. Infatti in questo caso se i nodi della rete non generano alcun traffico di dati, l'attività di routing è totalmente assente. Questo tipo di protocolli sono denominati reattivi, e sono caratterizzati dall'eliminazione delle tabelle di routing convenzionali che ogni nodo memorizzava nella sua cache e di conseguenza anche della necessità dei loro aggiornamenti dovuti a cambiamenti nella topologia della rete. L'attività di routing può essere scissa in tre fasi principali, cioè il "path discovery", il "path maintenance" e il "path deletion". La prima fase di "discovery" è quella che si occupa del reperimento delle informazioni per poter poi instradare i pacchetti verso la destinazione desiderata, ed è caratterizzata da un meccanismo di "query-reply" attraverso l'utilizzo del flooding. Al termine di questa fase i nodi della rete si trovano in un nuovo stato di routing nel quale memorizzano tutti i percorsi acquisiti durante la fase di discovery. Queste informazioni vengono poi conservate attraverso la procedura di mantenimento (path maintenance) per un certo periodo di tempo, dopodiché vengono cancellate attraverso la procedura di "path deletion".

2.3.1 Protocolli di Routing Proattivi

Uno dei primi protocolli proattivi ad essere stato sviluppato è il *Destination Sequenced Distance Vector* (DSDV). Qui ad ogni elemento della tabella di routing viene aggiunto un numero di sequenza che indica se il percorso è obsoleto o aggiornato; la presenza di un numero dispari sta ad indicare che la destinazione è irraggiungibile. Gli aggiornamenti dei percorsi vengono spediti secondo un intervallo di tempo prefissato, oppure ogni volta che c'è un cambiamento nella topologia di rete aggiornando il numero di sequenza. Per ogni aggiornamento ricevuto da un nodo relativo al percorso verso una certa destinazione, il corrispondente elemento all'interno della tabella di routing viene aggiornato solamente se il percorso ricevuto ha un numero di sequenza più recente, oppure se a parità di numero di sequenza viene individuata una metrica migliore. Un elemento della tabella di routing viene cancellato nel caso in cui il nodo non riceve più aggiornamenti relativi a quel nodo per un certo intervallo di tempo.

Un altro protocollo è il *Optimized Link State Routing* (OLSR). L'innovazione introdotta da questo protocollo è quella di introdurre dei nodi detti MPR (MultiPoint Relay) per effettuare un flooding verso tutta la rete in modo molto efficiente in modo da ridurre i messaggi duplicati all'interno di una stessa regione. Ogni nodo i seleziona, indipendentemente dagli altri nodi, un sottoinsieme minimo di nodi MPR, MPR(i), dall'insieme dei suoi vicini distanti 1 hop. Inoltre ogni nodo che si trova ad una distanza di due hop dal nodo i deve avere un link simmetrico verso i nodi MPR compresi in MPR(i). In questo modo come è possibile vedere nella Figura 2.2 come il flooding sia realizzato efficientemente, infatti quando un qualsiasi nodo i vuole effettuare un flooding verso la rete, prima spedisce il messaggio solamente ai nodi in MPR(i) che poi provvederà ad inoltrare il messaggio agli altri MPR fino al raggiungimento di tutti i nodi. Per poter effettuare tutto ciò naturalmente ciascun nodo deve memorizzare i dati relativi ai propri vicini, le informazioni relative alla topologia di rete e la tabella di routing.

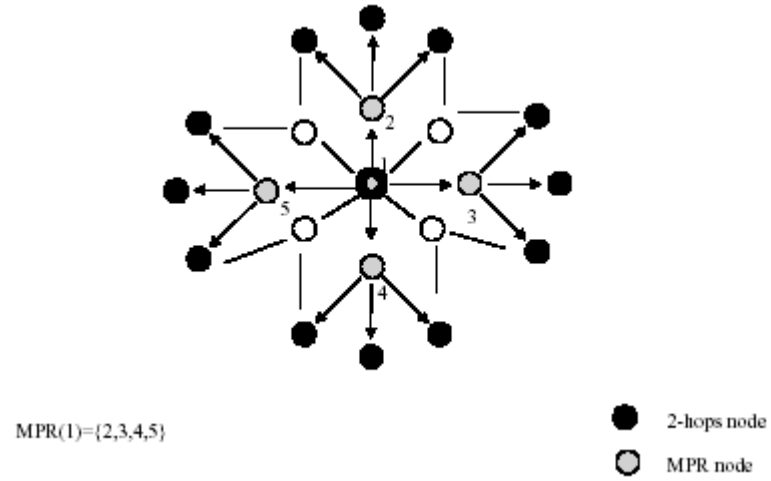

Figura 2.2 Un esempio di flooding usando nodi MPR

Un ulteriore protocollo è il Fisheye State Protocol (FSR), molto simile ai protocolli Link State in quanto ogni nodo memorizza una tabella contenente la topologia della rete,oltre ad una tabella di routing e alla lista dei suoi vicini. Al contrario del Link State però dove viene fatto il flooding degli aggiornamenti dei percorsi a tutta la rete, nel protocollo FSR i pacchetti vengono scambiati solamente con i vicini, mentre i numeri di sequenza vengono usati, come nel protocollo DSDV, per indicare quanto sono recenti le informazioni.
Un messaggio di aggiornamento dei percorsi di rete mantiene al suo interno sia l'indirizzo della destinazione sia la lista dei vicini. Inoltre nel protocollo FSR viene introdotto il concetto di scopo di un certo nodo i , definito come il numero di nodi che possono essere raggiunti mantenendosi ad una distanza massima di h hops dal nodo considerato (es. Figura 2.3).
I percorsi di rete vengono quindi aggiornati con frequenze maggiori per quei nodi il cui scopo è più piccolo, mentre vengono aggiornati con frequenze minori quando lo scopo è più grande.

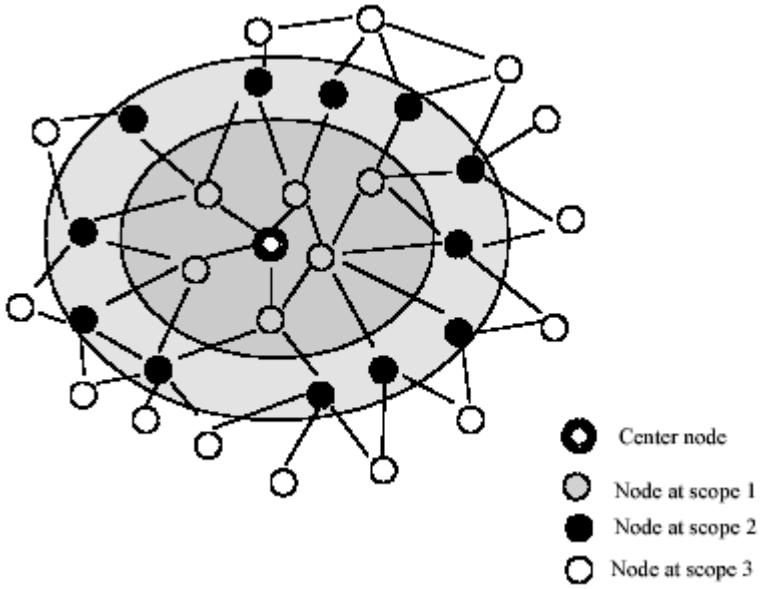

Figura 2.3 Un esempio di scopo nel protocollo FSR

2.3.2 Protocolli Di Routing Reattivi

Al contrario dei protocolli di tipo Proactive, in quelli di tipo Reactive il recupero dei percorsi di rete è effettuato su richiesta del nodo che deve trasmettere. Questi protocolli calcolano i percorsi di rete prima della trasmissione dei dati, infatti se non viene generato alcun traffico di dati da parte dei nodi, l'attività di routing è completamente assente.
Uno di questi protocolli è il Dynamic Source Routing (DSR), il quale è composto da due meccanismi principali: il Route Discovery e il Route Maintenance. Il primo adotta due pacchetti di controllo denominati Route Request (RREQ) e Route Replay (RREP), che vengono generati da una sorgente S che deve spedire un messaggio ad un nodo destinazione D e non ha il percorso in memoria. Viene fatto un flooding a tutta la rete del pacchetto RREQ, il quale contiene al suo interno l'indirizzo della sorgente,l'indirizzo del nodo destinazione, un identificativo della richiesta e un record per memorizzare il percorso.
Quando un nodo intermedio riceve questo pacchetto esso ha 3 diverse alternative. Nel caso abbia in memoria il percorso per raggiungere la destinazione può spedire un pacchetto RREP al nodo sorgente aggiornando il record relativo al percorso da seguire. Se ha già ricevuto il pacchetto precedentemente può semplicemente scartarlo, altrimenti può aggiornare il record contente il percorso con il proprio id e poi effettuare il broadcast del pacchetto ai propri vicini (Figura 2.4). Quando finalmente il nodo destinazione riceve il messaggio RREQ può ritrasmettere un pacchetto RREP alla sorgente attraverso il percorso

ottenuto invertendo quello contenuto nel record del pacchetto ricevuto (Figura 2.5).

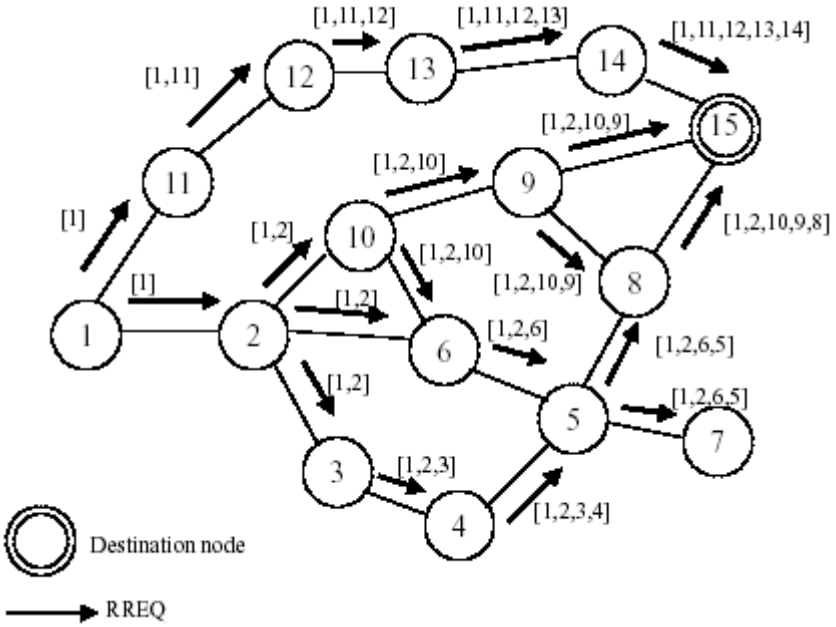

Figura 2.4 Un esempio di propagazione del pacchetto RREQ

Nel caso in cui un nodo intermedio invece rileva che un link nel percorso verso la destinazione è rotto, manda un messaggio di errore alla sorgente del messaggio che poi provvederà a rilanciare una nuova fase di Route Discovery.

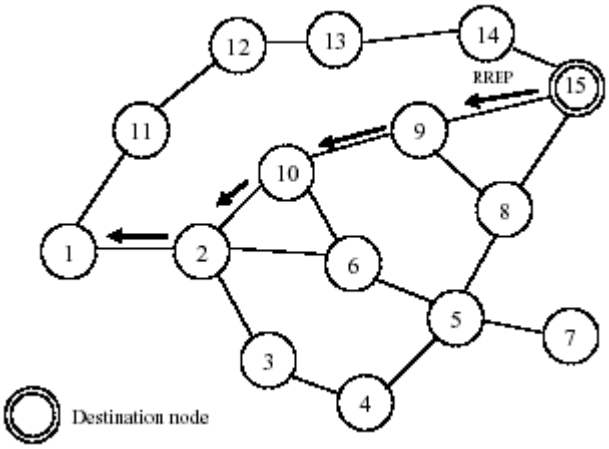

Figura 2.5 Propagazione del pacchetto RREP

Un altro protocollo di tipo reactive è il Ad Hoc On Demand Distance Vector (AODV) che usa i numeri di sequenza come nel protocollo DSDV per numerare i percorsi di rete mentre eredita dal protocollo DSR il meccanismo del Route Discovery. L'invio dei pacchetti RREQ (Figura 2.6) e RREP (Figura 2.7) è identico al protocollo DSR. La differenza

sostanziale è che il percorso che viene generato man mano verso la destinazione viene memorizzato all'interno dei singolo nodi e non all'interno del pacchetto.

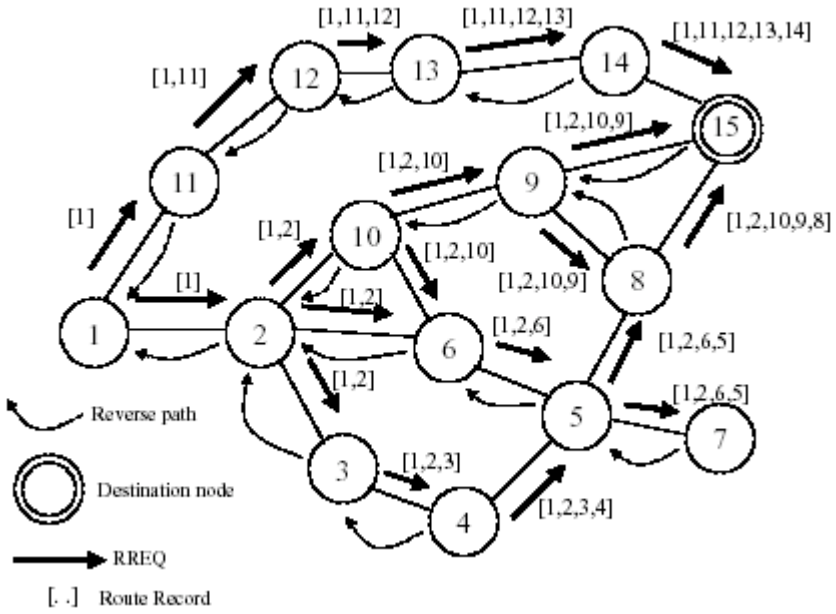

Figura 2.6 Invio del pacchetto RREQ nel protocollo AODV

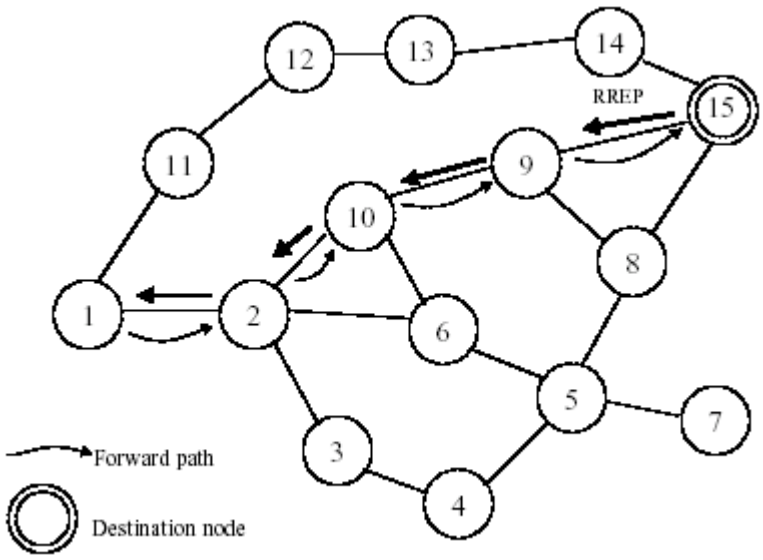

Figura 2.7 Invio del pacchetto RREP nel protocollo AODV

Il mantenimento dei percorsi di rete è reso possibile grazie all'invio periodico di messaggi HELLO verso tutti i nodi.
Un altro protocollo è il Adaptive Distance Vector (ADV), il quale appartiene alla categoria del Distance Vector ma con alcune caratteristiche dei protocolli con routing on-demand. In questo protocollo vengono mantenuti in memoria solamente i percorsi verso nodi ricevitori

“attivi”. Un nodo viene etichettato come “attivo” se esso è la destinazione di una qualsiasi delle connessioni attualmente aperte. Infatti prima che una trasmissione possa avere luogo il nodo sorgente fa un flooding di un pacchetto di controllo *init-connection* evidenziando la necessità di stabilire una connessione con il nodo destinazione. Allo stesso modo quando una connessione deve essere chiusa il nodo sorgente deve mandare un pacchetto di controllo *end-connection.* Questo protocollo inoltre riduce l’overhead dovuto al routing variando la frequenza e la dimensione degli aggiornamenti dei percorsi di rete in risposta al traffico elevato e alla mobilità dei nodi.
Il protocollo Temporally Ordered Routing Algorithm (TORA) è invece progettato per reagire efficientemente ai cambiamenti topologici. Il principale obiettivo non è quello di trovare dei percorsi ottimali, ma percorsi stabili e facilmente riparabili. Quando un nodo spedisce un pacchetto il protocollo costruisce un grafo sulla base delle direzioni logiche assegnate ai singoli nodi. Se per esempio (i, j) rappresenta un link del grafo, i viene definito come nodo superiore mentre j come nodo inferiore. Inoltre il grafo ha l’importante proprietà che c’è un solo nodo di tipo *sink* rappresentato dal nodo destinazione, mentre gli altri nodi hanno almeno un link uscente.
Il funzionamento del protocollo può essere diviso in tre fasi principali: route discovery, route maintenance, e route deletion. Nella prima fase vengono scambiati i messaggi RREQ e RREP e viene assegnata ai singoli nodi la propria direzione logica. Alla fine di questa fase viene costrutito il grafo (DAG).
Nella seconda fase relativa al mantenimento dei percorsi, eventuali cambiamenti nella topologia di rete possono non generare alcuna reazione. Infatti se per esempio il link superiore di un nodo si rompe, ma il link inferiore è ancora disponibile il nodo destinazione verrà raggiunto attraverso il percorso alternativo senza bisogno di alcuna riparazione. Al contrario se il nodo si accorge che non ha un link inferiore allora inizia la terza fase in cui vengono riassegnate le direzioni logiche a tutti i nodi della rete e il grafo viene ricostruito sulla base della nuova topologia.
Il protocollo TORA inoltre è molto abile nell’individuare eventuali partizioni di rete e in questo caso viene effettuato il flooding di un pacchetto *clear*, che resetta tutti i percorsi di routing dei vari nodi.

Cap. 3
Routing Multicast in ambienti MANET

Il routing multicast permette ai messaggi spediti da una o più sorgenti di essere consegnati a multiple destinazioni. In particolare questa tecnica consente di trasmettere un singolo pacchetto ad un gruppo di nodi. Un gruppo multicast è un insieme di host che si sono registrati a quel particolare gruppo; questo significa che se un host vuole ricevere un messaggio di tipo multicast, esso si deve prima di tutto registrare ad un gruppo e a questo punto tutti i membri di quel gruppo riceveranno il messaggio. Tipicamente l'appartenenza ad un gruppo multicast è dinamica; un host può lasciare o entrare in un gruppo in qualsiasi momento. Non c'è nessuna restrizione sulla locazione o sul numero di host che devono appartenere ad un certo gruppo ed inoltre un host può essere contemporaneamente membro di più gruppi. Al contrario dei protocolli di routing per le reti fisse, quelli per le reti Ad-Hoc devono avere un diverso range di caratteristiche, quali supporto della mobilità e basso consumo di risorse quali banda, memoria, energia. Infatti la mobilità degli host può apportare cambiamenti alla topologia della rete aumentando la volatilità dell'informazione nella rete mentre la limitata energia della batteria porta gli utenti a disconnettersi molto frequentemente al fine di limitare il consumo di energia. Nelle reti Ad-Hoc ci sono tre tipi di approcci nello sviluppo di algoritmi di multicast :

- Un'approccio Naive
- Un'approccio Proactive
- Un approccio in cui i percorsi tra i nodi sono creati su richiesta.

Nel caso Naive infatti ogni nodo in trasmissione spedisce il messaggio a tutti i propri vicini che a loro volta lo ritrasmettono a tutti i loro vicini e cosi via, effettuando il cosiddetto "flooding". Questo può portare ad un aumento esponenziale del traffico di rete a meno di introdurre dei meccanismi per il controllo dei duplicati al fine di evitare inutili ritrasmissioni. Nell'approccio proattivo vengono calcolati tutti i possibili percorsi per tutte le destinazioni e queste informazioni vengono poi memorizzate nelle tabelle di routing possedute da ogni nodo. Per tenere aggiornate queste tabelle le informazioni sono periodicamente distribuite attraverso la rete.
Nel terzo caso l'idea di fondo è basata su un meccanismo del tipo "query-response".Nella fase di query il nodo esplora l'ambiente di rete. Una volta che la query raggiunge la destinazione nella fase di response si stabilisce il percorso tra questa e la sorgente.

3.1 ODRMP (On Demand Multicast Routing Protocol)

Nel protocollo ODRMP, l'appartenenza o meno ai gruppi e il calcolo dei percorsi di multicast sono stabiliti ed aggiornati su richiesta dalla sorgente del messaggio. Come esempio consideriamo quello di Figura 2.1(a) in cui i nodi vengono rappresentati col colore nero se fanno parte di un gruppo, col colore grigio se sono solo nodi che inoltrano il pacchetto (Forwarding Group), e col colore bianco i nodi che non appartengono a nessun gruppo. In questo esempio il nodo **S** spedisce un pacchetto ad un gruppo multicast. Non avendo però nessuna conoscenza sugli eventuali percorsi da intraprendere per raggiungere tali nodi farà un flooding del pacchetto di controllo **JOIN_DATA** attraverso la rete. Quando un nodo intermedio riceve un pacchetto di tipo **JOIN_DATA** esso memorizza l'id della sorgente e il numero di sequenza del messaggio al fine di evitare eventuali duplicati. Inoltre ogni membro di un gruppo multicast nel ricevere questo dato fa un broadcast di una **JOIN_TABLE** che contiene la sorgente del pacchetto e il prossimo nodo **N** nel percorso tra il membro del gruppo e il nodo **S**.

Quando un nodo riceve una **JOIN_TABLE** , questo controlla se l'id del nodo successivo memorizzato nella tabella corrisponde al suo. In caso affermativo questo significa che il nodo si trova nel percorso della sorgente S e che quindi fa parte del "Forwarding Group"; il nodo quindi setta il **FG_FLAG** e fa il broadcast della propria **JOIN_TABLE** a quei nodi che fanno match con le entry della tabella. Il campo Id relativo al nodo successivo di questa tabella viene ottenuti estraendo l'informazione dalla routing table del nodo stesso.

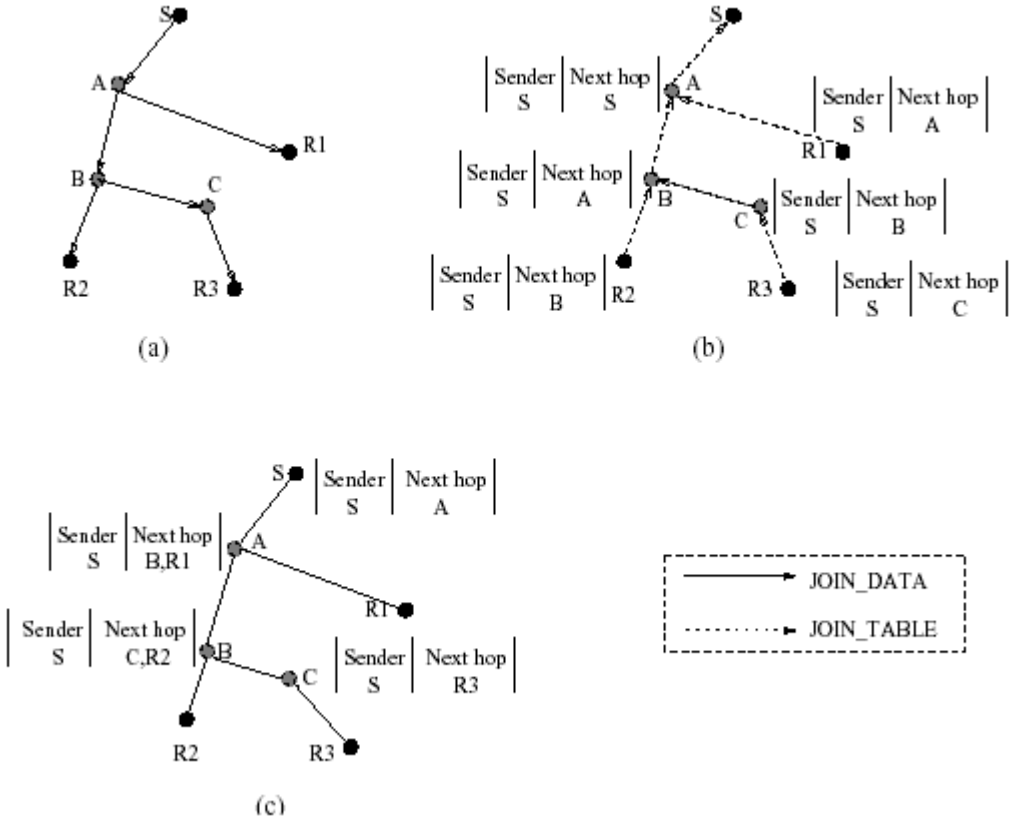

Figura 2.1 ODMRP : (a) Propagazione di un pacchetto **JOIN_DATA**.
(b) Propagazione di un pacchetto **JOIN_TABLE**. (c) Tutte le tabelle complete

La figura 2.1(b) mostra come i membri del "Forwarding Group" propagano la **JOIN_TABLE** fino a che questa non raggiunge la sorgente **S**. Inoltre un nodo dopo aver ricevuto questi dati costruisce la sua tabella di multicast per poter ritrasmettere futuri pacchetti di tipo multicast. Per esempio quando il nodo B riceve una **JOIN_TABLE** dal nodo R2 aggiungerà il nodo R2 come suo prossimo nodo. Nella figura 2.1(c) viene mostrata la tabella di multicast finale di ogni singolo nodo.
Nel protocollo ODMRP non sono necessari pacchetti di controllo per unirsi ad un gruppo o lasciarlo; semplicemente basta non spedire più messaggi di tipo **JOIN_DATA.** Inoltre un nodo appartenente al "Forwarding Group" cessa la sua attività se non riceve alcuna **JOIN_TABLE** entro un certo intervallo di Time-out **.**

3.2 Multicast AODV (MAODV)

Nel protocollo MAODV (Multicast Ad-Hoc On Demand Vector) quando un nodo mobile desidera unirsi ad un gruppo di multicast oppure spedirvi un messaggio, ma non ha un percorso per raggiungerlo esso deve mandare un messaggio **RREQ** (route request) di tipo join. Per rispondere

a questo tipo di dato un nodo deve appartenere obbligatoriamente al gruppo multicast desiderato, mentre se non è di tipo join allora qualsiasi nodo con un percorso abbastanza recente verso il gruppo di multicast può rispondere. Se un nodo intermedio invece riceve un **RREQ** di tipo join destinato ad un gruppo di cui non è membro oppure riceve un **RREQ** ed esso non ha alcun percorso verso quel gruppo, fa il broadcast del pacchetto ricevuto a tutti i suoi vicini. Nella Figura 2.2 possiamo vedere un possibile esempio della propagazione di un pacchetto **RREQ** all'interno di una rete spedito dal nodo S ad un gruppo di multicast. Inoltre, mentre i messaggi **RREQ** viaggiano all'interno della rete i nodi settano le loro tabelle dei percorsi al fine di ricordarsi il percorso inverso nell'eventualità di rispedire un messaggio alla sorgente. Per i messaggi **RREQ** di tipo join invece viene aggiunta un 'ulteriore entry nella tabella di routing di ciascun nodo.

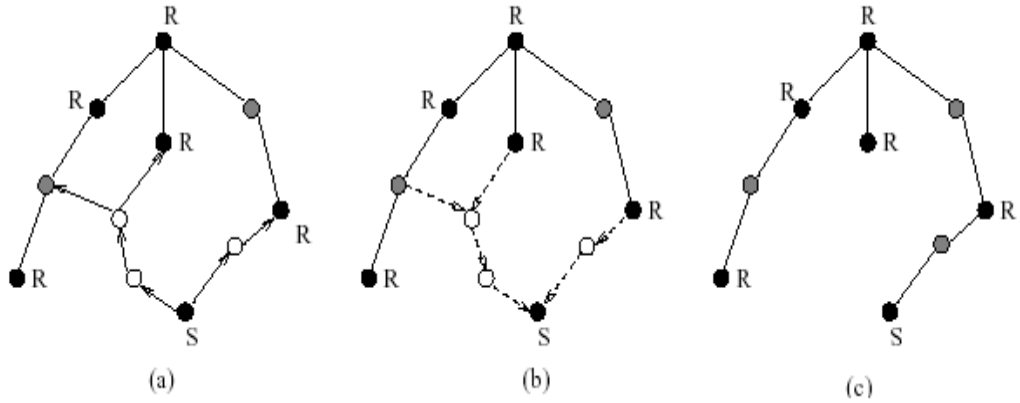

Figura 2.2 Un esempio del protocollo MAODV. (a) La propagazione dei pacchetti RREQ. (b) La propagazione dei pacchetti RREP. (c) L'albero multicast finale

Se un nodo riceve un **RREQ** di tipo join destinato ad un gruppo multicast esso può rispondere a condizione che esso sia un membro del gruppo in questione e il suo numero di sequenza per il gruppo multicast sia maggiore o uguale a quello contenuto all'interno del messaggio **RREQ**. Il nodo che risponde inoltre aggiorna le proprie tabelle dei percorsi aggiungendovi l'informazione relativa al prossimo nodo sul percorso, e poi manda un messaggio unicast **RREP** (rappresentato nella Figura 2.2(b) da frecce tratteggiate) di risposta ad S. Tutti i nodi lungo il percorso verso S che ricevono il pacchetto **RREP** aggiornano le loro tabelle di route e route multicast con il nodo dal quale hanno ricevuto il pacchetto. Quando il nodo S spedisce un pacchetto di tipo **RREQ** per un gruppo multicast, questo spesso riceve più di una risposta. Il nodo S però conserva solamente per un certo tempo il percorso con il più grande numero di sequenza e il minor numero di hop dalla sorgente alla

destinazione e scarta tutti gli altri. Al termine di questo periodo di tempo abilità il successivo hop corrispondente al percorso scelto nella propria tabella di multicast e manda un messaggio di attivazione al nodo selezionato. Il nodo che riceve questo messaggio di attivazione inserisce S nella sua tabella dei percorsi multicast; inoltre se questo appartiene all'albero non propaga oltre il pacchetto ricevuto. Se invece non è un membro dell'albero questo avrà ricevuto più risposte da parte dei suoi vicini; a questo punto il nodo conserva il migliore hop lungo il percorso verso il gruppo multicast e spedisce un messaggio di attivazione a quello selezionato. Questo procedimento si ripete fino a che non si raggiunge un membro dell'albero. La figura 2.2 (c) mostra l'albero finale.
Il primo nodo di un gruppo multicast diventa il leader di quel gruppo, cioè è responsabile del mantenimento del numero di sequenza del gruppo e del broadcast di questo numero a tutti i membri del gruppo attraverso un messaggio hello per l'albero considerato. Questo messaggio contiene delle informazioni che indicano l'indirizzo IP per quel gruppo ed i numeri di sequenza, i quali vengono incrementati ad ogni invio di questo messaggio, di tutti i gruppi del quale il nodo è leader. Queste informazioni vengono sfruttate dai nodi per aggiornare le informazioni contenute all'interno delle loro tabelle dei percorsi.

3.3 FGMP (Forwarding Group Multicast Protocol)

Questo protocollo al contrario di altri della stessa tipologia non tiene traccia dei link presenti tra i vari nodi, ma dei gruppi di nodi che partecipano all'invio di pacchetti di tipo multicast.
Ad ogni gruppo multicast **G** viene associato quindi un corrispondente gruppo di forwarding **FG.** Ogni nodo nel gruppo **FG** è quindi in grado di effettuare il broadcast dei pacchetti del gruppo **G**, cioè quando un nodo in questo gruppo, riceve un pacchetto di tipo multicast esso farà un broadcast di questo pacchetto ai suoi vicini se questo non è un duplicato di un pacchetto già spedito.
A questo punto solo i vicini che appartengono al gruppo **FG** potranno riconoscere se questo non è un duplicato e quindi farne il broadcast a sua volta ai propri vicini. Nella Figura 2.3 possiamo vedere un esempio di protocollo FGMP con due nodi trasmettitori e due nodi ricevitori, e quattro nodi appartenenti al gruppo **FG** che hanno il compito di inoltrare i pacchetti ai propri vicini.
Per ogni nodo del gruppo **FG** è necessario memorizzare solamente un flag ed un timer, riducendo di molto l'overhead dovuto alle grandi informazioni da memorizzare degli altri protocolli, aumentando cosi la flessibilità e le performance del protocollo stesso. Quando questo flag

viene settato il nodo inizia a trasmettere pacchetti di dati per i nodi del gruppo **G** fino a che non il timer non scade.

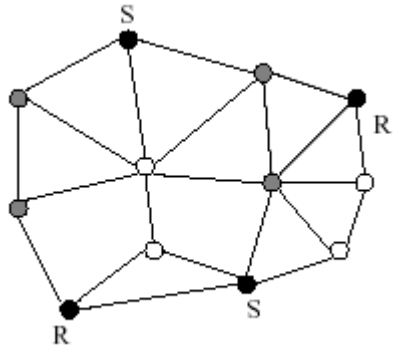

Figura 2.3 Esempio del protocollo FGMP

Il maggior problema nell'uso di questo protocollo è senza dubbio costituire e mantenere aggiornato il set di nodi del gruppo **FG**. Infatti la dimensione di questo gruppo dovrebbe essere il più piccola possibile in modo da diminuire l'overhead del canale di comunicazione, ed il cammino attraverso i nodi **FG** dalla sorgente al ricevitore dovrebbe essere il più breve possibile in modo da aumentare il throughput del sistema. Uno dei modi possibili per poter mantenere aggiornato il gruppo FG può essere quello di lasciare che il destinatario del messaggio effettui periodicamente un flooding delle sue informazioni verso tutta la rete. In questo modo quando un nodo trasmettitore S riceve un pacchetto **RREQ** di tipo join aggiorna la tabella dei membri del gruppo G e nello stesso tempo cancella le entry obsolete della tabella.
Il nodo S trasmetterà pacchetti di tipo multicast solamente se la sua tabella dei membri non è vuota e dopo aver aggiornato la tabella stessa creerà da questa la tabella FW relativa ai nodi del gruppo **FG**. La tabella FW sarà poi trasmessa in broadcast a tutti i suoi vicini che sono ad una distanza di un hop (questa lista viene ottenuta dalle precedenti tabelle di routing). A sua volta ogni vicino creerà la sua tabella FW estraendo dalla tabella ricevuta le entry dove esso è rappresentato come il nodo successivo e di nuovo trasmessa in broadcast ai suoi vicini. Questo procedimento viene ripetuto fino a che tutti i ricevitori R non sono stati raggiunti. Ogni volta che questi nodi ricevono la tabella FW settano il proprio flag ed inizializzano il timer relativo all'invio del pacchetto di forwarding.
Un altro modo possibile di implementare la soluzione sopra citata può essere quello di lasciare al ricevitore invece il compito di fare il flooding delle proprie informazioni attraverso la rete. Questo metodo è molto più efficiente del precedente poiché come è facile notare il numero di nodi S è minore del numero di ricevitori R. Non appena i nodi R ricevono lo

stato del nodo S, spediscono delle “joining table” contenenti l’id del nodo S che ha spedito il messaggio e i nodi intermedi settano i loro flag e il loro timer alla ricezione di queste “joining table”.

3.4 CAMP (Core-Assisted Mesh Protocol)

Molti protocolli si basano sull’uso di alberi di routing, i quali possono risultare estremamente efficienti per la ricerca di percorsi fra una coppia di nodi. Nel caso ci fossero più sorgenti che devono trasmettere informazioni verso la stessa destinazione si possono prendere in considerazione due tipologie di albero; un albero condiviso tra tutte le sorgenti oppure un singolo albero per ogni sorgente in trasmissione. Nel primo caso si ha lo svantaggio che i pacchetti diretti al gruppo multicast sono distribuiti lungo percorsi che non sempre coincidono con il percorso più breve per arrivare alla destinazione. Nel secondo caso invece avendo un singolo albero per ogni sorgente si costringono i nodi appartenenti al gruppo a mantenere un riferimento a tutte le sorgenti per ogni gruppo portando cosi ad un overhead maggiore. Inoltre poiché gli alberi forniscono una connettività minima tra i nodi di un gruppo il fallimento di un nodo richiede ai nodi coinvolti la riconfigurazione dell’albero considerato.

Per risolvere queste problematiche è stato introdotto il protocollo CAMP (Core-Assisted Mesh Protocol) il quale mantiene una rete multicast chiamata “multicast mesh” per la distribuzione delle informazioni all’interno di ogni singolo gruppo. Una rete “mesh” è una sottorete che fornisce almeno un percorso tra ogni sorgente della rete e ogni ricevitore appartenenti al gruppo multicast. Attraverso questo protocollo si è sicuri che i percorsi contenuti in questa sottorete sono sicuramente i percorsi più brevi tra sorgenti e ricevitori. CAMP però non ha percorsi predefiniti attraverso la sottorete mesh, ma vengono costruiti man mano che si procede nella trasmissione. Infatti, un nodo tiene traccia degli identificatori dei pacchetti che ha inoltrato recentemente, e quindi esso inoltra un pacchetto ricevuto da un vicino solamente se l’identificatore di questo messaggio non è nella sua cache e se gli è stato detto da almeno un vicino che quel nodo sia il successore in un eventuale percorso inverso fino alla sorgente di un eventuale gruppo.

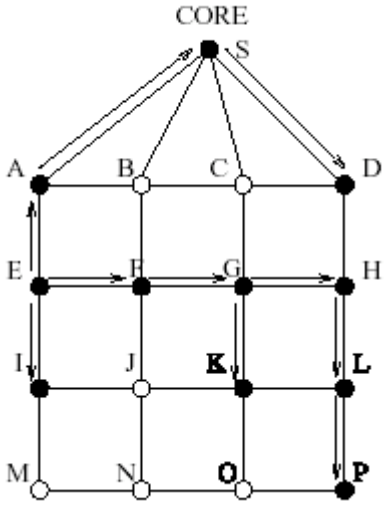

Figura 2.4 Un esempio di flusso di traffico generato dal nodo E in una sottorete Mesh

Nella figura 2.4 (le frecce marcate indicano il flusso del percorso inverso fino alla sorgente) possiamo vedere un esempio di come si evolva il flusso di traffico generato in una sottorete Mesh dal nodo E verso gli altri nodi del gruppo multicast. E' possibile inoltre notare la presenza di un nodo "CORE" S il quale viene usato per limitare il traffico prodotto dai nodi ricevitori per unirsi ai gruppi di multicast. Ci possono essere più CORE per ogni sottorete Mesh, e la loro presenza elimina la necessità di effettuare flooding attraverso la rete, a meno che tutti i CORE siano irraggiungibili da tutti i nodi connessi.

3.5 Altri protocolli di Multicast nelle reti Ad-Hoc

Recentemente sono stati sviluppati numerosi altri protocolli per le reti Ad-Hoc basati sull'utilizzo di nuove tecnologie. **OLAM** (on-demand location aware multicast protocol) per esempio, mira a fornire una completa visibilità della topologia di rete, assumendo che attraverso l'uso di dispositivi per sistemi di posizionamento come quelli GPS, ogni nodo conosca la propria posizione attuale e l'istante di tempo corrispondente, e sia in grado di trasmettere questi parametri, incluso il raggio di trasmissione, a tutti gli altri nodi della rete. Non appena gli altri nodi ricevono il pacchetto aggiornano la propria visione della topologia di rete; in questo modo quando un pacchetto deve essere spedito in multicast ad un gruppo viene usata una euristica per calcolare localmente l'albero per il gruppo multicast in questione basandosi sulla visione della topologia della rete posseduta dal nodo al momento. I risultati mostrano che questo protocollo consegna pacchetti a tutti i nodi del gruppo in

considerazione in più dell' 85% dei casi. Inoltre il protocollo **OLAM** comparato al flooding ha ottenuto un miglioramento di circa il 50% nelle reti multicast. Recentemente è stato presentato un altro protocollo denominato **AMRoute** (Adhoc Multicast Routing Protocol), il quale crea un albero condiviso bidirezionale per lo scambio dei dati, usato solamente dai gruppi di trasmettitori e ricevitori, e un mezzo unicast usato per rappresentare i link che connettono i nodi dell'albero multicast. Inoltre il costo di mantenimento dello stato dei gruppi è gravato solamente dai trasmettitori e dai ricevitori, e la struttura dell'albero non necessita di modifica anche in casi di tipologia dinamica, nei quali è stato possibile vedere come una gran percentuale dei nodi appartenenti ad un gruppo riceve i pacchetti multicast spediti da una eventuale sorgente. Nel 2001 è stato introdotto anche un altro protocollo di routing per le reti MANET molto efficiente, il cui principio di base si basa sulla costruzione di un albero generato dai nodi e una tabella di routing associata ad ognuno di essi. Questo algoritmo è combinato con un altro algoritmo del tipo "prefix routing", il quale ha una tabella di routing molto compatta associata ad ogni stazione; ognuna di queste ha una label che le contraddistingue e una viene scelta per inoltrare il pacchetto fino alla destinazione solamente se la sua label è un prefisso di quella del nodo successivo a cui trasmettere i dati. Nel 2000 è stato introdotto un altro protocollo basato sull'implementazione Real-Time denominato **WARM** (Wireless Ad Hoc Real Time Multicast), in cui le connessioni tra i vari nodi sono Real Time consentendo un significativo aumento della QoS (quality of service) in termini di larghezza di banda. Sempre nel 2001 invece è stato portato all'attenzione un altro protocollo di multicast, i cui obiettivi sono quelli di fornire un basso overhead della comunicazione fra i nodi ed un alta efficienza del multicast. Per far si che tali desideri si realizzino è necessario seguire tre punti chiave principali :

- Invocazione on-demand del setup dei percorsi tra i nodi e utilizzo di processi per il monitoraggio degli stessi.
- Implementazione di un processo che abbia il compito di inizializzare i percorsi per quei nodi che desiderano unirsi ad un gruppo multicast in modo da ridurre il numero di hop da attraversare per raggiungere la destinazione desiderata.
- Implementazione di un processo per l'ottimizzazione dei vari percorsi che provveda anche a rimuovere eventuali nodi ridondanti al fine di effettuare un forward dei pacchetti con maggiore efficienza.

Cap. 4
I Protocolli di Gossip

4.1 Introduzione ai protocolli di Gossip

Negli ultimi anni gli algoritmi di routing basati sul Gossip hanno guadagnato il riconoscimento di metodologie adatte alla progettazione di schemi di comunicazione robusti e scalabili. La scalabilità deriva dal fatto che questa tipologia di protocolli non richiede una sincronizzazione come quella per esempio che avviene nei tradizionali protocolli di multicast. Nei protocolli non deterministici come il Gossip la dinamica dell'esecuzione non è determinabile a priori. Tuttavia la natura estremamente mobile delle MANET suggerisce proprio approcci di tipo probabilistico; invece di ritrasmettere in ogni caso, come avviene nel flooding, un nodo che partecipi al broadcast quando riceve un pacchetto lo ritrasmette con probabilità p, o altrimenti lo scarta. Tutti i protocolli non deterministici, sono in genere statici, ovvero o eseguono immediatamente la ritrasmissione oppure interrompono il flooding: infatti non necessitano di alcun intervallo di attesa per la ritrasmissione e quindi hanno una latenza minore dei protocolli deterministici. Questa caratteristica li rende quindi degli ottimi candidati per la fase di discovery dei protocolli di routing reattivi o per applicazioni interattive. Essi realizzano una sorta di meccanismo di replicazione molto utile per ovviare ad eventuali guasti di stazioni della rete o dei link che le collegano, poiché un nodo riceve una copia multipla del messaggio da parte di più stazioni. Nessun nodo all'interno della rete ha un ruolo specifico e quindi il danneggiamento di una stazione non impedisce agli altri di continuare a trasmettere messaggi.

4.2 Caratteristiche principali ed affidabilità

I protocolli di Gossip hanno due caratteristiche principali: scalabilità, ed un degrado della rete molto lento e quindi gradevole.
Per quanto riguarda la scalabilità possiamo dire che le performance di questo tipo di protocolli non diminuiscono rapidamente quanto la crescita del numero di nodi del sistema ed inoltre ogni singola stazione trasmette un numero fisso di pacchetti indipendentemente dal numero dei dispositivi presenti all'interno della rete [Dav].
Inoltre non è affatto difficile aggiungere o rimuovere nuove stazioni all'interno della rete. In entrambi i casi queste operazioni possono essere effettuate utilizzando il protocollo di Gossip stesso.

Per quanto riguarda il degrado della rete, in molti protocolli di broadcast viene definita una certa soglia S di guasti che si possono verificare all'interno della rete. Se si rimane al di sotto di questa soglia allora il protocollo funzionerà correttamente altrimenti se la si oltrepassa il funzionamento non sarà più garantito.
L'affidabilità del protocollo è allora uguale alla probabilità che non si verifichino almeno S+1 guasti. Il calcolo di questa probabilità comunque, può essere difficile da fare e l'elaborazione a sua volta può essere basata su parametri che sono difficili da calcolare. Di conseguenza è perciò un vantaggio avere un protocollo la cui probabilità di funzionare correttamente non decresca rapidamente quanto il numero di guasti, una volta passata la soglia S.

4.3 Gossip e transizione di fase

[And03] Uno dei problemi che il gossip pone è la determinazione del valore di p che garantisca le prestazioni migliori. Un valore troppo elevato infatti aumenterebbe la raggiungibilità teorica (RE) della rete, ma diminuirebbe l'efficienza teorica (SRB), mentre un valore troppo basso diminuirebbe RE e aumenterebbe SRB; rispetto al flooding viene comunque risparmiata una frazione di ritrasmissioni pari a (1-p).

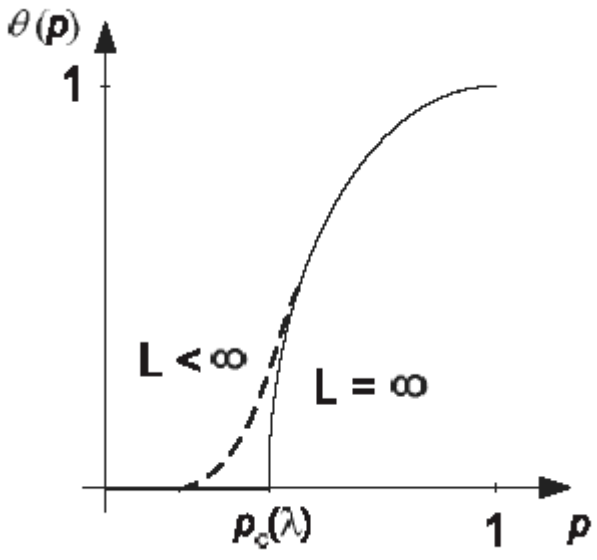

Figura 4.1 Transizione di fase

Tale problema è stato studiato, cercando di applicare alle MANET la teoria della *percolation*, studiata da Broadbent e Hammersley nel 1957 per studiare il flusso di un fluido in un mezzo casuale. Tale teoria è un'importante branca della fisica statistica e ha trovato nel corso degli

anni numerosi campi di applicazione: dallo studio dei sistemi fisici complessi, come l'analisi dell'orientamento dello spin atomico, agli algoritmi epidemiologici per la replicazione dei database, fino alle analisi di diffusione degli incendi boschivi.
Secondo tale teoria esistono dei sistemi in cui la variazione di un determinato parametro, può indurre una variazione improvvisa dello stato globale del sistema. Più precisamente dato un insieme di istanze del sistema, caratterizzate dalla stessa probabilità $p(\lambda)$ di riscontrare una determinata proprietà λ, e una proprietà osservabile del sistema θ, correlata a λ, esiste una soglia $p_c(\lambda)$ che demarca una improvvisa variazione della probabilità $p(\theta)$ (Figura 4.1). La soglia critica alla quale avviene il cambiamento repentino della proprietà osservabile è detta *transizione di fase*; essa è tanto più netta quanto più la dimensione L delle istanze del sistema considerato è elevata.
Uno degli obiettivi della ricerca, ad oggi non ancora raggiunto, è provare che la teoria della percolation e la transizione di fase è applicabile anche al flooding stocastico nelle MANET. Se i risultati di tale ricerca fossero positivi allora, in una MANET di dimensione infinita, esisterebbe un valore critico p_c della probabilità di gossip oltre il quale raggiungerebbe quasi tutti i nodi. Al di sotto di tale valore invece il numero di nodi raggiunto sarebbe finito. Il gossip ha quindi, al variare della probabilità p, un comportamento di tipo *bimodale*. Queste ipotesi sono incoraggiate dagli studi effettuati in passato sui grafi casuali. Un grafo casuale è un grafo generato da un esperimento aleatorio che soddisfi determinate regole. L'insieme di tali regole costituisce il *modello di grafo casuale*. Tra i vari modelli possibili ve ne sono due particolarmente interessanti ai fini dello studio delle MANET:

- **Il modello di Bernoulli (BRG)** genera i grafi G = G(n, p), dove n è il numero dei vertici e p è la probabilità che esista uno spigolo tra una qualunque coppia di vertici.
- **Il modello a raggio fisso (FRG)** genera i grafi G = G(n, R), dove n è il numero di punti posizionati casualmente, secondo una data distribuzione, sul piano euclideo, che costituiscono i vertici del grafo G, mentre R è la distanza tra una coppia di punti (v, w) al di sotto della quale esiste in G lo spigolo e_{ij}.

La teoria dei grafi casuali ha ampiamente studiato i grafi generati secondo il modello BRG: per essi è possibile determinare la soglia critica di un parametro alla quale si verifica la transizione di fase di una qualunque proprietà del grafo esprimibile tramite la logica del primo ordine. Il modello FR è invece quello che meglio modella il grafo di connessione di una MANET; ad oggi purtroppo sono pochi i risultati sulla transizione di fase per tale modello.

È però evidente che la probabilità di gossip p_c, alla quale si verifica la transizione di fase, dipenda dalla topologia della MANET e in particolare dipenda dal grado medio del grado di connessione o, alternativamente dalla densità media dei nodi. In altri termini utilizzare un algoritmo di gossip con probabilità p equivale ad estrarre un sottografo dal grafo di connessione originale della MANET, eliminando alcuni archi e diminuendo cosi la ridondanza del flooding, senza però tenere conto del grado di connessione dei nodi nel sottografo risultante. Non essendo però semplice determinare a priori in una MANET il grado medio di connessione, non è possibile fissare il valore di p che garantisca delle buone prestazioni, cioè quello appena al di sopra di p_c.
Inoltre tutti i risultati teorici si riferiscono sempre a grafi estesi, in cui sia chiaramente osservabile la transizione di fase. Le MANET hanno però, in genere, dimensioni abbastanza limitate e pertanto è necessario considerare gli effetti di bordo osservabili sul confine della rete, dove il grado medio di connessione è minore che sui nodi interni. Anche questo problema suggerisce che la soglia di probabilità p_c debba dipendere dalla densità dei nodi; in tal modo i nodi di bordo risulterebbero più "contagiosi", assicurando cosi la diffusione del broadcast anche sul confine. Queste considerazioni sono alla base di alcuni protocolli euristici il cui obiettivo è adattare il gossip alla tipologia dinamica della MANET.

4.4 GOSSIP1

Gossip1(p,k) è una prima semplice implementazione del gossip. Il gossip può riscontrare dei problemi nella fase di trasmissione iniziale soprattutto quando vi è un numero limitato di vicini intorno al nodo che deve trasmettere; infatti affinché il broadcast abbia successo è necessario raggiungere un certo numero di nodi, la cosiddetta "massa critica". Per tale motivo il broadcast dovrebbe inizialmente diffondersi con probabilità 1 similmente al flooding fino a "k" hops dal nodo di origine. Inoltre è importante notare come il Gossip1(1,1) corrisponda al flooding, mentre il Gossip1(p,0) corrisponda al gossip con probabilità p.

4.5 Gossip basato sulla conoscenza dei vicini

Sia il gossip che il Gossip1 hanno entrambi una scarsa adattabilità alla mobilità, dovuta al fatto che il gossip a probabilità costante non prende in considerazione la topologia variabile delle MANET. Essendo quindi la probabilità di trasmissione molto elevata per avere buone prestazioni anche nei casi peggiori di grafi sparsi è inevitabile che avere un numero di ritrasmissioni ridondanti. Una prima ottimizzazione di questi protocolli consiste nel rendere la probabilità di trasmissione funzione del numero di

vicini di un nodo seconda la relazione $p(k) = p_0/k$ dove p(k) rappresenta la probabilità di ritrasmissione, p_0 è una costante indipendente dalla topologia di rete considerata, e k è il numero di vicini del nodo in quel determinato istante.
Una variante della tecnica precedente può essere può essere il Gossip2(p1, k, p2, n) il quale usa un modello a soglia per tentare di risolvere il problema dei grafi sparsi. Per i primi k hop il funzionamento è identico a quello del Gossip1, ma terminata questa fase se il numero dei vicini è maggiore di n allora viene eseguito il gossip con probabilità p1, altrimenti se è inferiore viene eseguito il gossip con probabilità p2 > p1.
Uno schema non deterministico più complesso invece utilizza l'insieme N_i dei vicini di un generico nodo i per ottimizzare il flooding. Il nodo i quando esegue un broadcast a livello locale copre un'area A_i; tutti i nodi situati all'interno di questa area riceveranno il broadcast e decideranno se ritrasmetterlo o meno. Più un nodo si trova nelle vicinanze di chi ha originato il broadcast e minore è la possibilità che eventuali ritrasmissioni possano raggiungere dei nodi al di fuori dell'area A_i. Di conseguenza la probabilità di trasmissione dovrebbe dipendere sia dal numero di vicini sia dalla distanza dalla sorgente e destinatario.
Per determinare la distanza tra i nodi, quando non sia disponibile una misura dell'intensità del segnale, è possibile utilizzare una metrica equivalente, basata sulle differenze tra gli insieme dei vicini dei due nodi interessati. In questo schema la distanza μ_{ij} tra il trasmettitore *i* e il ricevitore *j* valutata dal nodo *j*, dati gli insiemi dei vicini N_i e N_j, è data ad esempio da :

$$\mu_{i,j} = \frac{card(N_i)}{card(N_j) + card(N_i \bigcap N_j)}$$

Si noti che μ_{ij} decresce se aumenta la sovrapposizione tra N_i e N_j e quindi μ è una metrica equivalente alla distanza tra i due nodi. Data la distanza μ_{ij} è possibile definire quindi una funzione empirica che determini il valore della probabilità di ritrasmissione.
Determinare però il numero di vicini istantanei di un nodo oppure ottenere informazioni sulla località non è un problema semplice; la soluzione più ovvia consiste nella ritrasmissione periodica di messaggi di HELLO da parte di ogni nodo. Nelle MANET però il periodo non può essere fisso ma deve dipendere dalla mobilità della rete stessa poiché altrimenti in condizioni di alta mobilità si avrebbero informazioni che diventano rapidamente obsolete creando cosi uno squilibrio e il conseguente decadimento delle prestazioni del protocollo di broadcast.

Se sono disponibili informazioni sulla densità locale allora allora è possibile che il sistema renda disponibili ulteriori informazioni sui nodi vicini. A questo punto ci si potrebbe chiedere se non sia migliore l'uso di un protocollo deterministico, poiché questi si avvalgono della raccolta di informazioni attraverso l'uso metodi proattivi e riescono sicuramente a farne un uso migliore. Tuttavia mentre i protocolli deterministici hanno bisogno di essere aggiornati sull'andamento del processo di flooding, queste informazioni non sono necessarie ai protocolli non deterministici, evitando cosi fastidiosi ritardi di trasmissione.

4.6 Gossip a zone

Se la rete è partizionata in cluster è possibile utilizzare uno schema di gossip che ne faccia uso. Le operazioni e le informazioni sulle zone possono ad esempio provenire sia dall'utilizzo di appositi protocolli di "cluster formation", che essere messe a disposizione da protocolli di routing basati sul concetto di zona.
Nel gossip a zone il nodo che inizia il broadcast trasmette prima il messaggio ai nodi della zona cui appartiene. La diffusione del broadcast all'interno di una zona viene eseguita tramite un protocollo deterministico che utilizza tutte le informazioni topologiche localmente disponibili. Quando il broadcast arriva ad un nodo periferico, questo decide con probabilità p se ritrasmetterlo o meno.
I nodi periferici eseguono quindi il broadcast in modo non deterministico. Il gossip non opera a livello di nodo ma a livello di zona.
La raggiungibilità è tanto migliore quanto maggiore è il diametro della zona considerata, anche se una dimensione eccessiva può portare un eccessivo overhead dovuto al mantenimento proattivo della topologia di ogni zona. Tale schema potrebbe avere in teoria una buona efficienza, soprattutto se non venissero eseguite ritrasmissioni verso zone che hanno già ricevuto il messaggio.

4.7 GOSSIP3

Uno schema di gossip che non richieda la conoscenza locale dei vicini è stato proposto da Zigmunt J. Haas e altri ricercatori; tale schema è detto GOSSIP3(p, k, m). Come è stato precedentemente visto il gossip non è adattabile alla topologia delle MANET, e le alternative proposte richiedono dei meccanismi proattivi che permettano di conoscere lo stato della MANET intorno al nodo che esegue il gossip. Questi possono richiedere l'accesso a funzioni di basso livello, rendendo quindi un sistema poco portabile o troppo complesso.

Pur non potendo avere accesso ad informazioni sulla topologia locale è però possibile avere indizi sull'andamento del processo di flooding, ad esempio controllando il numero di broadcast locali ricevuti per un certo messaggio. Il funzionamento di GOSSIP3(p, k, m) si basa proprio su tale principio. Il broadcast di un messaggio m_{id} procede esattamente come in GOSSIP1(p,k), ma se la ritrasmissione non viene eseguita, il messaggio m_{id} non viene immediatamente scartato.
Viene posto in una coda di attesa per un periodo predeterminato τ. Durante l'attesa il nodo registra il numero delle ritrasmissioni broadcast dello stesso messaggio m_{id}. Al termine dell'attesa il nodo avrà ricevuto n ritrasmissioni del messaggio m_{id}; se $n < m$ il nodo ritrasmette il messaggio, altrimenti lo scarta.
La soglia m determina l'esito della ritrasmissione allo scadere del tempo τ e dovrebbe dipendere dal valore atteso dell'area addizionale di copertura o EAC, ovvero l'area ulteriore che il nodo si aspetta di risucire a coprire ritrasmettendo il messaggio dopo aver ricevuto *m* broadcast. Se il numero dei broadcast ricevuti è inferiore a *m*, allora l'area attesa sarà superiore ad EAC(m) e quindi verrà eseguita la ritrasmissione. Il periodo di attesa τ determina la dimensione della "finestra di osservazione" e potrebbe essere regolato dinamicamente, ad esempio utilizzando una ulteriore soglia di valore maggiore di m, come avviene nella versione adattiva del broadcast counter-based.
La bassa latenza, dovuta alla propagazione attraverso la prima fase non deterministica rende GOSSIP3 un buon candidato per l'utilizzo in protocolli di routing reattivi come AODV.

4.8 AGAR: Adaptive Gossip-based Routing Algorithm

Gli autori di AGAR hanno analizzato le prestazioni di GOSSIP3 e lo hanno perfezionato sintetizzando un nuovo protocollo di broadcast per le MANET. AGAR evita la discontinuità iniziale dell'andamento della probabilità totale di ritrasmissione di GOSSIP3, rendendo non deterministico anche il secondo stadio di ritrasmissione, evitando uno schema a soglia counter-based.
Utilizzano AGAR con probabilità p un pacchetto broadcast viene ritrasmesso se, estratto un valore casuale, questo risulta minore di p (stadio di gossip iniziale) oppure se, trascorso il periodo di attesa in cui vengono ricevute n ritrasmissioni dello stesso messaggio (secondo stadio) e estratto un ulteriore valore casuale, questo risulti minore di p/(n+1). L'andamento della probabilità totale di ritrasmissione P_A ha quindi l'andamento della figura 4.2.

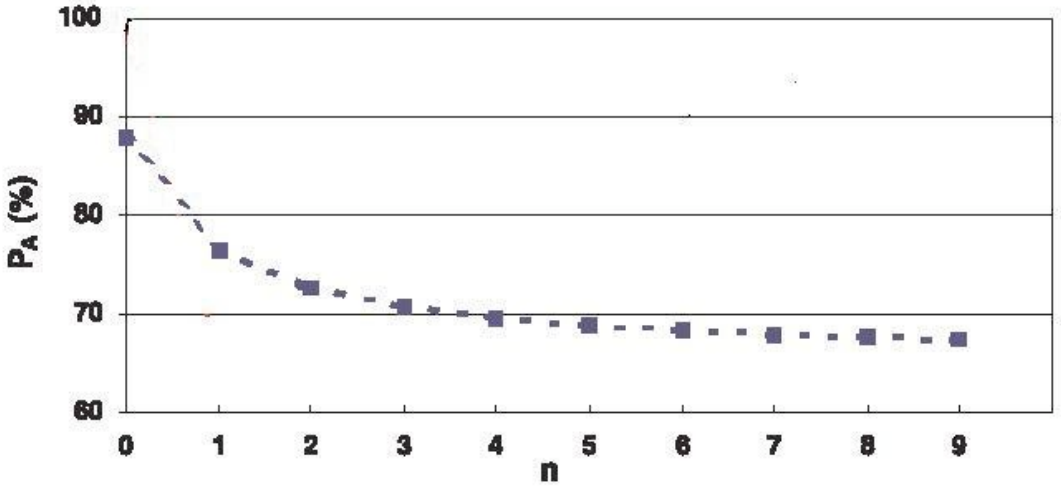

Figura 4.2 Probabilità complessiva di ritrasmissione in AGAR

4.9 Linee Guida

La ricerca di soluzioni di broadcast efficaci ed efficienti è come visto nei paragrafi precedenti uno dei temi più studiati negli ultimi anni dalla ricerca nel settore delle MANET. Dall'analisi comparata di queste soluzioni è emerso che i protocolli non deterministici come per esempio il GOSSIP1 hanno un'efficienza minore rispetto ai protocolli deterministici, tuttavia questi non avendo la necessità di utilizzare un tempo di attesa prefissato RAD per la ritrasmissione dei messaggi, hanno latenze molto inferiori ai protocolli deterministici basati su RAD, e questo li rende ottimi candidati per la realizzazione di applicazioni interattive e protocolli di routing, soprattutto considerando schemi adattativi come GOSSIP3 e AGAR.

Cap. 5
Network Simulator 2

5.1 L'ambiente Di Simulazione

Ns2 è un simulatore di reti ad eventi discreti scritto interamente in C++, con l'ausilio di un interprete OTcl (Fig. 5.1). [Kev03] Ns2 usa due linguaggi di programmazione poiché il simulatore ha due necessità principali; infatti, da una parte dettagliate simulazioni di protocolli richiedono un linguaggio di programmazione che possa efficientemente manipolare bytes, headers dei pacchetti, ed implementare algoritmi che utilizzino un gran numero di dati. Per queste attività la velocità a tempo di esecuzione (run-time) è importante, mentre il tempo impiegato per altre attività come l'esecuzione della simulazione, trovare degli errori, correggerli, ricompilare e rieseguire di nuovo il tutto , è di gran lunga meno importante. Dall'altro lato invece gran parte dello studio delle reti si evolve variando parametri o cambiando alcune configurazioni, o esplorando rapidamente un certo numero di scenari. In questi casi il tempo di iterazione (cambiare il modello e rieseguire il tutto) è più importante. Una volta che la configurazione della nostra simulazione viene eseguita la prima volta (all'inizio della simulazione), il tempo di esecuzione di questa parte di attività è meno importante.
Per questi motivi Ns2 utilizza sia il C++ basato sulla gerarchia di classi cosiddetta "compilata" sia OTcl basato sulla gerarchia cosiddetta "interpretata". Il C++ è rapido da eseguire ma più lento da modificare, rendendolo adatto per dettagliate implementazioni di protocolli. OTcl invece è lento da eseguire ma può essere modificato molto rapidamente (interattivamente).
Le due gerarchie, compilata ed interpretata, sono strettamente correlate l'una all'altra; infatti dal punto di vista dell'utente, c'è una corrispondenza 1:1 tra una classe nella gerarchia interpretata e una classe in quella compilata. La radice di questa gerarchia è la classe TclObject (Fig. 5.2). L'utente può creare nuovi oggetti del simulatore attraverso l'interprete, all'interno del quale vengono istanziati e dal quale vengono associati immediatamente ad un oggetto corrispondente nella gerarchia compilata. La gerarchia di classi interpretata è automaticamente stabilita attraverso i metodi definiti nella classe TclClass, mentre gli oggetti istanziati dall'utente, vengono associati all'oggetto della gerarchia compilata attraverso i metodi definiti nella classe TclObject.

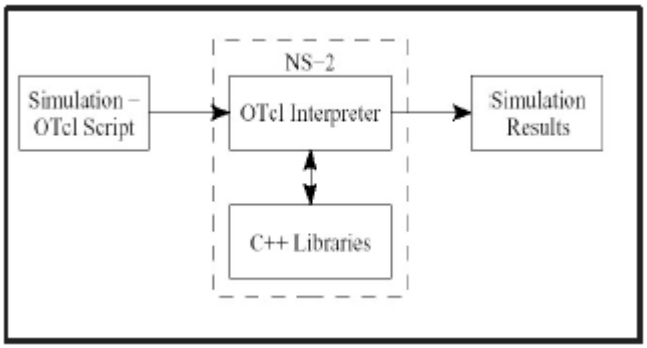

Figura 5.1 Architettura Di Ns2

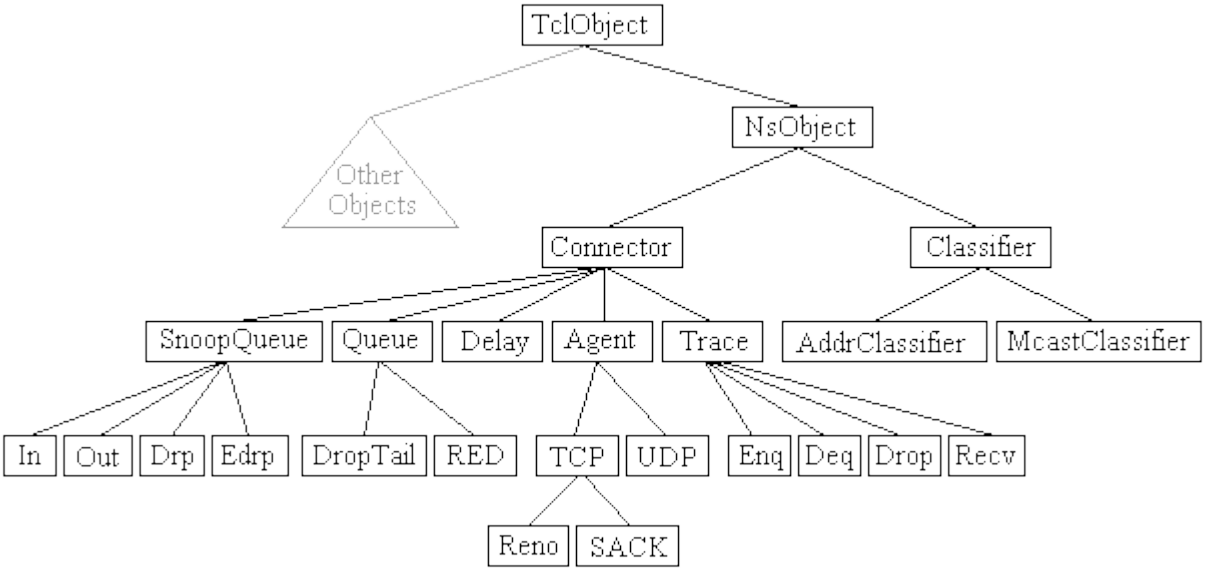

Figura 5.2 Gerarchia Delle Classi Ns2

5.2 Gerarchia delle Classi

Il codice che permette al simulatore Ns2 di interfacciarsi con l'interprete OTcl risiede in una directory separata ("tclcl") da quella del resto del codice del simulatore che invece risiede nella directory "ns-2". (Fig 5.3).
Ci sono numerose classi nella directory dell'interprete Otcl ma quelle su cui noi focalizzeremo la nostra attenzione sono principalmente sei:

- La classe Tcl: contiene i metodi che il C++ userà per accedere all'interprete.
- La classe TclObject: è la classe base per tutti gli oggetti del simulatore che hanno un corrispondente oggetto nella gerarchia compilata.
- La classe TclClass: definisce la gerarchia di classi interpretata ed i metodi che permettono all'utente di istanziare oggetti di tipo TclObject.

- La classe TclCommand: è usata per definire semplici comandi globali per l'interprete.
- La classe EmbeddedTcl: contiene i metodi per caricare comandi pre-assemblati ad alto livello che rendono la simulazione più semplice.
- La classe InstVar: contiene i metodi per accedere alle variabili membro del C++ per mezzo di variabili istanziate da Tcl.

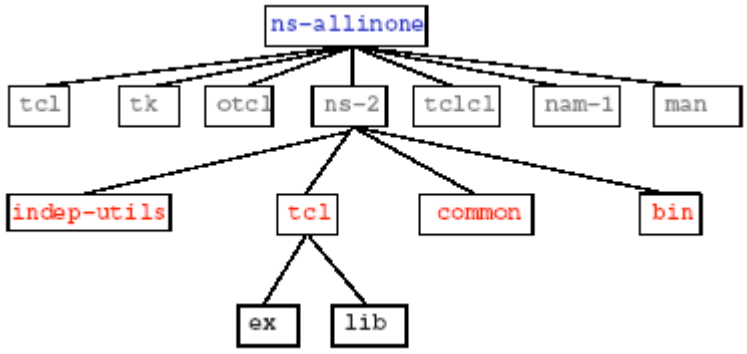

Figura. 5.3 Gerarchia Delle Cartelle

5.2.1 La Classe Tcl

La classe Tcl incapsula l'istanza attuale dell'interprete OTcl, e fornisce i metodi per accedere e comunicare con l'interprete. La classe fornisce i metodi per le seguenti operazioni :

- Ottenere un riferimento all'istanza Tcl.
- Invocare le procedure OTcl attraverso l'interprete.
- Recuperare o passare come parametri risultati all'interprete
- Notificare situazioni di errore ed uscire dalla simulazione in maniera sicura.
- Memorizzare e tenere traccia di tutti gli oggetti di tipo TclObject.

Acquisire un accesso diretto all'interprete.

5.2.1.1 Creare un'istanza della classe Tcl

Un'unica istanza della classe è dichiarata in *~tclcl*/Tcl.cc come una variabile membro statica; il programmatore deve ottenere un riferimento a questa istanza per accedere ai metodi di tale classe.
Il codice richiesto per accedere a tale istanza è :

```
Tcl& tcl = Tcl::instance();
```

5.2.1.2 Invocare Procedure OTcl

Per invocare un comando OTcl attraverso l'istanza tcl precedentemente creata sono disponibili quattro diverse funzioni che differiscono essenzialmente per il tipo di argomenti passati come parametri. Ogni

funzione passa una stringa all'interprete, che poi valuta la stringa in un contesto globale. Questi metodi ritorneranno il controllo al chiamante se l'interprete restituisce TCL_OK mentre se restituisce TCL_ERROR i metodi chiameranno `tkerror{}`. L'utente può sovraccaricare questa procedura per distinguere certi tipi di errori. I 4 tipi di procedure sono :

- `tcl.eval`(char* s) invoca `Tcl_GlobalEval()` da eseguire attraverso l'interprete.
- `tcl.eval(`const char * `s`) conserva il valore originale di s. Copia la stringa s in un suo buffer interno e poi invoca la precedente eval(char * s) sul buffer interno.
- `tcl.eval()` assume che il comando da eseguire si già memorizzato nel membro interno `bp_` ; invoca direttamente `tcl.eval(char* bp).`Un riferimento al buffer stesso è disponibile attraverso il metodo `tcl.buffer(void).`
- `tcl.evalf`(const char*) è un equivalente della `Printf`(3). Esso usa `vsprintf`(3) internamente per creare la stringa di input.

Come esempio ecco alcuni modi di utilizzare i metodi sopra citati :

```
Tcl& tcl = Tcl::instance();
char wrk[128];
strcpy(wrk, "Simulator set NumberInterfaces_ 1");
tcl.eval(wrk);

sprintf(tcl.buffer(), "Agent/SRM set requestFunction_ %s", "Fixed");
tcl.eval();

tcl.evalc("puts stdout hello world");
tcl.evalf("%s request %d %d", name_, sender, msgid);
```

5.2.1.3 Passare Risultati al /dal Interprete.

Quando l'interprete invoca un metodo C++, esso si aspetta di ricevere il risultato dell'invocazione nella variabile membro `tcl_->result`. Due metodi sono disponibili al fine di settare il valore di questa variabile:

- `tcl.result(const char* s)` Restituisce la stringa all'interprete.
- `tcl.resultf(const char* fmt,…)` E' una variante di `varargs(3)` del caso precedente per formattare il risultato usando `vsprintf(3)` , che ritorna la stringa all'interprete.

```
if (strcmp(argv[1], "now") == 0) {
        tcl.resultf("%.17g", clock());
        return TCL_OK;
}
tcl.result("Invalid operation specified");
return TCL_ERROR;
```

Analogamente quando un metodo C++ invoca un comando OTcl, l'interprete ritorna il risultato in `tcl_->result`.

- `tcl.result(void)` deve essere usata per recuperare il risultato. E' da notare come il formato del risultato sia una stringa che quindi va convertita nel formato interno più appropriato.

```
tcl.evalc("Simulator set NumberInterfaces_");
char* ni = tcl.result();
if (atoi(ni) != 1)
        tcl.evalc("Simulator set NumberInterfaces_ 1");
```

5.2.1.4 Notifica Degli Errori e termine della Simulazione

Questo metodo fornisce un modo uniforme per notificare gli errori nel codice :

- `tcl.error(const char* s)` esegue le seguenti operazioni: scrive s sullo `stdout`; scrive `tcl_->result` sullo `stdout` ed esce con codice di errore 1.

```
tcl.resultf("cmd = %s", cmd);
tcl.error("invalid command specified");
/*NOTREACHED*/
```

5.2.2 La Classe TclObject

La classe TclObject è la classe base per la maggior parte delle altre classi sia nella gerarchia interpretata che in quella compilata. Ogni oggetto nella classe TclObject è creato dall'utente all'interno dell'interprete e a questa creazione corrisponde sempre quella della copia equivalente nella gerarchia compilata. I due oggetti sono strettamente legati l'uno all'altro. Un esempio di configurazione di un TclObject è il seguente dove viene illustrata la gestione di un SRM agent (classe Agent/SRM/Adaptive).

```
set srm [new Agent/SRM/Adaptive]
$srm set packetSize_ 1024
$srm traffic-source $s0
```

La prima linea dell'esempio precedente mostra come un oggetto di tipo TclObject viene creato; la linea successiva configura una variabile membro; finalmente l'ultima linea illustra l'oggetto nella gerarchia interpretata che invoca un metodo C++ in quanto era una procedura di tale istanza.

5.2.2.1 Creare istanze di oggetti TclObject

Usando l'operatore `new{},` l'utente può creare un TclObject interpretato. L'interprete eseguirà il costruttore per quell'oggetto, `init{}`, passandogli qualsiasi argomento fornito dall'utente. Ns2 è responsabile della creazione automatica dell'oggetto compilato. Questo oggetto viene creato dal costruttore della classe base TclObject. Quindi, il costruttore per il nuovo TclObject deve prima chiamare il costruttore della classe padre. L'operatore `new {}` ritorna un riferimento all'oggetto, che quindi può essere usato per ulteriori operazioni su quell'oggetto. Un esempio di costruttore può essere il seguente:

```
Agent/SRM/Adaptive instproc init args {
        eval $self next $args
        $self array set closest_ "requestor 0 repairor 0"
        $self set eps_    [$class set eps_]
}
```

5.2.2.2 Eliminare un'istanza di TclObject

L'operazione `delete` distrugge l'oggetto interpretato, e il suo corrispondente nella gerarchia compilata. Per esempio, use-scheduler {(scheduler)} usa la procedura di `delete` per rimuovere la lista di default dello scheduler, ed instanziare uno scheduler alternativo al suo posto.

```
Simulator instproc use-scheduler type {
        $self instvar scheduler_

        delete scheduler_                          ;# first delete the existing list scheduler
        set scheduler_ [new Scheduler/$type]
}
```

5.2.2.3 Binding di variabili

In molti casi l'accesso, l'accesso alle variabili membro nella gerarchia compilata (C++) è ristretto solamente a quest'ultima gerarchia ed ugualmente l'accesso alle variabili membro nella gerarchia interpretata (Tcl) è limitato localmente ad essa; comunque, è possibile stabilire un collegamento bidirezionale in modo tale che le variabili membro di

entrambe le gerarchie accedano agli stessi dati e cambiando il valore di una delle due variabili cambi anche il valore dell'altra variabile corrispondente.
La relazione (binding) tra le due variabili è stabilita dal costruttore (C++) quando un oggetto viene istanziato; esso è automaticamente accessibile dal corrispondente oggetto interpretato per mezzo di una variabile di tale istanza.
Il seguente esempio mostra il costruttore per l'agente ASRM:

```
ASRMAgent::ASRMAgent() {
        bind("pdistance_", &pdistance_);                    /* real variable */
        bind("requestor_", &requestor_);                    /* integer variable */
        bind_time("lastSent_", &lastSessSent_);             /* time variable */
        bind_bw("ctrlLimit_", &ctrlBWLimit_);               /* bandwidth variable */
        bind_bool("running_", &running_);                   /* boolean variable */
}
```

E' da notare che a tutte le funzioni sopra elencate vengono passati due argomenti, il nome di una variabile OTcl, e il riferimento della corrispondente variabile membro dell'oggetto compilato con cui viene effettuato il binding. Ognuna della variabili con cui viene effettuato il binding viene automaticamente inizializzata con il valore di default quando l'oggetto viene creato specificato dalla variabile della classe interpretata (OTcl). L'inizializzazione viene effettuata dalla procedura `init-instvar{}` della classe `Instvar`.
Per esempio, se le seguenti variabili di classe per l'agente ASRM vengono definite come :

```
Agent/SRM/Adaptive set pdistance_ 15.0
Agent/SRM set pdistance_ 10.0
Agent/SRM set lastSent_ 8.345m
Agent set ctrlLimit_    1.44M
Agent/SRM/Adaptive set running_ f
```

Allora, ogni nuovo oggetto Agent/SRM/Adaptive avrà la variabile `pdistance_` settata ad 15.0; la variabile `lastSent_` settata ad 8.345m derivato dall'inizializzazione della variabile di classe nella classe padre; la variabile `ctrlLimit_` settata ad 1.44M usando l'inizializzazione della classe progenitore; la variabile `running_` viene settata a "false";al contrario la variabile `pdistance_` in seconda riga non viene inizializzata poiché non esiste nessuna variabile di classe con tale nome in tutta la gerarchia interpretata dell'oggetto. In questo caso la procedura `init-instvar{}` invocherà a sua volta la procedura

`warn-instvar{}` che provvederà a stampare un avviso dell'errore commesso.

5.2.2.4 Metodi command() : Definizione ed Invocazione

Per ogni oggetto TclObject creato, Ns2 definisce la procedura cmd{} come punto di aggancio per eseguire metodi attraverso l'oggetto corrispondente nella gerarchia compilata. L'utente può invocare il metodo cmd{} in due modi : può invocare la procedure esplicitamente, specificando l'operazione desiderata come primo argomento, o implicitamente, nel caso ci fosse una procedura con lo stesso nome della operazione desiderata. La maggior parte degli script di simulazione usano questa ultima forma.
Consideriamo il fatto che per esempio il calcolo della distanza nell'agente SRM è fatto dall'oggetto compilato; comunque, è spesso anche effettuato anche dall'oggetto interpretato. L'invocazione è la seguente:

```
$srmObject distance? ⟨agentAddress⟩
```

Se non c'è nessuna procedura chiamata `distance`, l'interprete invocherà la procedura `unknown{}` definita nella classe base TclObject. La procedura `unknown` quindi invocherà

```
$srmObject cmd distance? ⟨agentAddress⟩
```

al fine di eseguire l'operazione attraverso la procedura command() dell'oggetto compilato.
L'utente però potrebbe invocare l'operazione direttamente. Una ragione potrebbe essere sovraccaricare l'operazione usando una procedura con lo stesso nome.

Un possibile esempio di tale eventualità potrebbe essere il seguente :

```
Agent/SRM/Adaptive instproc distance? addr {
        $self instvar distanceCache_
        if ![info exists distanceCache_($addr)] {
                set distanceCache_($addr) [$self cmd distance? $addr]
        }
        set distanceCache_($addr)
}
```

Ora passeremo ad illustrare invece l'utilizzo del metodo `command()` nell'agente ASRM :

```
int ASRMAgent::command(int argc, const char*const*argv) {
        Tcl& tcl = Tcl::instance();
        if (argc == 3) {
                if (strcmp(argv[1], "distance?") == 0) {
                        int sender = atoi(argv[2]);
                        SRMinfo* sp = get_state(sender);
                        tcl.tesultf("%f", sp->distance_);
                        return TCL_OK;
                }
        }
        return (SRMAgent::command(argc, argv));
}
```

Da questo spezzone di codice possiamo trarre le seguenti conclusioni:

- La funzione è chiamata con due argomenti; il primo argomento (argc) indica il numero di argomenti specificato da linea di comando all'interprete. Il vettore(argv) passato come argomento continene :
  1. `argv[0]` contiene il nome del metodo, "cmd".
  2. `argv[1]` specifica l'operazione desiderata.
  3. Se l'utente specifica ulteriori argomenti, essi vengono inseriti in `argv[2 … (argc – 1)]`.

  Gli argomenti passati sono stringhe, perciò devono essere convertiti nel formato appropriato.
- Se l'operazione desiderata esiste, dovrebbe ritornarne il risultato, usando i metodi descritti precedentemente.
- Il metodo `command()` stesso deve ritornare `TCL_OK` o `TCL_ERROR` per indicare il successo o il fallimento dell'operazione.

```
.....
if (strcmp(argv[1], "slot") == 0)
{
 int slot = atoi(argv[2]);
 if ((slot \>= 0) || (slot \< nslot_))
 {
  tcl.resultf("Operazione riuscita");
  return TCL_OK;
 }
 else
 {
  tcl.resultf("Operazione fallita");
  return (TCL_ERROR);
 }
}
```

- Se l'operazione desiderata non esiste in questo metodo, deve invocare il metodo command della classe padre, e ritornare il risultato corrispondente.

5.2.3 La Classe TclClass

La classe compilata `TclClass` è una classe virtuale. Le classi derivate da essa forniscono due funzioni principali: costruire la gerarchia di classi interpretata in relazione alla gerarchia compilata e forniscono metodi per istanziare nuovo oggetti di tipo `TclObject`. Ogni classe derivata è associata con una particolare classe compilata nella gerarchia di classi compilata e può istanziare nuovi oggetti nella classe associata.
Come esempio consideriamo la classe `RenoTcpClass`. E' derivata dalla classe `TclClass`, ed è associata con la classe `RenoTcpAgent`. Essa istanzierà nuovi oggetti nella classe `RenoTcpAgent`. La gerarchia di classi compilata corrispondente per la classe `RenoTcpAgent` è quella che deriva da `TcpAgent`, che a sua volta deriva da `Agent`, che a sua volta deriva da `TclObject`. `RenoTcpClass` è definita come:

```
static class RenoTcpClass: public TclClass {
public:
        RenoTcpClass() : TclClass("Agent/TCP/Reno") {}
        TclObject* create(int argc, const char*const* argv) {
                return (new RenoTcpAgent());
        }
} class_reno;
```

Possiamo quindi fare le seguenti osservazioni :

1. La classe definisce solo il costruttore, e un metodo addizionale, per creare istanze del `TclObject` associato.
2. Ns2 eseguirà il costruttore della classe `RenoTcpClass` per la variabile statica `class_reno` quando esegue per la prima volta. Questo provvederà a costituire i metodi appropriati e la corrispondente gerarchia di classi interpretata.
3. Il costruttore specifica la classe interpretata esplicitamente come `Agent/TCP/Reno`. Questo inoltre specifica implicitamente anche la gerarchia di classi interpretata.
4. Questa classe è associata con la classe `RenoTcpAgent`; essa crea nuovi oggetti nella classe associata.
5. Il metodo `RenoTcpClass::create` ritorna oggetti di tipo `TclObject` alla classe `RenoTcpAgent`
6. Quando l'utente specifica new `Agent/TCP/Reno`, viene invocata la routine `RenoTcpClass::create`.
7. Il vettore argv contiene :
    - `argv[0]` contiene il nome dell'oggetto.
    - `argv[1…3]` contiene `$self`, `$class`, e `$proc`. Dal momento in cui la routine `create` viene chiamata

attraverso la procedura `create-shadow`, `argv[3]` contiene `create-shadow`.

- `argv[4]` contiene qualsiasi parametro aggiuntivo fornito dall'utente.

5.2.3.1 Binding di variabili di classe statiche C++

Abbiamo visto precedentemente come effettuare il binding di variabili membro di un oggetto C++ nello namespace Otcl. Questo comunque non è applicabile a membri statici di una classe C++. E' evidente come non si possa risolvere il problema usando una soluzione simile al binding nel `TclObject`, il quale è basato su `InstVar`, poiché gli oggetti di tipo `InstVar` richiedono in TclCl la presenza di un `TclObject`. Comunque, possiamo creare un metodo della corrispondente TclClass e accedere ai membri statici di una classe C++ attraverso i metodi della `TclClass`. La procedura è la seguente :

1. Creare la propria `TclClass` derivata come precedentemente descritto;
2. Dichiarare i metodi `bind()` e `method()` nella propria classe derivata;
3. Implementare i propri metodi per effettuare il binding all'interno del metodo bind() con la sintassi `add_method("metodo")`, ed implementare il gestore nel metodo method() in maniera simile a come faremmo nel metodo `TclObject::command()`. E' importante notare come il numero di argomenti passati al metodo `TclClass::method()` è differente dal numero passato al metodo `TclObject::command().`

Come esempio, mostriamo una versione semplificata della classe `PacketHeaderClass` (nella cartella ~ns/packet.cc). Supponiamo di avere la seguente classe Racket che ha una variabile statica `hdrlen_` a cui vogliamo accedere da OTcl :

```
class Packet {
        ......
        static int hdrlen_;
};
```

Poi scriviamo il codice seguente per l'accesso a tale variabile :

```
class PacketHeaderClass : public TclClass {
protected:
        PacketHeaderClass(const char* classname, int hdrsize);
        TclObject* create(int argc, const char*const* argv);
                                                /* These two implements OTcl class access methods */
        virtual void bind();
        virtual int method(int argc, const char*const* argv);
};

void PacketHeaderClass::bind()
{
                                                /* Call to base class bind() must precede add_method() */
        TclClass::bind();
        add_method("hdrlen");
}

int PacketHeaderClass::method(int ac, const char*const* av)
{
        Tcl& tcl = Tcl::instance();
                /* Notice this argument translation; we can then handle them  as if in TclObject::command() */
        int argc = ac - 2;
        const char*const* argv = av + 2;
        if (argc == 2) {
                if (strcmp(argv[1], "hdrlen") == 0) {
                        tcl.resultf("%d", Packet::hdrlen_);
                        return (TCL_OK);
                }
        } else if (argc == 3) {
                if (strcmp(argv[1], "hdrlen") == 0) {
                        Packet::hdrlen_ = atoi(argv[2]);
                        return (TCL_OK);
                }
        }
        return TclClass::method(ac, av);
}
```

Ora possiamo usare i seguenti comandi OTcl per accedere e cambiare il valore della variabile `Packet::hdrlen_` :

```
PacketHeader hdrlen 120
set i [PacketHeader hdrlen]
```

5.2.4 La Classe TclCommand

Questa classe fornisce il meccanismo ad Ns2 per esportare semplici comandi all'interprete, che possano essere eseguiti all'interno di un contesto globale dall'interprete stesso.
Ora descriveremo come definire un nuovo comando usando la classe di esempio say_hello. L'esempio definisce il comando hi al fine di stampare la stringa "hello world". Seguita da qualsiasi argomento a linea di comando specificato dall'utente. Per esempio :

```
% hi this is ns [ns-version]
hello world, this is ns 2.0a12
```

1. Il comando deve essere prima definito all'interno della classe, derivata dalla classe `TclCommand`. La definizione della classe è :

```
class say_hello : public TclCommand {
public:
        say_hello();
        int command(int argc, const char*const* argv);
};
```

2. Il costruttore della classe deve invocare il costruttore della classe `TclCommand` con il comando:

```
say_hello() : TclCommand("hi") {}
```

   Il costruttore della classe `TclCommand` setta "hi" come una procedura globale che invoca il comando `TclCommand::dispatch_cmd()`.

3. Il metodo `command()` deve effettuare l'azione desiderata. Al metodo vengono passati due argomenti. Il primo, `argc`, contiene il numero di parametri attuali passati dall'utente. Il secondo è il vettore `argv` che contiene :
   - `argv[0]` contiene il nome del comando ("hi")
   - `argv[1 …(argc – 1)]` contiene i parametri addizionali specificati dall'utente.

   Il comando `command()` viene invocato dal metodo `dispatch_cmd()`.

```
#include <streams.h>                          /* because we are using stream I/O */

int say_hello::command(int argc, const char*const* argv) {
        cout << "hello world:";
        for (int i = 1; i < argc; i++)
                cout << ' ' << argv[i];
        cout << '\ n';
        return TCL_OK;
}
```

4. Finalmente otteniamo un'istanza di questa classe. Le istanze della classe `TclCommand` sono create nella routine `inti_misc(void).`

```
new say_hello;
```

5.2.5 La Classe InstVar

Questa classe definisce i metodi e i meccanismi per mettere in relazione una variabile membro C++ con una variabile di un'istanza Otcl nell'equivalente oggetto interpretato. Il "binding" è attivo dal momento

in cui il valore della variabile può essere settato o recuperato o dall'interprete o dal codice compilato in qualsiasi momento.
C sono cinque istanze di sottoclassi di tale classe: `InstVarReal`, `InstVarTime`, `InstVarBandwidth`, `InstVarInt`, e la classe `InstVarBool`, corrispondenti al binding per variabili di tipo reale, tempo, banda, intere, e boleane.
Al fine di garantire la corretta esecuzione dei metodi, deve essere effettuato il binding di una variabile solo se la sua classe esiste già all'interno dell'interprete, e l'interprete sta correntemente operando su di un oggetto in quella classe.

5.3 Creare una nuova Simulazione

In questa sezione provvederemo a fornire i principali strumenti al fine di consentire all'utente una conoscenza adeguata per poter creare una nuova simulazione. [Mar] Nel prossimo paragrafo verrà illustrato come sviluppare uno script Tcl per Ns2 che simuli una semplice tipologia di rete. Verrà mostrato come settare nodi e collegamenti, come spedire dati da un nodo all'altro, come monitorare una coda e come lanciare "Nam" dal nostro script Tcl per visualizzare il risultato della simulazione.

5.3.1 Come iniziare

La prima cosa da fare nello scrivere un file di script Tcl è creare un nuovo oggetto simulatore con il comando :

```
set ns [new Simulator]
```

Esso genera una istanza di ns ed assegna questa istanza alla variabile *ns*. Inoltre inizializza il formato dei pacchetti, crea uno scheduler degli eventi e seleziona il formato degli indirizzi da utilizzare nella simulazione. Un oggetto simulatore, una volta creato consente di effettuare numerose operazioni come creare oggetti composti come link e nodi, connettere oggetti composti (es. agganciare gli agenti ai nodi), creare connessioni tra agenti (es. connessioni tra agenti TCP e agenti Sink), oppure specificare delle opzioni per l'ambiente grafico Nam.
Ora apriremo un file in scrittura che verrà usato per scrivere il trace dei dati da dare in pasto al "Nam" :

```
set nf [open out.nam w]
$ns namtrace-all $nf
```

La prima di queste due istruzioni imposta il file out.nam come file di output della simulazione aprendolo in scrittura, mentre nella seconda

viene detto all'oggetto simulatore che in questo file va scritta la traccia della simulazione in formato tale che Nam possa poi interpretarla.
Il prossimo passo è definire una procedura che andrà in esecuzione al termine della simulazione e che provvederà a chiudere i trace-file della simulazione:

```
proc finish {} {
 global ns nf
 $ns flush-trace
 close $nf
 exec nam out.nam &
 exit()
}
```

Questa funzione contiene tutti quei processi da eseguire al termine della simulazione. In questo caso infatti prima viene chiuso il trace-file e quindi viene invocato Nam a cui viene passato come parametro proprio il file contenente i dati ottenuti dalla simulazione.

La riga di codice successiva dirà al simulatore di eseguire la procedura `finish` dopo 5 secondi :

```
$ns at 5.0 "finish"
```

L'ultima riga infine fa partire la simulazione :

```
$ns run
```

5.3.2 Nodi e Links

In questa sezione definiremo una semplice topologia con due nodi connessi da un link. Per creare due nuovi nodi basterà scrivere le due seguenti linee di codice (Va Inserito prima della linea `"$ns at 5.0 finish"`) :

```
set n0 [$ns node]
set n1 [$ns node]
```

In Otcl il Nodo è una classe particolare anche se la maggior parte dei suoi componenti sono comunque TclObjects. La tipica struttura di un nodo (unicast) è mostrata in figura 5.4 . Questa semplice struttura consiste di due TclObjects : un classificatore di indirizzi(`classifier_`) e un classificatore di porte (`dmux_`).
La funzione corretta di questi due oggetti è distribuire i pacchetti in arrivo al corretto agente o verso il link in uscita appropriato.

Tutti i nodi contengono almeno i seguenti componenti :

- Un indirizzo o `id_` incrementato di volta in volta di un' unità ogni volta che un nuovo nodo viene creato.
- Una lista di vicini (`neighbor_`)
- Una lista di Agenti (`agent_`)
- Un identificativo del tipo di nodo (`nodetype_`)
- Un modulo di routing per l'instradamento dei pacchetti.

La linea successiva connette invece i due nodi precedentemente creati :

```
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
```

Questo comando non fa altro che creare un link bidirezionale tra i due nodi, specificando la larghezza di banda (es. 1Mb), il ritardo di propagazione (es. 10 ms), e la politica di gestione della coda di output (`Drop Tail`) che in Ns2 viene implementata come parte integrante di un link.

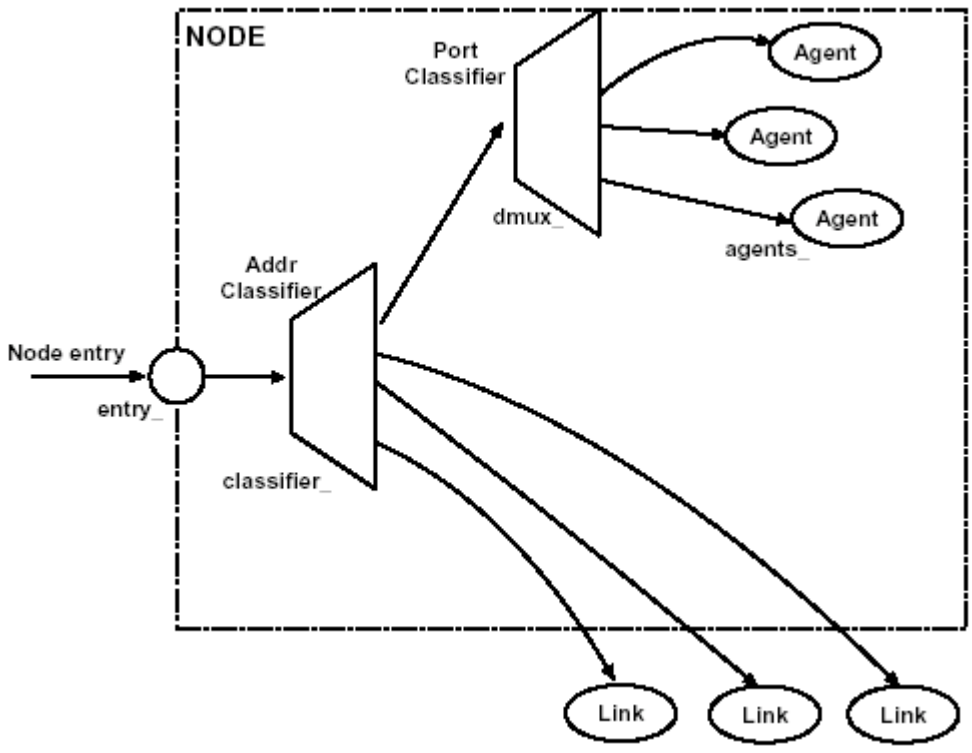

Figura 5.4 Struttura Di Un Nodo (unicast)

A questo punto potremmo lanciare il nostro primo script con il comando `ns esempio1.tcl` riassemblando tutti i comandi sopra descritti come segue :

```
#Open the nam trace file
set nf [open out.nam w]
```

```
$ns namtrace-all $nf

#Define a 'finish' procedure
proc finish {} {
       global ns nf
       $ns flush-trace
        #Close the trace file
       close $nf
        #Execute nam on the trace file
       exec nam out.nam &
       exit 0
}

#Create two nodes
set n0 [$ns node]
set n1 [$ns node]


#Create a duplex link between the nodes
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

#Call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"

#Run the simulation
$ns run
```

A questo punto una volta lanciato il nostro script "Nam" partirà automaticamente e dovremmo vedere a video una schermata simile quella in figura 5.5.
Nam (Network Animator Module) non è altro che un programma ad interfaccia grafica che permette di ricostruire una simulazione attraverso una animazione. In questo modo ha una maggiore comprensione del funzionamento della rete e dell'interazione tra i vari componenti; inoltre nel corso dell'animazione è possibile realizzare dei fermoimmagine, aumentare o diminuire la velocità di riproduzione e, "cliccando" sui vari elementi grafici che compongono la simulazione, ottenere informazioni specifiche (ad esempio, cliccando su un collegamento si ottiene il numero dei pacchetti persi su quel link).

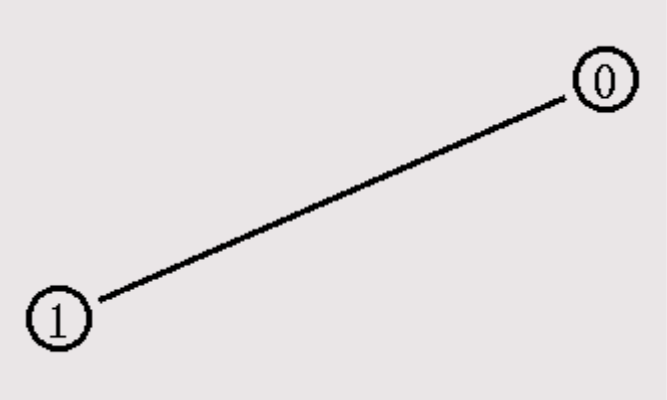

Figura 5.5 Snapshot Nam

5.3.3 Trasferimento di dati tra Nodi

In Ns2 i dati vengono sempre spediti da un Agente ad un altro. Il prossimo passo sarà quindi creare un oggetto Agent che spedisce dati dal nodo n0, e un altro oggetto Agent che riceve i dati sul nodo n1.

```
#Crea un agente UDP e lo attacca al nodo n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
# Crea una sorgente di traffico cbr e la collega
ad   udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

Nell'ultima riga il generatore di traffico cbr (constant bit rate) viene collegato all'agente udp0. Nelle due linee precedenti invece viene settata la dimensione del pacchetto a 500 bytes e settato l'intervallo di invio a 5 ms (200 pacchetti per secondo). Le linee successive creano un agente "Sink" che riceve i pacchetti e lo collegano al nodo1:

```
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
```

Ora i due agenti devono essere connessi l'uno all'altro:

```
$ns connect $udp0 $null0
```

Ed ora dobbiamo dire all'agente CBR quando iniziare a spedire dati e quando fermarsi:

```
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

Modificando l'esempio precedente con queste nuove linee di codice e facendo ripartire la simulazione lanciando di nuovo il nostro script Tcl, Nam dovrebbe visualizzare uno scenario simile a quello di figura 5.6.

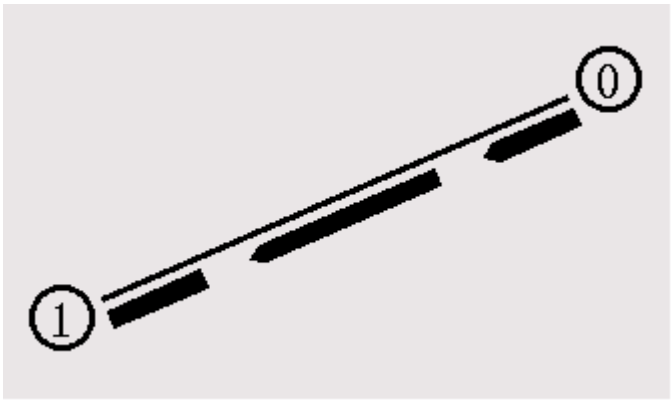

Figura 5.6 Nam: flusso di dati da nodo 0 a nodo 1

5.3.4 Flussi di Dati Multipli e Monitoraggio Di Code

Modificando l'esempio precedente con l'aggiunta di altri due nodi e cambiandone l'orientazione attraverso i seguenti comandi

```
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right
```

dovremmo ottenere uno scenario simile a quello in figura 5.7. Creeremo due agenti UDP con generatori di traffico CBR e li collegheremo ai nodi 0 ed 1; poi creeremo un agente Sink e lo collegheremo al nodo 3.

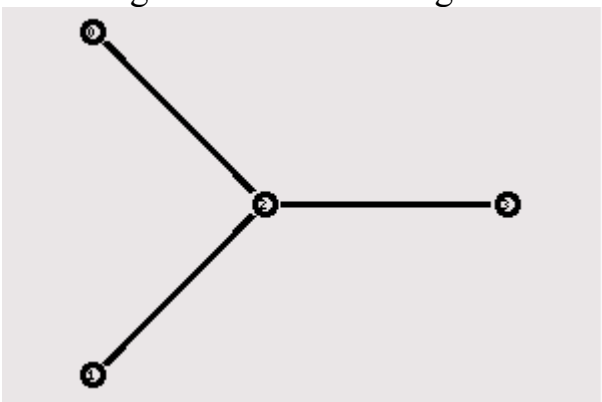

Figura 5.7 Nuovo Scenario

Aggiungendo le seguenti due linee alla definizione dei nostri agenti CBR

```
$udp0 set class_ 1
$udp1 set class_ 2
```

ed inserendo due nuove righe di codice all'inizio del nostro script Tcl potremo distinguere in Nam i flussi di pacchetti generati dai due agenti CBR poiché caratterizzati da due colori distinti:

```
$ns color 1 Blue
$ns color 2 Red
```

Quello che vedremo in Nam una volta lanciato il nostro script sarà simile alla figura 5.8 con appunto ben visibili i 2 flussi di pacchetti spediti dal nodo 0 e dal nodo 1 nel loro percorso verso il nodo 3.

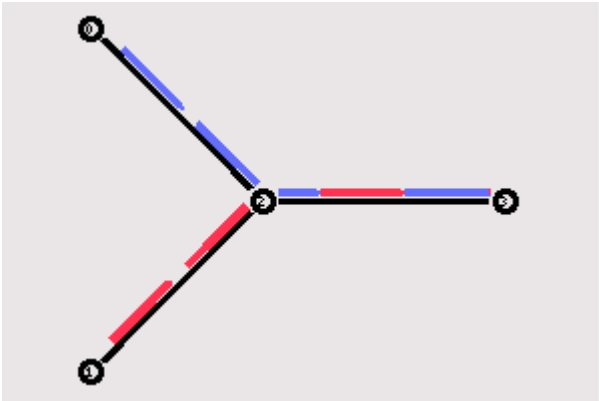

Figura 5.8 Flusso di dati in Nam

Supponiamo adesso di voler monitorare la coda di pacchetti in uscita dal nodo 2 in direzione del nodo 3. Basterà semplicemente aggiungere una semplice linea di codice nella definizione del link dal nodo 2 al nodo 3 :

```
$ns duplex-link-op $n2 $n3 queuePos 0.5
```

Facendo partire ancora il nostro script modificato la visualizzazione che avremo in Nam sarà simile a quella di Figura 5.9. Durante l'esecuzione potremo ora vedere i pacchetti all'interno della coda e successivamente notare come solamente i pacchetti indicati con il flusso di colore blu vengano scartati. Questo dipende però dal tipo di coda di uscita che abbiamo impostato al nodo; infatti cambiando il tipo di coda con una di tipo SFQ dovrebbero venire scartati lo stesso numero di pacchetti blu e rossi come è possibile notare in Figura 5.10.

```
$ns duplex-link $n3 $n2 1Mb 10ms SFQ
```

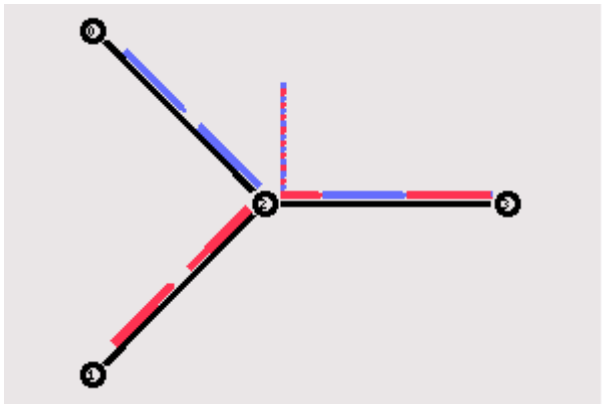

Figura 5.9 Monitoraggio della Coda

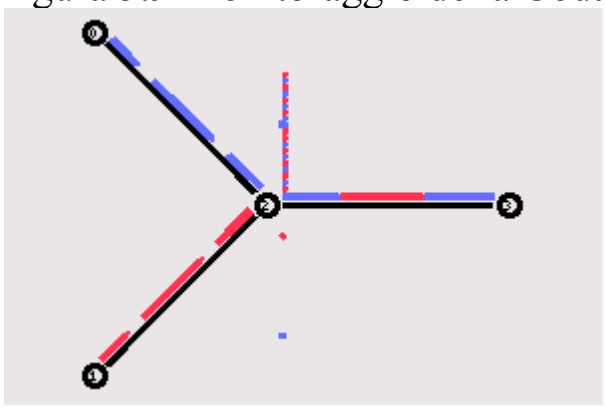

Figura 5.10 Monitoraggio di una coda SFQ

5.3.5 Altre tipologie di Scenari

Ora passeremo ad analizzare una topologia un po' più grande della precedente con sette nodi collegati tra loro in un anello come in Figura 5.11. Il seguente codice non fa altro che creare i sette nodi e memorizzarli in un vettore:

```
for {set i 0} {$i < 7} {incr i} {
  set n($i) [$ns node]
}
```

Provvederemo ora a collegare i nodi in una tipologia circolare utilizzando code di tipo DropQueue.

```
for {set i 0} {$i < 7} {incr i} {
  $ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms DropTail
}
```

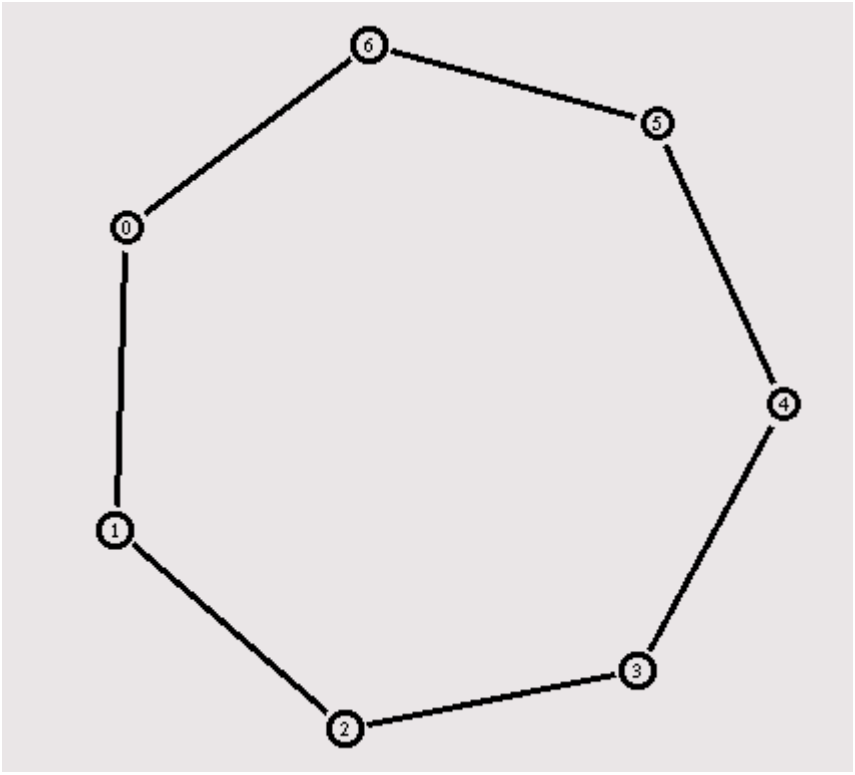

Figura 5.11 Tipologia ad Anello

5.3.6 Creare file di output per Xgraph

Una delle parti del package di ns-allinone è 'Xgraph', un programma di disegno che può essere usato per creare rappresentazioni grafiche dei risultati della simulazione. In questa sezione verrà mostrato come creare degli output files nei nostri script Tcl, i quali possono essere usati come data-sets per Xgraph. Verrà spiegato inoltre come usare i generatori di traffico. Come esempio scegliamo uno scenario con cinque nodi con tre sorgenti di traffico ed un nodo sink. Andremo poi a rappresentare graficamente per mezzo di Xgraph il risultato della nostra simulazione.

5.3.6.1 Topologia e Sorgenti di Traffico

Per prima cosa creiamo la seguente tipologia :

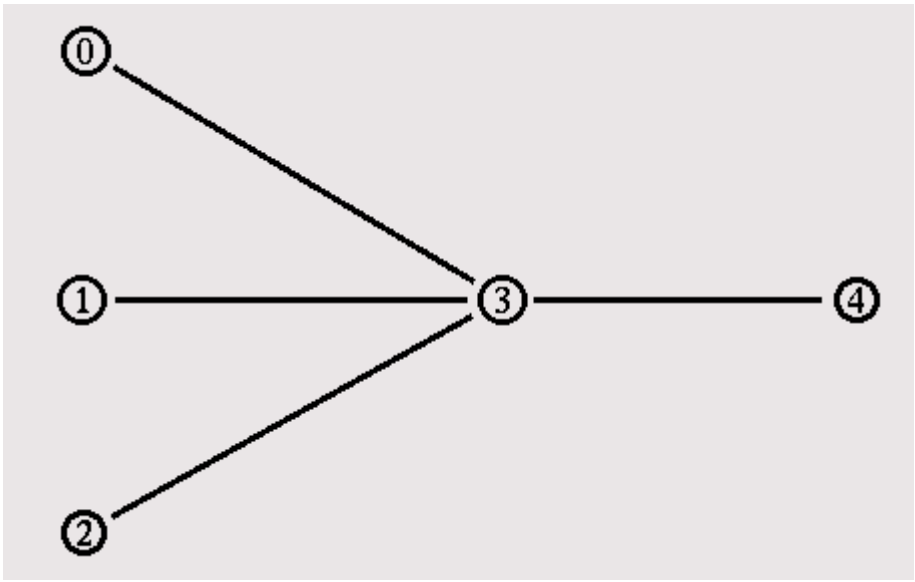

Figura 5.12 Tipologia Di rete

Il codice è il seguente:

```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]

$ns duplex-link $n0 $n3 1Mb 100ms DropTail
$ns duplex-link $n1 $n3 1Mb 100ms DropTail
$ns duplex-link $n2 $n3 1Mb 100ms DropTail
$ns duplex-link $n3 $n4 1Mb 100ms DropTail
```

Adesso non resta che collegare le sorgenti di traffico ai nodi n0, n1, n2, ma prima scriveremo una semplice procedura che ci semplificherà il collegamento di sorgenti di traffico e generatori ai nodi :

```
proc attach-expoo-traffic { node sink size burst idle rate } {
       #Get an instance of the simulator
       set ns [Simulator instance]

       #Create a UDP agent and attach it to the node
       set source [new Agent/UDP]
       $ns attach-agent $node $source

       #Create an Expoo traffic agent and set its
configuration parameters
       set traffic [new Application/Traffic/Exponential]
       $traffic set packet-size $size
       $traffic set burst-time $burst
       $traffic set idle-time $idle
       $traffic set rate $rate

        # Attach traffic source to the traffic generator
```

```
        $traffic attach-agent $source
        #Connect the source and the sink
        $ns connect $source $sink
        return $traffic
}
```

Questa procedura riceve in ingresso sei parametri: un nodo, un già creato agente Sink,la dimensione del pacchetto per le sorgenti di traffico, il tempo di end della generazione dei pacchetti, il tempo di idle (per la distribuzione esponenziale) e la frequenza di invio massima dei pacchetti.
Inizialmente la procedura crea una sorgente di traffico e la attacca al nodo, poi crea un oggetto di di tipo `Traffic/Expoo`, setta i suoi parametri di configurazione e lo attacca alla sorgente di traffico, prima di collegare la sorgente al nodo associato con l'agente Sink. Al termine di ciò essa ritorna un riferimento alla sorgente di traffico. Ora useremo la procedura per attaccare sorgenti di traffico con differenti frequenze di invio dei pacchetti ai nodi n0, n1, n2 e connetterli con i tre ricevitori Sink sul nodo n4 che devono essere precedentemente creati :

```
set sink0 [new Agent/LossMonitor]
set sink1 [new Agent/LossMonitor]
set sink2 [new Agent/LossMonitor]
$ns attach-agent $n4 $sink0
$ns attach-agent $n4 $sink1
$ns attach-agent $n4 $sink2

set source0 [attach-expoo-traffic $n0 $sink0 200 2s 1s 100k]
set source1 [attach-expoo-traffic $n1 $sink1 200 2s 1s 200k]
set source2 [attach-expoo-traffic $n2 $sink2 200 2s 1s 300k]
```

In questo esempio usiamo oggetti di tipo `LossMonitor,` dal momento che questi agenti memorizzano il numero di bytes ricevuti, molto utili per calcolare la larghezza di banda.

5.3.6.2 Registrare i dati nei trace-file

Ora dobbiamo aprire tre file di output. Le seguenti linee di codice devono apparire all'inizio del nostro file Tcl :

```
set f0 [open out0.tr w]
set f1 [open out1.tr w]
set f2 [open out2.tr w]
```

Questo file verranno poi chiusi all'interno della procedura `finish` :

```
proc finish {} {
        global f0 f1 f2
```

```
        #Close the output files
        close $f0
        close $f1
        close $f2
        #Call xgraph to display the results
        exec xgraph out0.tr out1.tr out2.tr -geometry 800x400 &
        exit 0
}
```

Questa funzione non solo chiude i file, ma richiama anche Xgraph per visualizzare il risultato.
Ora passiamo alla funzione che c permette di scrivere i dati do output della simulazione sui 3 file aperti :

```
proc record {} {
        global sink0 sink1 sink2 f0 f1 f2
        #Get an instance of the simulator
        set ns [Simulator instance]
        #Set the time after which the procedure should be
called again
        set time 0.5
        #How many bytes have been received by the traffic
sinks?
        set bw0 [$sink0 set bytes_]
        set bw1 [$sink1 set bytes_]
        set bw2 [$sink2 set bytes_]
        #Get the current time
        set now [$ns now]
        #Calculate the bandwidth (in MBit/s) and write it to
the files
        puts $f0 "$now [expr $bw0/$time*8/1000000]"
        puts $f1 "$now [expr $bw1/$time*8/1000000]"
        puts $f2 "$now [expr $bw2/$time*8/1000000]"
        #Reset the bytes_ values on the traffic sinks
        $sink0 set bytes_ 0
        $sink1 set bytes_ 0
        $sink2 set bytes_ 0
        #Re-schedule the procedure
        $ns at [expr $now+$time] "record"
}
```

Questa procedura legge il numero di bytes ricevuti dai 3 ricevitori Sink attaccati al nodo n4, poi calcola la larghezza di banda (in Mbit/s) e la scrive nei tre file di output insieme con listante di tempo corrente prima di resettare il numero di bytes ricevuti `(bytes_)` dai 3 ricevitori Sink. Infine non fa altro che rischedulare se stessa ogni 0,5 sec.

5.3.6.3 Lanciamo la Simulazione

Arrivati a questo punto non ci resta che lanciare la simulazione, ma prima dobbiamo far schedulare una serie di eventi al nostro script Tcl attraverso il seguente codice:

```
$ns at 0.0 "record"
$ns at 10.0 "$source0 start"
$ns at 10.0 "$source1 start"
$ns at 10.0 "$source2 start"
$ns at 50.0 "$source0 stop"
$ns at 50.0 "$source1 stop"
$ns at 50.0 "$source2 stop"
$ns at 60.0 "finish"

$ns run
```

Prima viene invocata la procedura “record” che verrà rischedulata ogni 0,5 secondi e poi vengono fatte partire all’istante 10 le tre sorgenti di traffico e stoppate all’istante 50. Infine all’istante 60 viene richiamata la procedura “finish” e la simulazione ha cosi’ termine.
Una volta fatta partire la simulazione una finestra come quella in figura 5.13 si dovrebbe aprire e mostrare l’andamento grafico della larghezza di banda per le 3 sorgenti di dati.

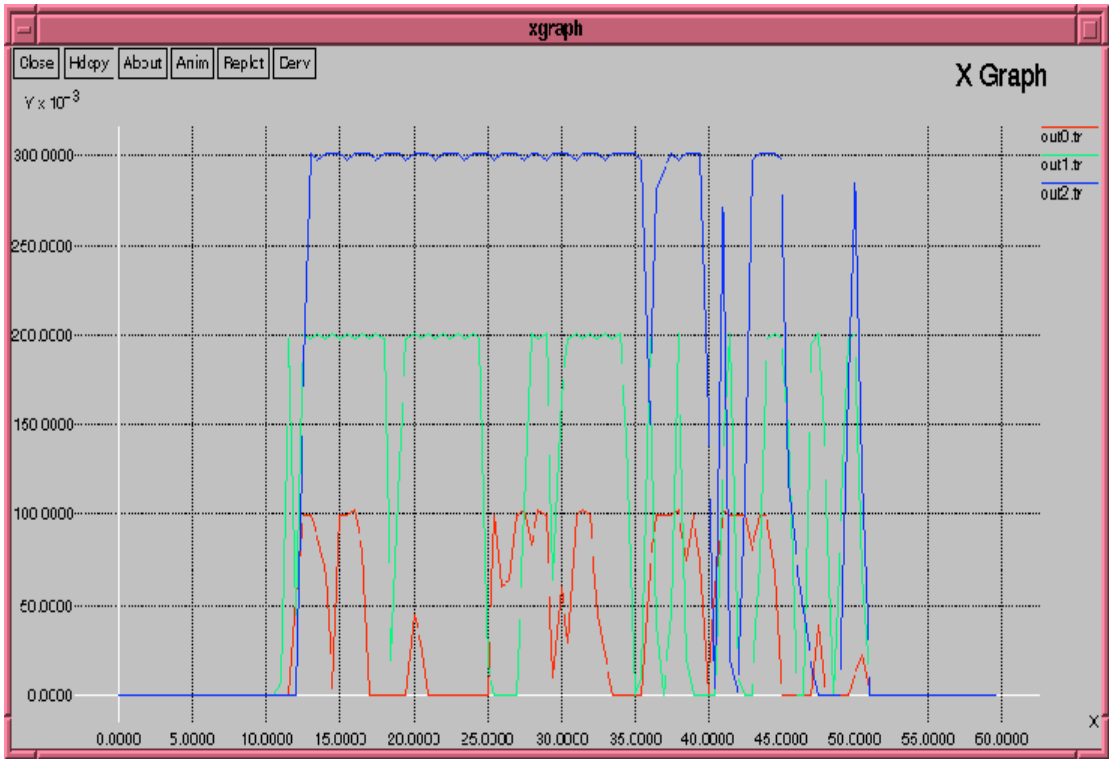

Figura 5.13 Snapshot Simulazione Xgraph

5.3.7 Creare Scenari Wireless in NS2

In questa sezione verrà illustrato come utilizzare un modello di simulazione wireless mobile in Ns2.
Inizieremo con un semplice esempio di uno scenario con due soli nodi wireless in un area di 500m x 500 m. All'inizio della simulazione i due nodi sono in punti opposti dello scenario, poi si muovono l'uno verso l'altro nella prima parte della simulazione e poi si allontanano di nuovo nella seconda parte. I pacchetti tra i due nodi iniziano ad essere scambiati non appena uno dei nodi entra nel raggio di trasmissione dell'altro e non appena si allontanano al di fuori di tale raggio i pacchetti vengono persi.
Il nostro script Tcl dovrà prima definire il tipo di ognuno dei parametri di rete che sono necessari per una simulazione wireless. Nel nostro caso imposteremo i seguenti valori :

```
#
# ======================================================================
# Define options
#
# ======================================================================
set val(chan)   Channel/WirelessChannel  ;# channel type
set val(prop)   Propagation/TwoRayGround ;# radio-propagation model
set val(ant)     Antenna/OmniAntenna      ;# Antenna type
set val(ll)      LL                       ;# Link layer type
set val(ifq)     Queue/DropTail/PriQueue  ;# Interface queue type
set val(ifqlen)  50                       ;# max packet in ifq
set val(netif)   Phy/WirelessPhy          ;# network interface type
set val(mac)     Mac/802_11               ;# MAC type
set val(rp)      DSDV                     ;# ad-hoc routing protocol
set val(nn)      2                        ;# number of mobilenodes
```

Nella parte principale del nostro script dopo aver definito una nuova istanza del simulatore ed aver aperto i trace-file su cui andare a scrivere i risultati della nostra simulazione, dovremo definire un nuovo oggetto "topologia" che tenga traccia dei movimenti dei nodi all'interno del nostro scenario (500m x 500m). Dopodichè dovremo creare l'oggetto "God"(General Operations Director) attraverso il comando `create-god $val(nn)` dove "nn" è il numero di nodi precedentemente definito. Questo è un oggetto che viene usato per memorizzare tutte quelle informazioni globali relative all'ambiente,alla rete e ai nodi, che un osservatore onnisciente dovrebbe avere, ma che gli altri partecipanti nella simulazione non dovrebbero sapere che c sia. Al momento l'oggetto "God" memorizza solo il numero totali di nodi e il cammino minimo da un nodo all'altro.
Il prossimo passo è la creazione dei nodi mobili, ma prima dobbiamo configurarli; le API di configurazione dei nodi possono essere definite come segue :

```
                                    (parameter examples)
 $ns_ node-config -addressingType   flat or hierarchical or expanded
                  -adhocRouting     DSDV or DSR or TORA
                  -llType           LL
                  -macType          Mac/802_11
                  -propType         "Propagation/TwoRayGround"
                  -ifqType          "Queue/DropTail/PriQueue"
                  -ifqLen           50
                  -phyType          "Phy/WirelessPhy"
                  -antType          "Antenna/OmniAntenna"
                  -channelType      "Channel/WirelessChannel"
                  -topoInstance     $topo
                  -energyModel      "EnergyModel"
                  -initialEnergy    (in Joules)
                  -rxPower          (in W)
                  -txPower          (in W)
                  -agentTrace       ON or OFF
                  -routerTrace      ON or OFF
                  -macTrace         ON or OFF
                  -movementTrace    ON or OFF
```

Il valore de default per queste voci è null tranne che per la voce addressingType il cui valore è "flat". Una volta fissati i valori di questi parametri procediamo alla creazione dei due nodi mobili:

```
for {set i 0} {$i < $val(nn) } {incr i} {
                set node_($i) [$ns_ node ]
                $node_($i) random-motion 0
        }
```

Con il comando "random-motion 0" disattiviamo ogni movimento casuale dei nodi, in modo tale da impostarne poi posizione e movimento. Ora che abbiamo creato i nodi mobili diamogli una posizione inziale:

```
# Fornisce le coordinate iniziali (X,Y, for now Z=0) for
node_(0) and node_(1)
#
$node_(0) set X_ 5.0
$node_(0) set Y_ 2.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 390.0
$node_(1) set Y_ 385.0
$node_(1) set Z_ 0.0
```

Quindi adesso in nodo 0 ha posizione (5,2,0) mentre il nodo 1 ha posizione (390,385,0).
Ora produciamo alcuni movimenti per i due nodi:

```
# Node_(1) inizia a muoversi verso il  node_(0)

$ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0"
$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"
```

```
# Node_(1) si allontana dal  node_(0)
$ns_ at 100.0 "$node_(1) setdest 490.0 480.0 15.0"
```

Se prendiamo come riferimento il comando di movimento del nodo1 all'istante 50, questo vuol dire che in quell'istante di tempo il nodo1 inizia a muoversi verso la destinazione (X=25, Y=20) ad una velocità di 15m/s. Questa API è usata per cambiare direzione e velocità di movimento dei nodi.
A questo punto dopo aver settato il flusso del traffico di dati tra i due nodi ed aver definito l'istante di tempo in cui il flusso di dati si deve interrompere e l'istante di stop della simulazione già precedentemente descritti nei casi di connessioni cablate, possiamo lanciare la nostra simulazione con il solito comando "`ns nome_script.tcl`".

5.3.7.1 Pattern-files per la configurazione dei Nodi

Un estensione al precedente esempio può essere l'uso di file esterni al nostro script al fine di reperire pattern relativi alle connessioni esistenti tra i nodi della simulazione e ai movimenti effettuati da questi ultimi durante tutta la durata dell'esperimento, evitando cosi' di includere nel file Tcl copiose linee di codice aumentandone leggibilità e comprensione. In questo caso infatti i movimenti dei nodi saranno letti da un file di nome scen3-test mentre le connessioni saranno lette da un altro file di nome cbr3-test. I parametri iniziali della simulazione andranno quindi modificati come segue:

```
set val(chan)        Channel/WirelessChannel
set val(prop)        Propagation/TwoRayGround
set val(netif)       Phy/WirelessPhy
set val(mac)         Mac/802_11
set val(ifq)         Queue/DropTail/PriQueue
set val(ll)          LL
set val(ant)         Antenna/OmniAntenna
set val(x)               670   ;# X dimension of the topography
set val(y)               670   ;# Y dimension of the topography
set val(ifqlen)          50            ;# max packet in ifq
set val(seed)            0.0
set val(adhocRouting)    DSR
set val(nn)              3             ;# how many nodes are
simulated
set val(cp)              "../mobility/scene/cbr-3-test"
set val(sc)            "../mobility/scene/scen-3-test"
set val(stop)          2000.0          ;# simulation time
```

E' stato inoltre aggiunto alla simulazione un ulteriore nodo ed è stata modificata anche la topografia della rete in termini di dimensioni.

A questo punto dopo la creazione dei nodi basterà caricare il contenuto dei due file scenario per completare la loro definizione attraverso i seguenti comandi:

```
# Define node movement model
#
puts "Loading connection pattern..."
source $val(cp)

# Define traffic model
#
puts "Loading scenario file..."
source $val(sc)
```

Nel file scen-3-test potremo vedere comandi per settare movimento e posizione dei nodi simili al seguente:

```
$ns_ at 50.000000000000 "$node_(2) setdest 369.463244915743 \
170.519203111152 3.371785899154"
```

5.3.7.2 Creazione Patterns di traffico casuali

Connessioni di traffico casuali di tipo TCP o CBR possono essere settate tra nodi mobili usando uno script di generazione di scenari di traffico. Questo script è disponibile nella directory di Ns2 `~ns/indep-utils/cmu-scen-gen` ed il suo nome è **`cbrgen.tcl.`** Questo script può essere eseguito da linea di comando attraverso il comando

```
ns cbrgen.tcl [-type cbr|tcp] [-nn nodes] [-seed seed] [-mc
connections]
[-rate rate]
```

il quale genererà un nuovo file nella directory corrente col nome passato da linea di comando dall'utente.
Un esempio potrebbe essere il seguente :

```
ns cbrgen.tcl -type cbr -nn 10 -seed 1.0 -mc 8 -rate 4.0 >
cbr-10-test
```

Il file generato potrà essere simile al seguente:

```
set udp_(0) [new Agent/UDP]
$ns_ attach-agent $node_(2) $udp_(0)
set null_(0) [new Agent/Null]
$ns_ attach-agent $node_(3) $null_(0)
set cbr_(0) [new Application/Traffic/CBR]
$cbr_(0) set packetSize_ 512
$cbr_(0) set interval_ 0.25
$cbr_(0) set random_ 1
$cbr_(0) set maxpkts_ 10000
$cbr_(0) attach-agent $udp_(0)
```

```
$ns_ connect $udp_(0) $null_(0)
$ns_ at 82.557023746220864 "$cbr_(0) start"
```

5.3.7.3 Creazione Patterns di movimento casuali

Il generatore di movimento casuale dei nodi è situato nella cartella `~ns/indep-utils/cmu-scen-gen/setdest.` Per farlo funzionare è necessario eseguire le seguenti operazioni:

- Spostarsi nella directory principale di Ns2 ed eseguire il comando "`configure`"
- Spostarsi nella directory `indep-utils/cmu-scen-gen/setdest` ed eseguire il comando "`make`" che provvedera a creare l'eseguibile "`setdest`".
- Eseguire il comando "`setdest`" con i seguenti argomenti :

```
./setdest [-n num_of_nodes] [-p pausetime] [-s maxspeed] [-t
simtime] \
      [-x maxx] [-y maxy] > [outdir/movement-file]
```

Un esempio potrebbe essere quello di volere creare un pattern di 20 nodi che abbiano un a velocità massima di 10m/s con una pausa tra ogni movimento di 2 s. Vogliamo inoltre che la simulazione abbia fine dopo 200s e la topologia di rete sia definita come 500m(x) x 500m(y).
Il comando risultante sarà quindi il seguente :

```
./setdest -n 20 -p 2.0 -s 10.0 -t 200 -x 500 -y 500 > scen-20-test
```

5.3.8 Implementare un Nuovo Protocollo in NS2

In questa sezione presenteremo un esempio di un nuovo protocollo che possa essere poi usato in Ns2. La nostra implementazione del nuovo protocollo si dividerà in 4 fasi principali :

- Scrittura degli header files in C++ relativi alla struttura del pacchetto da spedire e all'agente.
- Scrittura dell'Agente in C++
- Sviluppo dello script Tcl
- Modifica dei file sorgente di ns2 al fine di poter utilizzare il nostro nuovo protocollo

5.3.8.1 Gli Header Files

Il primo header file da scrivere sarà quello relativo al nuovo tipo di pacchetto da spedire che chiameremo "`Bcast_flood_packet.h`". Esso sarà strutturato similmente al seguente codice:

```
struct hdr_Bcast_flood {
  char ret;
  double send_time;
};
```

L'altro header file sarà quello relativo all'agente dove ne verrà definita l'interfaccia che chiameremo Bcast_flood.h. La sua struttura sarà simile alla seguente :

```
class Bcast_flood : public Agent {
 public:
  Bcast_flood();
  int command(int argc, const char*const* argv);
  void recv(Packet*, Handler*);
 protected:
  int off_flood_;
};
```

5.3.8.2 Il codice C++

La prima cosa da fare come precedentemente spiegato nella sezione 4.2 è creare un collegamento tra il codice C++ e quello Tcl. Per fare questo il nostro codice C++ dovrà contenere le seguenti procedure :

```
static class Bcast_floodHeaderClass : public PacketHeaderClass {
public:
  Bcast_floodHeaderClass() :
PacketHeaderClass("PacketHeader/Bcast_flood",
                                        sizeof(hdr_Bcast_flood)) {}
} class_Bcast_floodhdr;


static class Bcast_floodClass : public TclClass {
public:
  Bcast_floodClass() : TclClass("Agent/Bcast_flood") {}
  TclObject* create(int, const char*const*) {
    return (new Bcast_floodAgent());
  }
} class_Bcast_flood;
```

Il codice seguente è il costruttore per la classe Bcast_floodAgent. Esso mette in relazione le variabili a cui si dovrà accedere sia da C++ che da Tcl :

```
Bcast_floodAgent::Bcast_floodAgent(nsaddr_t id) :
Agent(PT_BCAST_FLOOD),send_hello_timer_(this)
{
  bind("packetSize_", &size_);
  bind("off_flood_", &off_flood_);
}
```

La funzione `command()` viene chiamata quando un comando Tcl per la classe Bcast_floodAgent viene eseguito :

```
int Bcast_floodAgent::command(int argc, const char*const* argv)
{
  if (argc == 2) {
    if (strcmp(argv[1], "send") == 0) {
      // Create a new packet
```

```
      Packet* pkt = allocpkt();
      // Access the Bcast_flood header for the new packet:
      hdr_Bcast_flood* hdr = (hdr_Bcast_flood*)pkt-
>access(off_flood_);
      // Set the 'ret' field to 0, so the receiving node knows
      // that it has to generate an echo packet
      hdr->ret = 0;
      // Store the current time in the 'send_time' field
      hdr->send_time = Scheduler::instance().clock();
      // Send the packet
      send(pkt, 0);
      // return TCL_OK, so the calling function knows that the
      // command has been processed
      return (TCL_OK);
    }
  }
  // If the command hasn't been processed by
Bcast_floodAgent()::command,
  // call the command() function for the base class
  return (Agent::command(argc, argv));
}
```

La funzione recv() invece definisce il comportamento del nostro Agente quando un pacchetto viene ricevuto e la sua implementazione potrebbe essere simile alla seguente:

```
void PingAgent::recv(Packet* pkt, Handler*)
{
  // Access the IP header for the received packet:
  hdr_ip* hdrip = (hdr_ip*)pkt->access(off_ip_);
  // Access the Ping header for the received packet:
  hdr_Bcast_flood* hdr = (hdr_Bcast_flood*)pkt->access(off_flood_);
  // Is the 'ret' field = 0 (i.e. the receiving node is being
pinged)?
  if (hdr->ret == 0) {
    // Send an 'echo'. First save the old packet's send_time
    double stime = hdr->send_time;
    // Discard the packet
    Packet::free(pkt);
    // Create a new packet
.....................................
.....................................
char out[100];
    // Prepare the output to the Tcl interpreter. Calculate the
round
    // trip time
    sprintf(out, "%s recv %d %3.1f", name(),
            hdrip->src_.addr_ >> Address::instance().NodeShift_[1],
            (Scheduler::instance().clock()-hdr->send_time) * 1000);
    Tcl& tcl = Tcl::instance();
    tcl.eval(out);
    // Discard the packet
    Packet::free(pkt);
  }
}
```

5.3.8.3 Il codice Tcl

La parte importante da far notare nel codice Tcl non è tanto la configurazione dei nodi in quanto è molto simile alle precedenti simulazioni, mentre è utile far vedere la procedura '`recv`', la quale viene chiamata dalla procedura recv() del C++ ogni volta che si riceve un pacchetto. Una possibile implementazione potrebbe essere la seguente :

```
Agent/Ping instproc recv {from rtt} {
        $self instvar node_
        puts "node [$node_ id] received ping answer from \
              $from with round-trip-time $rtt ms."
}
```

5.3.8.4 Modifica dei file sorgenti di NS2

Per prima cosa è necessario modificare il file packet.h situato nella directory ns-x.x/common/ in modo da poter aggiungere alle strutture presenti nel file il pacchetto appena creato rendendolo così disponibile ad Ns2 :

```
enum packet_t {
        PT_TCP,
        PT_UDP,
        ......
        // insert new packet types here
        PT_TFRC,
        PT_TFRC_ACK,
        PT_BCAST_FLOOD,     //  ID del pacchetto per il nostro
Bcast_flood Agent
        PT_NTYPE // This MUST be the LAST one
};

class p_info {
public:
        p_info() {
                name_[PT_TCP]= "tcp";
                name_[PT_UDP]= "udp";
                 ...........
                name_[PT_TFRC]= "tcpFriend";
                name_[PT_TFRC_ACK]= "tcpFriendCtl";

                name_[PT_BCAST_FLOOD]="Bcast_flood";

                name_[PT_NTYPE]= "undefined";
        }
        .....
 }
```

Come secondo passo è necessario modificare il file **ns_packet.tcl** nella directory ns-x.x/tcl/lib/ nel seguente modo:

```
foreach prot {
      AODV
      . . . . . .
```

```
        MIP
        Ping
        Flood_P
        Bcast_flood                    # Dobbiamo aggiungere qui il nostro agente
        . . . . . .
        TORA
        GAF
        UMP
        Pushback
        SCTP
        Smac
        NV
} {
        add-packet-header $prot
}
```

Il successivo file da modificare è **ns-default.tcl** nella directory ns-x.x/tcl/lib/ aggiungendo le seguenti linee di codice :

```
Agent/Bcast_flood set packetSize_ 1024
```

Per integrare le funzionalità del DiscoveryPartner (nel nostro caso DummyAgent) in Ns2 bisogna aggiungere il codice ad esso relativo e cioè :

```
DummyAgent set addr_ 0
DummyAgent set port_ 900
DummyAgent set shift_ 0
DummyAgent set mask_ [AddrParams set ALL_BITS_SET]
DummyAgent set debug_ false
```

La stessa linea di codice che c indica la dimensione del nostro pacchetto dovrà essere inserita anche nel file **ns_tcl.cc** nella directory ns-x.x/gen/, mentre per quanto riguarda l'integrazione del DiscoveryPartner (nel nostro caso DummyAgent) in Ns2 in questo file vanno aggiunte le seguenti linee di codice :

```
Node/MobileNode instproc attach-dummy {} {\n\
$self instvar dummy_ address_ ll_ \n\
\n\
set dummy_ [new DummyAgent]\n\
\n\
$dummy_ set mask_ [AddrParams NodeMask 1]\n\
$dummy_ set shift_ [AddrParams NodeShift 1]\n\
set nodeaddr [AddrParams addr2id [$self node-addr]]\n\
\n\
\n\
\n\
$dummy_ set addr_ $nodeaddr\n\
$dummy_ set port_ 900\n\
\n\
\n\
$dummy_ target [$self entry]\n\
$ll_(0) up-target $dummy_\n\
```

```
}\n\
\n\
Node/MobileNode instproc unset-dummy {} {\n\
$self instvar dummy_\n\
\n\
$dummy_ set-dummyagent 0\n\
\n\
}\n\
```

La modifica successiva riguarda il **Makefile** di Ns2; infatti al suo interno va aggiunta la linea di codice che dice ad Ns2 che il nostro Agente fa parte dei protocolli installati a disposizione :

```
Broadcast/Bcast_flood.o \
```

dove Broadcast è la cartella all'interno della directory di Ns2 dove sono contenuti tutti i sorgenti del nostro protocollo.
L'ultima modifica riguarda ancora Il DiscoveryPartner (nel nostro caso DummyAgent) e in particolare il file che va modificato è **ns_mobilenode.tcl** nella directory ns-x.x/tcl/lib/ :

```
Node/MobileNode instproc attach-dummy {} {
        $self instvar dummy_ address_ ll_

        set dummy_ [new DummyAgent]

      $dummy_ set mask_ [AddrParams NodeMask 1]
      $dummy_ set shift_ [AddrParams NodeShift 1]
      set nodeaddr [AddrParams addr2id [$self node-addr]]

      #$gafpartner_ set addr_ [expr ( ~([AddrParams NodeMask 1] << \
      #           [AddrParams NodeShift 1]) & $nodeaddr )]


      $dummy_ set addr_ $nodeaddr
      $dummy_ set port_ 900

      #puts [$gafpartner_ set addr_]

        $dummy_ target [$self entry]
      $ll_(0) up-target $dummy_
}

Node/MobileNode instproc unset-dummy {} {
      $self instvar dummy_

      $dummy_ set-dummyagent 0

}
```

Ora per quanto riguarda simulazioni di nodi mobili c sono altre modifiche da fare e più precisamente la prima modifica è relativa al file **cmu-trace.cc** nella directory ns-x.x/trace/ :

. . . .

```
            case PT_CBR:
                  format_rtp(p, offset);
                  break;
            case PT_BCAST_FLOOD:
                  format_Bcast_flood(p,offset);
                  break;
              case PT_DIFF:
                  break;
            case PT_GAF:
            case PT_PING:
                  break;
            ....

void CMUTrace::format_Bcast_flood(Packet* p,int offset)
{
      struct hdr_cmn *ch = HDR_CMN(p);
      struct hdr_ip *ih = HDR_IP(p);

      // hack the IP address to convert pkt format to hostid format
      // for now until port ids are removed from IP address. -Padma.
      int src = Address::instance().get_nodeaddr(ih->saddr());
      int dst = Address::instance().get_nodeaddr(ih->daddr());

      if (newtrace_) {
       sprintf(pt_->buffer() + offset,
         "-Is %d.%d -Id %d.%d -It %s -Il %d -If %d -Ii %d -Iv %d ",
                src,                            // packet src
                ih->sport(),                    // src port
                dst,                            // packet dest
                ih->dport(),                    // dst port
                packet_info.name(ch->ptype()),  // packet type
                ch->size(),                      // packet size
                ih->flowid(),                    // flow id
                ch->uid(),                       // unique id
                ih->ttl_);                       // ttl
      } else {
       printf(pt_->buffer() + offset, "------- [%d:%d %d:%d %d %d] ",
            src, ih->sport(),
            dst, ih->dport(),
            ih->ttl_, (ch->next_hop_ < 0) ? 0 : ch->next_hop_);
      }
}
```

Ora al fine di assicurare un corretto funzionamento c dobbiamo assicurare che anche il file **cmu-trace.h** rifletta i cambiamenti effettuati nel file **cmu-trace.cc** e quindi apporteremo le seguenti modifiche :

```
....
      void  format_msg(Packet *p, int offset);
      void  format_tcp(Packet *p, int offset);
      void   format_Bcast_flood(Packet *p, int offset);
      void    format_sctp(Packet *p, int offset);
      void  format_rtp(Packet *p, int offset);
      void  format_tora(Packet *p, int offset);
....
```

Ora non resta che modificare le variabili di ambiente di Ns2 al fine di impostare i giusti percorsi per tutti gli strumenti che useremo nella nostra simulazione e più precisamente :

PATH = $PATH:/root/NS/ns-allinone-2.27/ns-2.27/bin:/root/NS/ns-allinone-2.27/tcl8.4.5/unix:/root/NS/ns-allinone-2.27/tk8.4.5/unix

LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/root/NS/ns-allinone-2.27/otcl-1.8:/root/NS/ns-allinone-2.27/lib

TCL_LIBRARY = $TCL_LIBRARY:/root/NS/ns-allinone-2.27/tcl8.4.5/library

A questo punto non resta che fare il "**make depend**" per ricompilare tutte le dipendenze dei file modificati e poi successivamente se tutto è andato a buon fine fare il "**make**" di Ns2.
Ora non resta altro che lanciare la nostra applicazione con il comando

```
ns Bcast_flood.tcl
```

Cap. 6

Analisi di protocolli di comunicazione

Il nostro obiettivo in questo lavoro di tesi, è quello di effettuare uno studio sulle prestazioni offerte una determinata tipologia di protocolli di comunicazione in ambienti wireless, cioè i protocolli di Gossip. In particolare volevamo analizzarne il comportamento in scenari di reti complessi, nei quali far interagire numerosi dispositivi grazie ad un continuo scambio di informazioni, estrapolando poi i dati dalle varie simulazioni effettuate utilizzando questo protocollo di comunicazione.
E' particolarmente difficile pensare di poter avere a disposizione un grande gruppo, dell'ordine delle centinaia, migliaia, di persone in movimento all'interno di una certa zona fornendo a ciascuna un dispositivo wireless, farli interagire al fine di effettuare poi una analisi delle prestazioni ottenute variando di volta in volta determinati parametri di set-up. Dalla necessità di poter analizzare le prestazioni di vari scenari di rete, in cui un gran numero di dispositivi eterogenei interagiscono tra loro attraverso l'utilizzo di diversi protocolli di comunicazione, nasce proprio il bisogno di effettuare delle simulazioni attraverso l'utilizzo di ambienti software ad-hoc. E' stato possibile ovviare a questo problema attraverso l'utilizzo dell'ambiente di simulazione di rete Ns2.
In particolare attraverso l'utilizzo di questo strumento di simulazione, vogliamo analizzare alcune metriche come per esempio la percentuale di stazioni che ricevevano un certo numero di messaggi spediti,all'interno dello scenario di rete considerato. Inoltre era nostro interesse cercare di capire quanto i protocolli considerati fossero efficienti anche in termini di costo del numero di messaggi inviati, e quindi essere in grado di valutare l'effettivo grado di risparmio in termini di consumo di energia che queste trasmissioni potevano comportare. E' quindi molto importante riuscire ad individuare protocolli di comunicazione che minimizzassero il numero di messaggi trasmessi e quindi consentissero di conseguenza di massimizzare il numero di dispositivi raggiungibili nelle loro trasmissioni.

6.1 Le simulazioni effettuate

In questo lavoro di tesi abbiamo implementato un protocollo di Gossip in Ns2, analizzandone il comportamento in diverse tipologie di scenari e variandone di volta in volta le condizioni di lavoro. Lo scenario più generale in cui ci si può trovare a lavorare è quello casuale dove tutti i nodi sono appunto disposti casualmente all'interno della nostra rete.
E' stato quindi scelto di valutare le prestazioni del protocollo creato in uno scenario di questo tipo, analizzando metriche come la percentuale di

nodi raggiunti dai messaggi inviati e l'energia consumata nelle trasmissioni. Più in particolare nelle simulazioni che utilizzano questo scenario, vengono prima analizzati i dati ottenuti nel caso di invio di 1 solo messaggio trasmesso a tutti i nodi della rete, e successivamente nel caso di una finestra di trasmissione di dimensione variabile da 2 a 4 messaggi. Nei casi relativi all'invio di 1 solo messaggio viene analizzata la percentuale di stazioni che lo ricevono mentre nei casi relativi alla trasmissione di una finestra variabile vengono analizzati i dati ottenuti relativi alla percentuale di nodi che ricevono almeno 1 messaggio, almeno 2, almeno 3 e 4 messaggi. Sia nel caso di invio di un singolo pacchetto che nella finestra di dimensione 4 inoltre viene valutata sia l'energia consumata nelle comunicazioni tra i vari nodi dal singolo dispositivo, sia l'energia consumata globalmente dall'insieme dei dispositivi della rete.
Per meglio valutare le prestazioni delle nostre simulazioni compareremo poi i dati ottenuti con quelli di un generico protocollo di flooding mettendo in risalto il guadagno sia in termini di copertura della rete che di energia spesa dal sistema nelle comunicazioni fra i vari nodi.

6.1.1 Il Protocollo di Gossip

Per realizzare il protocollo di Gossip da poter utilizzare nelle nostre simulazioni abbiamo dovuto implementare un nuovo agente che permettesse ai dispositivi di rete di interagire tra loro effettuando comunicazioni di tipo broadcast a tutti gli altri, e consentisse di minimizzare il numero di messaggi inviati da un nodo all' altro attraverso l'utilizzo di trasmissioni effettuate con una probabilità prestabilita. Ogni nodo una volta ricevuto il messaggio, dopo aver controllato di non averlo già ricevuto in precedenza si limiterà a schedulare un timer per la ritrasmissione del pacchetto ai propri vicini.
Il funzionamento dell'agente stesso quindi può essere suddiviso in due fasi essenziali: quella di trasmissione e quella di ricezione.
Nella prima fase quindi il nostro agente si occuperà di creare un nuovo pacchetto, settarne l'header in modo tale da effettuare una trasmissione di tipo broadcast e successivamente, se la probabilità casualmente generata al suo interno risulta minore di quella di gossip, inviare il messaggio a tutti i suoi vicini.
Nella fase di ricezione invece il nostro agente si occuperà di estrarre le informazioni dall'header del pacchetto ricevuto e dopo aver verificato di non averlo già ricevuto, rischedulare la trasmissione attraverso l'utilizzo di un timer interno all'agente, del pacchetto stesso a tutti i suoi vicini.
Di seguito è illustrato il diagramma di sequenza delle azioni svolte dall'agente nella fase di trasmissione di un messaggio (Figura 6.1).

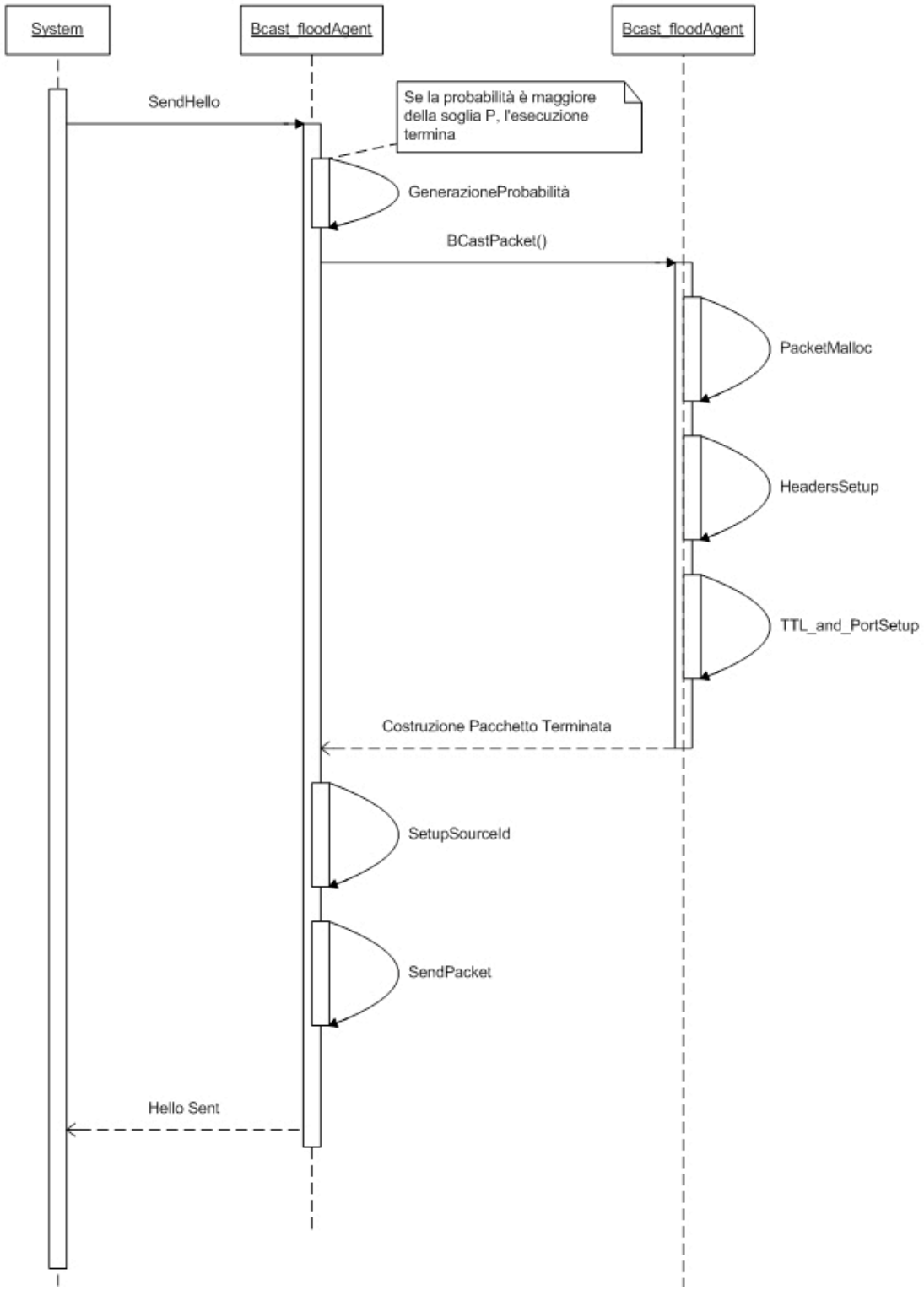

Figura 6.1 Diagramma di sequenza della fase di trasmissione

Possiamo vedere come, quando un nodo deve trasmettere un messaggio agli altri nodi suoi vicini, viene invocato il metodo SendHello() dell'agente Gossip Bcast_flood. All'interno di questo metodo viene

generata una probabilità casuale; se questa probabilità è minore della probabilità di Gossip ricavata dall'interazione con il codice Tcl, viene richiamato il metodo BcastPacket(), altrimenti la trasmissione termina incrementando il numero di pacchetti risparmiati. All'interno del metodo BcastPacket() del nostro agente viene poi costruito il pacchetto da trasmettere in broadcast a tutti i vicini del nodo considerato; infatti viene prima allocato un nuovo pacchetto, ne vengono settati gli header ed infine impostata la porta di trasmissione, l'indirizzo di destinazione di tipo broadcast ed il tempo di vita del messaggio. Il controllo ritorna poi al metodo SendHello(), il quale dopo aver settato i campi dell'header del nostro pacchetto relativi all'id del nodo che ha trasmesso il messaggio e l'identificativo del pacchetto stesso provvede ad invocare il metodo send() che spedisce il messaggio in broadcast a tutti i vicini inoltrandolo al mezzo. Ora analizziamo l'altra fase, e cioè quella di ricezione di un messaggio illustrata in Figura 6.2. Possiamo vedere come quando un nodo riceve un nuovo messaggio viene invocato il metodo recv() dell'agente Gossip il quale al suo interno controlla se il pacchetto ricevuto è del tipo BCAST_FLOOD_HELLO.
Se il tipo non corrisponde la ricezione termina subito e il pacchetto viene scartato, altrimenti viene richiamato il metodo RecvHello() del nostro agente che si occuperà di estrarre dall'header del messaggio, l'id della sorgente del messaggio e l'identificativo del pacchetto stesso. Se questo è già stato ricevuto in precedenza viene scartato e la trasmissione termina, altrimenti viene aggiunto alla lista dei pacchetti ricevuti per evitare ricezioni multiple dello stesso pacchetto. Infine il metodo non fa altro che settare un timer interno allo scadere del quale verrà invocato il metodo sendHello() al fine di ritrasmettere il pacchetto ricevuto ai suoi vicini.

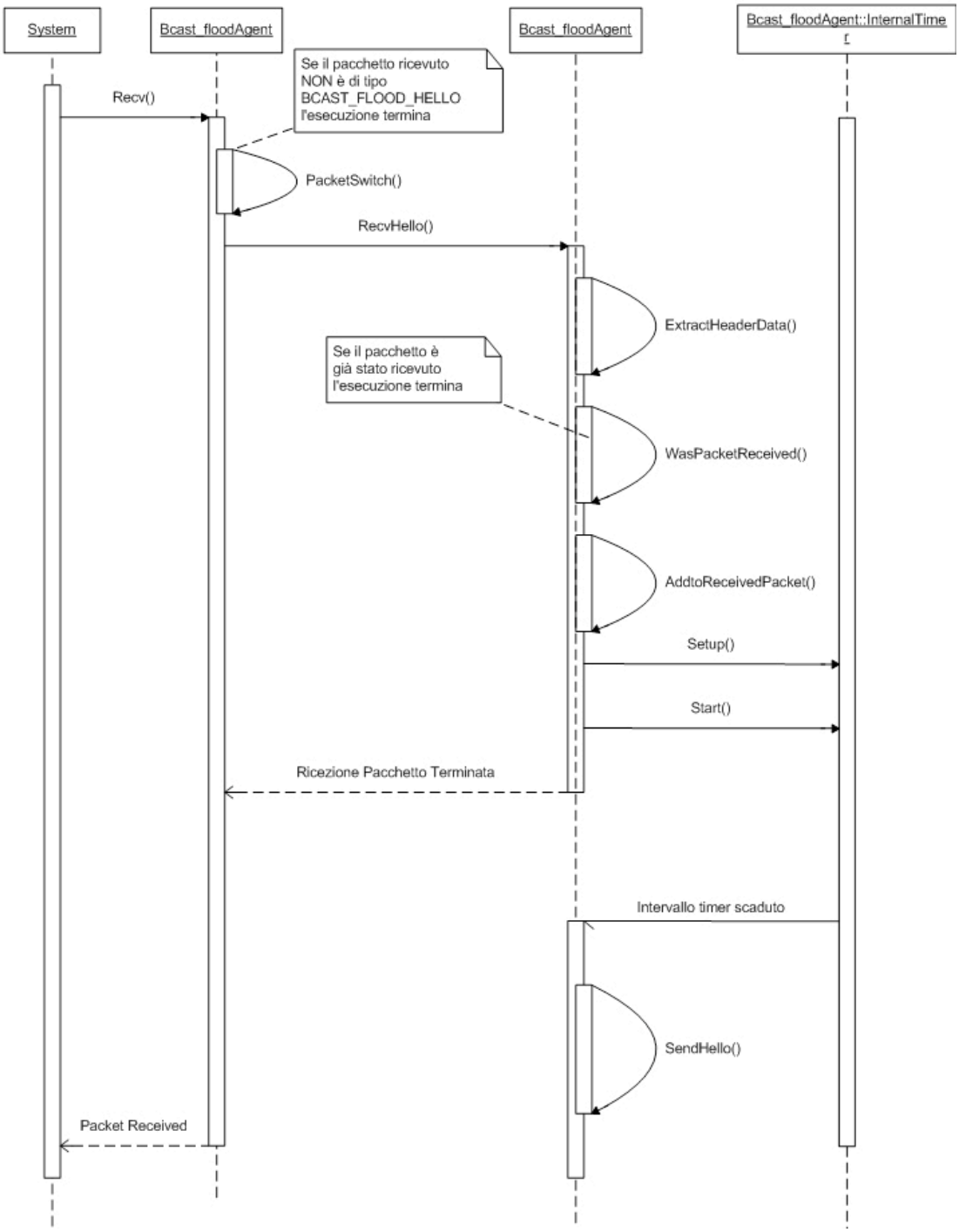

Figura 6.2 Diagramma di sequenza della fase di ricezione

6.1.1.1 Implementazione del Packet Header

La struttura interna del file relativo all'header dei pacchetti utilizzati nelle trasmissioni all'interno delle simulazioni è la seguente:

```
/*
 * Macro relative all'header del protocollo Bcast_Flood
 */

#define HDR_BCAST_FLOOD(p)
   ((struct hdr_Bcast_flood*)hdr_Bcast_flood::access(p))
#define HDR_BCAST_FLOOD_HELLO(p)
   ((struct hdr_Bcast_flood_hello*)hdr_Bcast_flood::access(p))

/*
 * Header generico Bcast_flood – condiviso da tutti
 */

struct hdr_Bcast_flood {
        u_int8_t        type;
        u_int32_t       srcid;

      // Metodi per l'accesso all'header
      static int offset_; // richiesto  dal PacketHeaderManager
      inline static int& offset() { return offset_; }
      inline static hdr_Bcast_flood* access(const Packet* p) {
            return (hdr_Bcast_flood*) p->access(offset_);
      }
};


/*Pacchetto di HELLO contenente le informazioni*/
struct hdr_Bcast_flood_hello {
      u_int8_t        hello_type;
      u_int32_t       hello_srcid;
      u_int32_t       hello_number;
      u_int32_t       time_to_live;
};
```

La prima struttura denominata *hdr_Bcast_flood* riguarda la parte di header comune a tutti i pacchetti, mentre la seconda struttura denominata *Bcast_flood_hello*, che è quella che più ci interessa, contiene tutte le informazioni che vogliamo contenga il pacchetto trasmesso dai vari nodi, quali l'indirizzo della sorgente che lo ha trasmesso, il numero di sequenza del pacchetto e il tempo di vita indicante il numero di hop che questo pacchetto può raggiungere prima di essere scartato.

6.1.1.2 Implementazione dell'Agente Gossip

Successivamente si è passati all'implementazione dell'agente Gossip denominato Bcast_flood. Il codice relativo all'interfaccia dell'agente è il seguente:

```
// Interfaccia dell'agente Bcast_flood

class Bcast_floodAgent;
// Sender uses this timer to
// schedule next app data packet transmission time
class SendHelloTimer : public TimerHandler {
 public:
    SendHelloTimer(Bcast_floodAgent * a) : TimerHandler(), a_(a) {}
//        inline virtual void expire(Event*);
        void      expire(Event*);
 protected:
        Bcast_floodAgent* a_;
       Event      intr;
};

class Bcast_floodAgent : public Agent {
//friend class SendTimer;
 public:
  Bcast_floodAgent(nsaddr_t id);
  int command(int argc, const char*const* argv);
   void recv(Packet*, Handler*);
  Packet* BCastPacket();
  void timeout(int);

  /*Boundaries Maintenance protocol*/
  void sendHello();
  void recvHello(Packet*);

  //The hello period
  static float hello_period_;
  //The hello delay
  static float hello_delay_;
  // The hello packet send probability
  static float hello_prob_;
//  void recv(Packet*, Handler*);
  static int hello_num_;
private:

  // indentificatore del nodo
  int nid_;
  int packet_id_;
  int packet_life_;
  int* packet_received_;
  int* packet_sent_;
  int length_;


  //The Hello Timer
  SendHelloTimer send_hello_timer_;
};

/*================    DUMMY AGENT    ============*/
```

```
/*
 * The Dummy Agent helps in filtering broadcast messages
 */

class DummyAgent : public Connector {
public:
        DummyAgent();
        void recv(Packet *p, Handler *h);
protected:
      int command(int argc, const char*const*argv);
        ns_addr_t here_;
      int dummyagent_;
        int mask_;
        int shift_;

};
#endif
```

All'inizio di questo file viene dichiarata l'interfaccia della classe *SendHelloTimer* derivante dalla classe Timer già implementata all'interno delle librerie C++ che verrà usata all'interno del nostro agente per schedulare gli invii dei pacchetti ad intervalli regolari. Infine viene presentato il prototipo dell' agente *DummyAgent*, che verrà descritto più approfonditamente in seguito necessario per aiutare il nostro agente nel filtrare i messaggi di tipo broadcast.

```
// Implementazione dell'agente Bcast_flood

int hdr_Bcast_flood::offset_;

static class Bcast_floodHeaderClass : public PacketHeaderClass {
public:
  Bcast_floodHeaderClass() :
PacketHeaderClass("PacketHeader/Bcast_flood",
sizeof(hdr_Bcast_flood)) { bind_offset(&hdr_Bcast_flood::offset_);}
} class_Bcast_floodhdr;


static class Bcast_floodClass : public TclClass {
public:
  Bcast_floodClass() : TclClass("Agent/Bcast_flood") {}
 TclObject* create(int argc , const char*const* argv){
    return (new Bcast_floodAgent((nsaddr_t) atoi(argv[4])));
  }
  virtual void bind();
  virtual int method (int argc, const char*const* argv);
} class_Bcast_flood;

static class DummyClass : public TclClass {
public:
        DummyClass() : TclClass("DummyAgent") {}
        TclObject* create(int, const char*const*) {
                return (new DummyAgent());
```

```
        }
} class_dummyagent;
```

In questa prima parte vengono implementate le classi Bcast_floodClass e DummyClass. Attraverso queste classi vengono creati gli oggetti corrispondenti nella gerarchia interpretata permettendo ai due linguaggi C++ e Tcl di interagire tra loro. I loro metodi verranno implementati successivamente.

```
void Bcast_floodClass::bind()
{
  TclClass::bind();

  add_method("hello_period");
  add_method("hello_delay");
  add_method("hello_prob");
  add_method("hello_num");

 }

 int Bcast_floodClass::method(int ac, const char*const* av)
{
      Tcl& tcl = Tcl::instance();
      char out[100];
      int argc = ac - 2;
      const char*const* argv = av + 2;

      if (argc == 2) { // letture da TCL di parametri statici (x
tutti i nodi)

               if (strcmp(argv[1], "hello_period") == 0) {
                    tcl.resultf("%f", Bcast_floodAgent::hello_period_);
                    return (TCL_OK);
               } else if (strcmp(argv[1], "hello_delay") == 0) {
                    tcl.resultf("%f", Bcast_floodAgent::hello_delay_);
                    return (TCL_OK);
               }
                else if (strcmp(argv[1], "hello_prob") == 0) {
                    tcl.resultf("%f", Bcast_floodAgent::hello_prob_);
                    return (TCL_OK);
               }
                else if (strcmp(argv[1], "hello_num") == 0) {
                    tcl.resultf("%d", Bcast_floodAgent::hello_num_);
                    return (TCL_OK);
               }

      }
      else if (argc==3) { // scritture variabili statiche
               if(strcmp(argv[1],"hello_period")==0) {
                    Bcast_floodAgent::hello_period_=atof(argv[2]);
                    return (TCL_OK);
               }
                 else if(strcmp(argv[1],"hello_delay")==0) {
                    Bcast_floodAgent::hello_delay_=atof(argv[2]);
                    return (TCL_OK);
               }
```

```
            else if(strcmp(argv[1],"hello_prob")==0) {
                  Bcast_floodAgent::hello_prob_=atof(argv[2]);
                  return (TCL_OK);
            }
             else if(strcmp(argv[1],"hello_num")==0) {
                  Bcast_floodAgent::hello_num_=atoi(argv[2]);
                  return (TCL_OK);
            }
      }

    return TclClass::method(ac, av);
}

float Bcast_floodAgent::hello_period_;
float Bcast_floodAgent::hello_delay_;
float Bcast_floodAgent::hello_prob_;
int Bcast_floodAgent::hello_num_;
```

In questa sezione viene effettuato il binding tra le variabili membro del nostro agente e le corrispondenti variabili dello script Tcl. Le variabili *hello_period* ed *hello_delay* riguardano il periodo al termine del quale viene schedulato l'invio di un pacchetto. I valori di queste variabili vengono combinati con altri valori generati casualmente ad ogni invio di un pacchetto attraverso una formula lineare. Il valore che si ottiene è il periodo di scheduler del messaggio da spedire.

```
Bcast_floodAgent::Bcast_floodAgent(nsaddr_t id) :
Agent(PT_BCAST_FLOOD),send_hello_timer_(this)
{
  int j=0;
  nid_=id;
  packet_id_=-1;
  packet_life_=500;
  packet_received_= NULL;
  packet_sent_= NULL;
  length_=0;
  int seme= time(NULL);
  srand(seme);


  bind("packetSize_", &size_);
}

int Bcast_floodAgent::command(int argc, const char*const* argv)
{
   if (argc == 2) {
        /*
         *This command starts the Hello packets delivery on the node
that receives it.
         *Moreover it schedules the Check Neighbors background
routine.
         */
        if(strcmp(argv[1],"startHello") == 0) {
        packet_id_ = packet_id_ + 1;

          sendHello();
```

```
          return (TCL_OK);
        }
  }

  return (Agent::command(argc, argv));
}

//==============================RECEIVER AND
DISPATCHER=====================================//
void Bcast_floodAgent::recv(Packet* pkt, Handler*) {
  // Access the Bcast_flood header for the received packet:
  hdr_Bcast_flood* hdr = HDR_BCAST_FLOOD(pkt);

  switch (hdr->type)
      {
           case BCAST_FLOOD_HELLO:
             recvHello(pkt);
             Packet::free(pkt);
             break;
      }
  }

  //=======================ACTION DISPATCHER WHEN TIMERS
FIRE====================//

  /*This method is called when the discovery timeout expires*/
  void Bcast_floodAgent::timeout (int t)
  {
   switch (t) {
        case SEND_HELLO:
                /// CODE HERE
                  sendHello();
        break;
   }
  }
```

In questa sezione viene implementato il metodo *command()* del nostro agente. Quando da Tcl viene lanciato il comando "startHello" viene subito richiamato il metodo sendHello() che si occuperà di inviare il messaggio a tutti i suoi vicini. Infine vengono implementati il metodo recv() e il metodo timeout(). Ogni qual volta un nodo riceve un messaggio l'agente invoca il metodo recv() solamente se il tipo di pacchetto corrisponde a quello precedentemente definito e cioè BCAST_FLOOD_HELLO. Mentre il metodo timeout() viene richiamato dalla classe SendHelloTimer ogni qual volta scade il periodo di scheduler del pacchetto termina, richiamando il metodo sendHello().

```
/*
 *This method only sends a new Hello message and reschedules the
Hello Period timer
 */

void Bcast_floodAgent::sendHello()
{
     Tcl& tcl = Tcl::instance();
```

```
   float p= Bcast_floodAgent::hello_prob_;
   char str[100];
    int c_p= Bcast_floodAgent::hello_num_;

    if((nid_==0)&&(length_==0))
    {

      packet_received_= new int[c_p];
      packet_sent_= new int[c_p];
      for(int w=0;w<c_p;w++)
      {
            packet_received_[w]=1;
            packet_sent_[w]=0;
      }
      length_=c_p;
    }
  float gen;
 char vett[9];
       char bu[100];
      gen = (float) rand()/RAND_MAX;

   char buff[100];
   if((gen <= p)||(nid_==0))
   {
    sprintf(buff,"probability %f",gen);
   tcl.eval(buff);
    // Create a new packet
    Packet* pkt = BCastPacket();
    // Access the Density header for the new packet
    hdr_Bcast_flood_hello* hdr = HDR_BCAST_FLOOD_HELLO(pkt);
    hdr->time_to_live= packet_life_;
    hdr->hello_type = BCAST_FLOOD_HELLO;
    hdr->hello_srcid = nid_;
    hdr->hello_number= packet_id_;
    packet_sent_[packet_id_]=1;
    float rit,x,y,k;
    char out[100];
   sprintf (out, "outSendHello %d %d", nid_,packet_id_);
    // Send the packet
     tcl.eval(out);
    send(pkt, 0);
    x=Bcast_floodAgent::hello_period_;
    y= (float) rand()/RAND_MAX;
    k=Bcast_floodAgent::hello_delay_;
    rit=x+y*k;
    sprintf (out, "prova %f %f", Scheduler::instance().clock(),
rit);
    tcl.eval(out);
    if(nid_==0)
    {
      packet_id_= packet_id_ + 1;
      packet_life_=50;
    }
   if((packet_id_< c_p)&&(packet_sent_[packet_id_]==0))
       send_hello_timer_.resched(rit);
  }
  else{
   sprintf(buff, "%s rejected %f", name(),gen);
   tcl.eval(buff);
   }
```

```
}

  //Functions that prepares a broadcast packet
  Packet* Bcast_floodAgent::BCastPacket()
  {
      // Create a new packet
      Packet* pkt = allocpkt();
      // Access the IP and common headers for the new packet
      hdr_ip* hdrip= hdr_ip::access(pkt);
      hdr_cmn* hdrc= hdr_cmn::access(pkt);
      // Send in Broadcast to all neighbors
      hdrc->next_hop_ = IP_BROADCAST;
      hdrc->addr_type_ = NS_AF_INET;
      hdrip->daddr() = IP_BROADCAST;
      hdrip->dport()=900;
      hdrip->ttl() = packet_life_;

      return (pkt);
  }
```

Il metodo sendHello() consente al nostro agente di creare un pacchetto, impostarne i relativi campi e poi inviarlo in broadcast a tutti i vicini del nodo a cui è fa riferimento.

```
void SendHelloTimer::expire(Event*) {
   a_->timeout(SEND_HELLO);
}


/*
 *This method, called upon the reception of an Hello packet, updates
the entry in the current Neighbors
 *table or inserts new records in that table if the sender was
previously unseen.
 */
void Bcast_floodAgent::recvHello(Packet* pkt) {
Tcl& tcl = Tcl::instance();
    hdr_Bcast_flood_hello* hdr=HDR_BCAST_FLOOD_HELLO(pkt);
    int snd=hdr->hello_srcid;
    int num= hdr->hello_number;
    int n = Bcast_floodAgent::hello_num_;
    char str[100];
    float y=0;
    if((nid_!=0)&&(length_==0))
    {
      packet_received_= new int[n];
      packet_sent_= new int[n];
      for(int w=0;w<n;w++)
      {
            packet_received_[w]=0;
            packet_sent_[w]=0;
      }
      length_=n;
    }
      if(packet_received_[num]==0)
    {
      packet_id_=num;
      packet_life_= hdr->time_to_live -1;
```

```
      float x;
      char out[100];
      char out2[100];
      float time=Scheduler::instance().clock();
      sprintf(out, "%s received %d %d", name(),snd,num);

      tcl.eval(out);
      packet_received_[num]=1;
      if(send_hello_timer_.status()!=TIMER_PENDING)
send_hello_timer_.resched(Random::uniform()*Bcast_floodAgent::hello_
delay_);
     }
}
```

Nel metodo recvHello() vengono estratti dal pacchetto di tipo BCAST_FLOOD_HELLO i dati relativi alla sorgente del messaggio e al suo numero di sequenza. Se il messaggio con quel determinato id è stato già ricevuto lo si ignora, altrimenti viene decrementato il tempo di vita del pacchetto e viene rischedulato l'invio di un pacchetto con lo stesso numero di sequenza a tutti i vicini del nodo considerato, attivando il timer della classe Send_Hello_Timer.

```
/*==============     DummyAgent    ==========*/
/*
 *Add Density Partner class to manage broadcast messages: this is
fundamental stuff
 */
  DummyAgent::DummyAgent() : Connector(),
mask_(0xffffffff),shift_(8)
    {
        bind("addr_", (int*)&(here_.addr_));
        bind("port_", (int*)&(here_.port_));
        bind("shift_", &shift_);
        bind("mask_", &mask_);
    }

void DummyAgent::recv(Packet* p, Handler *h)
{
        hdr_ip* hdr = hdr_ip::access(p);
      hdr_cmn *hdrc = hdr_cmn::access(p);

      if ( hdrc->ptype() == PT_BCAST_FLOOD )
               if (((u_int32_t)hdr->daddr()) == IP_BROADCAST)  hdr-
>daddr() = here_.addr_;
  target_->recv(p,h);
}


int DummyAgent::command(int argc, const char*const* argv)
{
  if (argc == 3)
     {
          if (strcmp (argv[1], "set-dummyagent") == 0)
          {
               dummyagent_ = atoi(argv[2]);
               return (TCL_OK);
              }
     }
```

```
 return Connector::command(argc, argv);
}
```

Questa ultima sezione riguarda l'implementazione dell'agente Dummy. Esso agisce da partner del nostro agente Bcast_flood filtrando per lui i pacchetti broadcast in arrivo. Più in particolare, l'agente Dummy riceve i pacchetti direttamente dallo strato di collegamento dati (Link Layer) e li inoltra al nodo considerato. Durante questa fase l'agente Dummy controlla se l'indirizzo di destinazione del messaggio è di tipo broadcast; in caso affermativo cambia l'indirizzo di destinazione 255.255.255.255 con l'indirizzo del nodo corrente. Questo permette di ricevere i pacchetti di tipo broadcast senza che questi vengano poi scartati dal protocollo di routing.

6.1.1.3 Lo script Tcl

Nello script Tcl abbiamo definito lo scenario di comunicazione nel quale il nostro agente di Gossip ha operato. Nella parte iniziale vengono definiti i parametri che definiscono le caratteristiche della topologia di rete e quelle del modello di comunicazione, poi utilizzati all'interno del nostro script.

```
set val(chan)           Channel/WirelessChannel

set val(prop)           Propagation/TwoRayGround

set val(netif)          Phy/WirelessPhy

set val(mac)            Mac/802_11

set val(ifq)            Queue/DropTail/PriQueue

set val(ll)             LL

set val(ant)            Antenna/OmniAntenna

set val(x)              3000   ;# X dimension of the topography

set val(y)              3000   ;# Y dimension of the topography

set val(sc)             "./Broadcast/scenario-500-test"

set val(nn)             500          ;# number of nodes

set val(stop)           500.0          ;# simulation time

set val(tr)             tempElection.tr          ;# trace file

set val(lm)             "off"           ;# log movement

set val(energymodel)    EnergyModel     ;

set val(initialenergy)  500             ;# Initial energy in Joules
```

Il primo parametro “chan” indica che il canale di comunicazione utilizzato in questa simulazione è ti tipo wireless. Il secodo parametro “prop” rappresenta il modello di propagazione scelto per le onde radio che viaggiano da stazione in stazione che in questo caso corrisponde al modello TwoRayGround. Esso considera una doppia modalità di propagazione delle onde radio, sia direttamente attraverso l’etere sia per mezzo della riflessione attraverso la terra. In “netif” viene settata invece il tipo di interfaccia hardware utilizzata per l’accesso al canale. Successivamente infatti verranno settate le relative frequenze per interfaccia scelta. Il parametro “mac” rappresenta invece la tipologia dell’accesso al mezzo che nel nostro caso è quella utilizzata dall’802.11.
Vengono poi inizializzate le dimensioni dello scenario all’interno del quale avverranno le comunicazioni di dimensioni 3000m per 3000m (parametri “x” ed “y”) mentre all’interno della variabile “sc” viene caricato il path del file contenete lo scenario da caricare in cui verranno definite le posizioni di ciascuno dei nodi all’interno della rete. Infine vengono caricati in altri parametri il trace-file principale della simulazione contenente anche l’energia consumata dai dispositivi, la tipologia di antenna posseduta da ciascun nodo per poter ricevere e trasmettere i segnali radio e l’energia iniziale a disposizione di ciascun dispositivo espressa in Joules.

```
# Other default settings

LL set mindelay_        50us
LL set delay_                25us
LL set bandwidth_       0     ;# not used

Queue/DropTail/PriQueue set Prefer_Routing_Protocols    1

# unity gain, omni-directional antennas
# set up the antennas to be centered in the node and 1.5 meters
above it
Antenna/OmniAntenna set X_ 0
Antenna/OmniAntenna set Y_ 0
Antenna/OmniAntenna set Z_ 1.5
Antenna/OmniAntenna set Gt_ 1.0
Antenna/OmniAntenna set Gr_ 1.0

# Initialize the SharedMedia interface with parameters to make
# it work like the 914MHz Lucent WaveLAN DSSS radio interface
Phy/WirelessPhy set CPThresh_ 10.0
Phy/WirelessPhy set CSThresh_ 1.559e-11
Phy/WirelessPhy set RXThresh_ 3.652e-10
Phy/WirelessPhy set Rb_ 2*1e6
Phy/WirelessPhy set Pt_ 0.28183815
Phy/WirelessPhy set freq_ 914e+6
Phy/WirelessPhy set L_ 1.0
```

Nella sezione di codice precedente vengono settati alcuni valori relativi alle dimensioni dell'antenna posseduta dai nodi, atri valori relativi ad alcuni ritardi che vengono introdotti nell'accesso al Link Layer ed infine vengono settati una serie di parametri relativi alle frequenze di funzionamento dell'interfaccia hardware per l'accesso al canale.

```
proc finish {} {
      global ns_ quiet val nf
      $ns_ flush-trace
      set tracefd [open $val(tr) r]

      close $tracefd
      close $nf
      close $f0
      close $f1
      puts "finish.."
      exit 0
       }

# Initialize Global Variables

set ns_          [new Simulator]
set topo    [new Topography]

set tracefd     [open $val(tr)  w]
set f0 [open packets.tr w]
set f1 [open packets_risp.tr w]

$ns_ trace-all $tracefd
set nf [open bcastout.nam w]
        $ns_ namtrace-all-wireless $nf  $val(x) $val(y)
      set curr ""

$topo load_flatgrid $val(x) $val(y)

puts $tracefd "M 0.0 nn:$val(nn) x:$val(x) y:$val(y)
energy:$val(initialenergy)"
      puts $tracefd "M 0.0 sc:$val(sc) cp:$val(cp) seed:$val(seed)"
      puts $tracefd "M 0.0 prop:$val(prop) ant:$val(ant)"

set god_ [create-god $val(nn)]
set chan_1_ [new $val(chan)]
set chan_2_ [new $val(chan)]
```

All'interno della procedura "finish", la quale viene richiamata dall' interprete Tcl al termine della simulazione, vengono chiusi tutti i trace-file nei quali sono stati scritti i dati risultanti dalla simulazione appena terminata. Successivamente viene creata l'istanza del simulatore e vengono aperti tutti i trace-file ed infine creiamo l'oggetto "God" che si occupa di monitorare lo stato della rete ed istanziamo un nuovo canale di comunicazione che verrà poi associato a ciascun nodo della rete.

```
# configure node
$ns_ node-config -adhocRouting $val(rp) \
                 -llType $val(ll) \
                 -macType $val(mac) \
                 -ifqType $val(ifq) \
                 -ifqLen $val(ifqlen) \
                 -antType $val(ant) \
                 -propType $val(prop) \
                 -phyType $val(netif) \
             -channel $chan_1_ \
             -topoInstance $topo \
                 -energyModel $val(energymodel) \
                 -routerTrace OFF \
                 -macTrace OFF \
                 -agentTrace ON \
                 -energyTrace OFF \
                 -movementTrace OFF \
                 -rxPower 0.6 \          # Watt
                 -txPower 0.3 \
                 -idlePower 0 \
                 -initialEnergy $val(initialenergy)

for {set i 0} {$i < $val(nn) } {incr i} {

      set node_($i) [$ns_ node]
      $node_($i) random-motion 0    ;# disable random motion
      $node_($i) attach-dummy
      $ns_ initial_node_pos $node_($i) 250
      $node_($i) color red
      $ns_ at 0.01 "$node_($i) color red"
  }

for {set i 0} {$i < $val(nn)} {incr i} {

      set den_($i) [new Agent/Bcast_flood [$node_($i) id]]
      $node_($i) attach $den_($i) 900

            }

      Agent/Bcast_flood hello_period 20
      Agent/Bcast_flood hello_delay 5
      Agent/Bcast_flood hello_prob 0.6
      Agent/Bcast_flood hello_num 4


Agent/Bcast_flood instproc received {from packet} {
       global ns_ t f0
       $self instvar node_
       set id [$node_ id]
       puts " node  $id  received an hello packet from $from whose
id is $packet"
       puts $f0 " node  $id  received an hello packet from $from
whose id is $packet"
       global node_
       }

Agent/Bcast_flood instproc rejected {packet} {
       global ns_ t f1
```

```
        $self instvar node_
        set id [$node_ id]
        puts $f1 " node  $id  do not send an hello packet with
probability $packet"
        global node_
        }

proc outSendHello {me num} {
puts "$me sent his hello con id $num"}

proc prova {time rit} {
puts "Current time $time; Delay $rit"}

proc probability {prob} {
puts "Sent hello with probability $prob"}

proc number_packets {num} {
puts "Il numero di pacchetti da spedire è  $num"}

puts "Sto caricando lo scenario...."
source $val(sc)
puts "Load complete..."

#STOP
for {set i 0} {$i < $val(nn) } {incr i} {
        $ns_ at $val(stop).000000001 "$node_($i) reset";
}
 $ns_ at $val(stop).0001 "finish"
 $ns_ at 10 "$den_(0) startHello"
 #RUN
 $ns_ run
```

In questa ultima sezione di codice viene prima di tutto configurata l'entità nodo utilizzando i parametri definiti inizialmente, con l'aggiunta di altri parametri relativi ai dati che devono essere scritti nei trace-file riguardanti le azioni dei singoli nodi, e all'energia. Viene infatti settata l'energia consumata dal nodo ad ogni trasmissione e ricezione di un messaggio. L'energia consumata negli istanti di idle è stata appositamente posta a 0 Watt per meglio valutare l'energia consumata nelle trasmissioni. Tutti i parametri energetici in questo caso sono espressi in Watt. Vengono poi creati tutti i nodi della nostra simulazione a cui vengono attaccati gli agenti Bcast_flood e Dummy precedentemente illustrati disabilitandone il movimento casuale e settando la porta a cui ricevere i messaggi di broadcast al valore 900.
Successivamente vengono settati i valori delle quattro variabili membro dell'agente Bcast_flood. I valori di queste variabili potranno poi essere usati all'interno dei metodi dell'agente e se necessario modificati attraverso le procedure descritte nelle sezioni precedenti. I loro valori riguardano il ritardo utilizzato dallo scheduler per l'invio dei pacchetti, la soglia della probabilità che non deve essere superata per poter spedire un messaggio con successo e la dimensione della finestra dei messaggi da

trasmettere. Infine l'ultima parte è dedicata alle procedure che vengono invocate dal nostro agente all'interno dei propri metodi. La procedura "received" per esempio, invocata dal metodo recvHello(), a cui vengono passati la sorgente del pacchetto e l'identificativo del messaggio stesso non fa altro che trascrivere su un trace-file apposito che il nodo in questione ha ricevuto il messaggio con quel determinato numero di sequenza. La procedura "rejected" invece viene invocata dal metodo sendHello() del nostro agente e si occupa di trascrivere su di un trace-file apposito la mancata trasmissione del pacchetto poiché la probabilità che è stata generata è maggiore della soglia stabilita. Non resta altro infine che invocare il metodo "starthello" sul nodo trasmettitore, che nel nostro caso corrisponde al nodo 0. Più precisamente dopo 10 secondi dall'inzio della simulazione l'agente del nodo 0 invocherà il metodo sendHello() ed inizierà la trasmissione in broadcast a tutti i suoi vicini.

6.1.2 Risultati ottenuti dalle simulazioni

Analizziamo ora i dati estratti dalle simulazioni effettuate. Queste sono state suddivise in ulteriori sottocasi in base alla soglia relativa alla probabilità di trasmissione variabile tra 0,3 e 0,9. Questo significa per esempio che nel caso questa soglia sia pari a 0,5 vengono spediti solamente tutti quei messaggi con probabilità di trasmissione inferiore alla soglia. Dai dati ottenuti vengono poi estrapolati i valori delle grandezze che ci interessano. Nello scenario casuale le stazioni considerate all'interno della rete sono 500 (1 trasmettitore e 499 ricevitori) disposti appunto in maniera del tutto casuale (Figura 6.3). Inoltre ogni caso relativo ad una diversa soglia della probabilità di trasmissione viene iterativamente ripetuto per 1000 volte. Viene poi ricavata una media dei risultati ottenuti da ogni iterazione, tenendo conto dei relativi scarti quadratici per meglio analizzare la distribuzione dei valori.

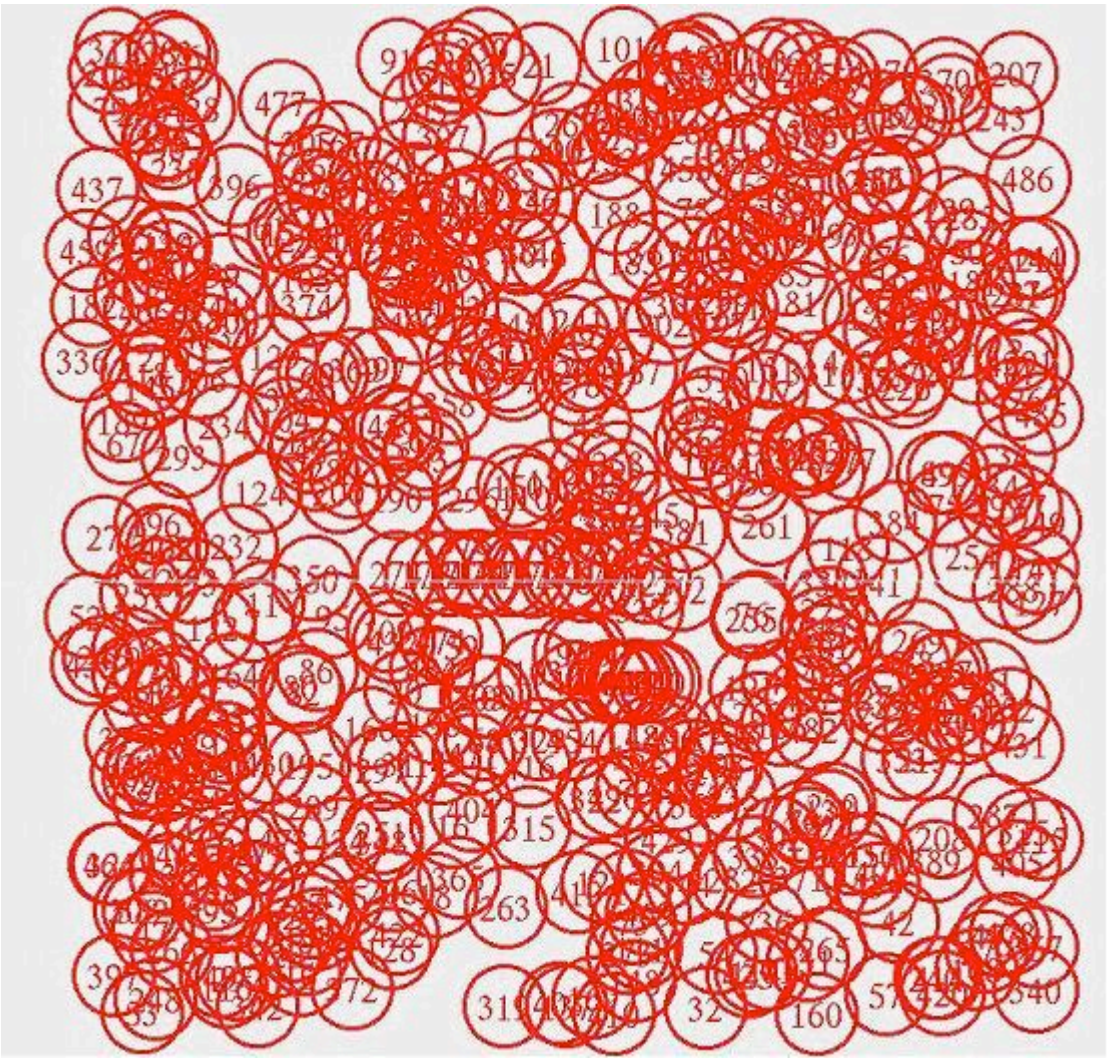

Figura 6.3 Lo scenario casuale

Analizziamo prima l'invio di 1 solo pacchetto da parte della stazione trasmettitrice in questo scenario di rete casuale. Riassumiamo i dati ottenuti nel grafico di Figura 6.4 dove nelle ordinate viene posta la media della percentuale(calcolata su 1000 iterazioni) dei nodi che hanno ricevuto il pacchetto e nelle ascisse tutti i valori assunti dalla soglia di probabilità della trasmissione in base alla quale sono state effettuate le simulazioni. Possiamo notare come con una probabilità pari a 0,5 si riescono a raggiungere l'80% dei nodi e già ad una probabilità di 0,7 il 100% dei nodi che ricevono il messaggio con un risparmio pari al 30% dei pacchetti

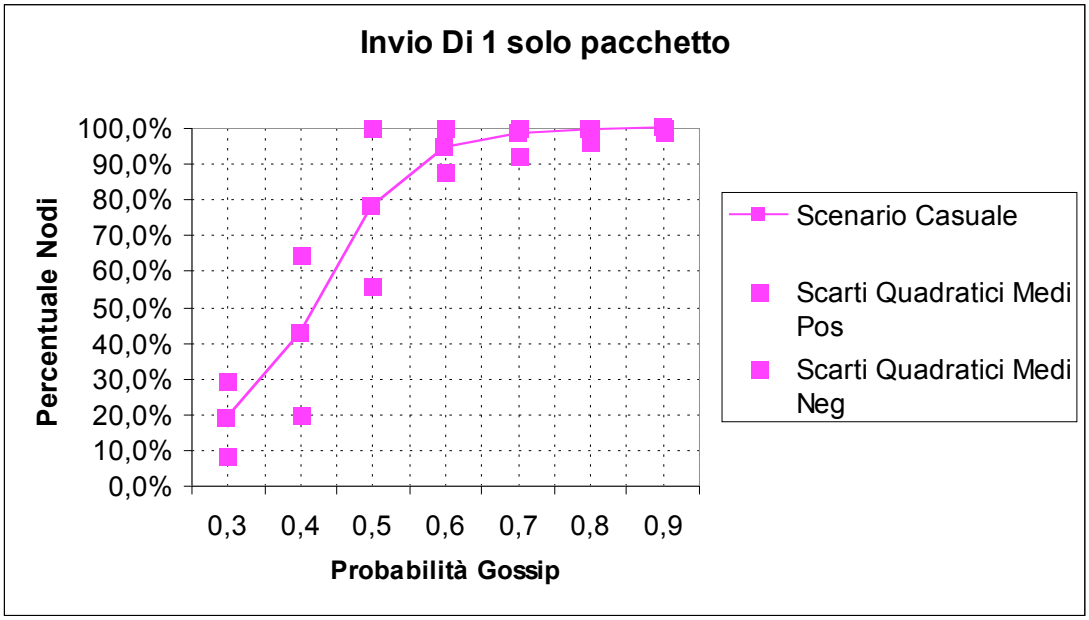

Figura 6.4 Media e Dispersione Percentuale dei nodi che ricevono il messaggio

Analizziamo ora la dispersione dei dati intorno alla media per meglio valutare l'affidabilità di questi dati nelle simulazioni che utilizzano lo scenari casuale precedentemente visto. Possiamo notare come questa sia all'incirca del 5-10% al massimo, eccezion fatta per quelle probabilità più basse dove la dispersione raggiunge il 20% circa. Se prendiamo per esempio il caso della probabilità 0,6 vediamo che ha uno scarto di circa il 10% e ci permette già di raggiungere quasi il 100% dei nodi in trasmissione.

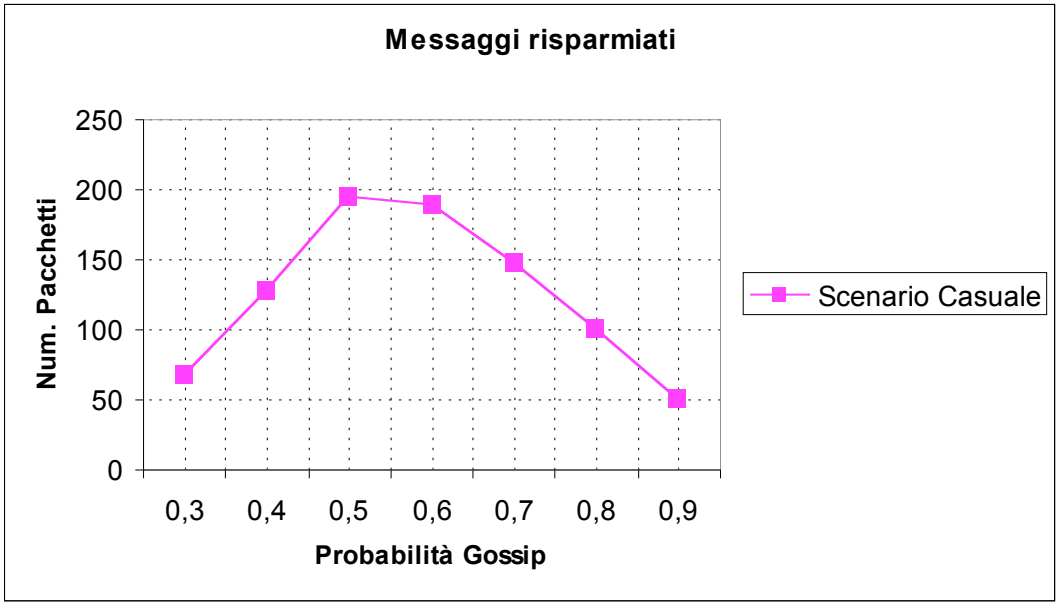

Figura 6.5 Scenario casuale: invio di 1 pacchetto

In Figura 6.5 possiamo osservare il numero di pacchetti risparmiati nel caso di invio di un singolo messaggio. Qui notiamo come per probabilità piccole il numero di messaggi risparmiati sia piccolo, poiché il numero di quelli spediti è decisamente basso. Per le probabilità intermedie come 0.4 o 0.5, si ha il massimo dei pacchetti risparmiati, poiché si trasmette un numero di messaggi abbastanza alto ma ci sono altrettanti pacchetti che non vengono spediti. Per le probabilità più alte invece essendo molto alto il numero di messaggi inviato, si riduce di conseguenza quello dei pacchetti risparmiati.
Passiamo ora ad analizzare il grafico di Figura 6.6 e Figura 6.7 dove vengono illustrati rispettivamente il consumo medio di energia per ogni dispositivo e il consumo di energia totale del sistema utilizzando come parametri della simulazione un consumo di energia in trasmissione pari a 0,3 mWatt ed in ricezione pari a 0,6 mWatt. L'energia finale consumata è espressa invece in Joules.

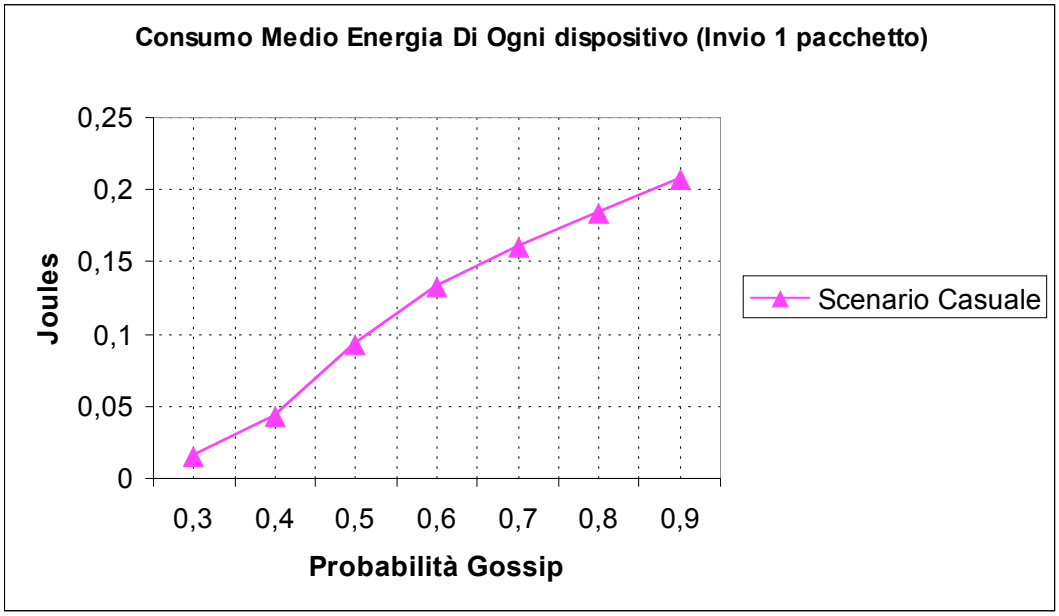

Figura 6.6 Scenario casuale: energia consumata dai dispositivi

Il risparmio di messaggi ottenuto nei vari casi della simulazione si ripercuote anche sull'energia mediamente consumata dai singoli dispositivi. Se consideriamo ad esempio il caso con probabilità 0,6, risparmiamo circa il 40% dell'energia consumata dai nostri dispositivi. Infine nella Figura 6.8 è possibile vedere il consumo medio complessivo di energia, da parte dell'interno sistema, il cui andamento della curva ovviamente seguirà quello del grafico relativo al consumo dei singoli dispositivi.

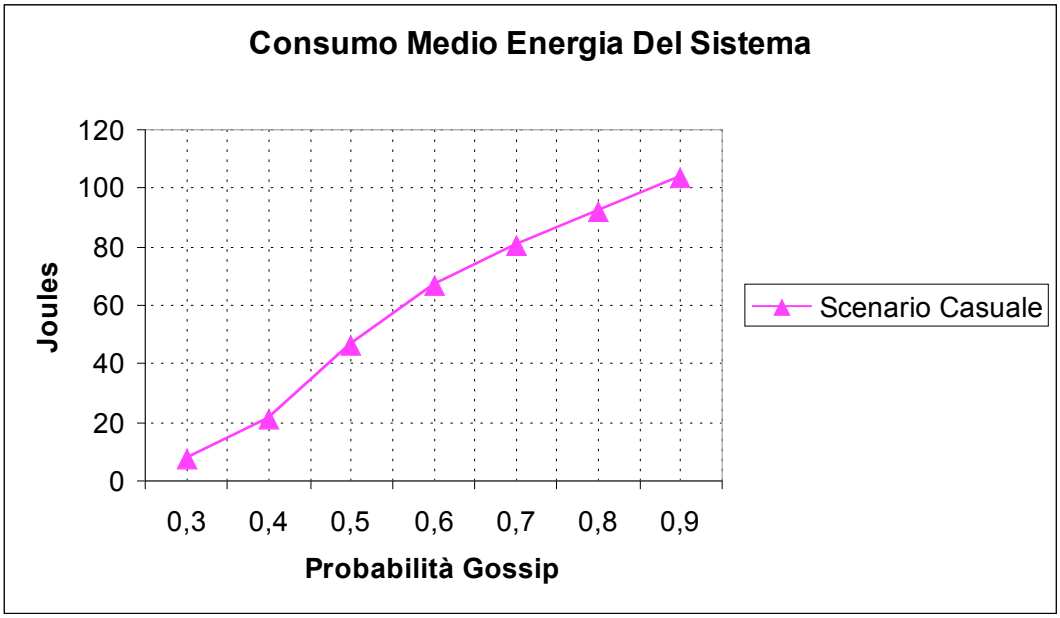

Figura 6.7 Scenario casuale: invio di un solo messaggio

Passiamo ora ad analizzare il comportamento del nostro protocollo di Gossip nel caso di trasmissione di una finestra di dimensioni variabili. Il nodo trasmettitore all'interno della rete invia questa finestra di dimensioni variabili (da 2 a 4 pacchetti) agli altri 499 nodi dello scenario e come nel caso di invio del singolo messaggio viene variata la soglia di probabilità di trasmissione in modo da analizzare il comportamento della rete. Per ogni caso sono stati estrapolati dai dati della simulazione la media percentuale dei nodi che hanno ricevuto, a seconda del caso considerato, almeno 1 pacchetto,almeno 2, almeno 3, e 4 pacchetti con le relative dispersioni ed il numero di pacchetti non inviati a causa della soglia di probabilità. Inoltre per quanto riguarda la finestra di dimensione 4 è stato analizzato anche il consumo di energia dei singolo dispositivi e quello totale della rete. Nella Figura 6.8 sono illustrate le percentuali dei nodi che ricevono almeno 1 messaggio nei 3 casi contraddistinti dalla differente dimensione della finestra di trasmissione, cioè 4,3,e 2 messaggi.

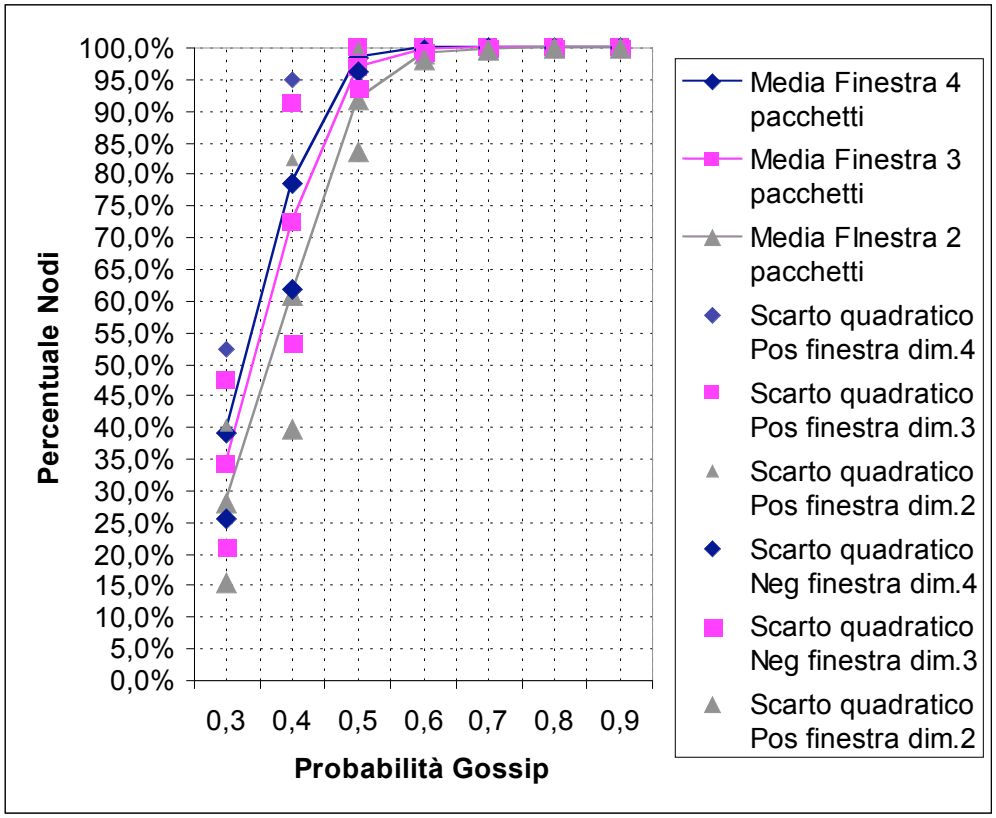

Figura 6.8 Media e Dispersione Percentuale dei nodi
che ricevono almeno 1 messaggio

Come possiamo vedere dal grafico precedente, già con una probabilità pari a 0,5 per qualsiasi dimensione della finestra di trasmissione la percentuale dei nodi che riusciamo a raggiungere è pari ad almeno il 90% dei nodi della rete. Infatti, nel caso della finestra di dimensioni 3 e 4 la percentuale dei nodi è quasi del 100%, mentre per quella di dimensione 2 la percentuale scende al 90% che come probabilità è ancora altissima visto che la soglia di probabilità è pari a 0,5. Nella Figura 6.8 possiamo inoltre analizzare la dispersione dei dati intorno alla media nelle 1000 iterazioni che sono state svolte per ogni singolo caso.
Nel grafico precedente, possiamo notare come tranne per la soglia di probabilità 0.4, dove vi è un picco di dispersione pari a circa il 20% dei nodi, dalla probabilità 0,5 in poi la dispersione è inferiore al 10%. Questi risultati sono significativi poiché ci assicurano con una forte sicurezza di poter raggiungere con almeno un messaggio, quasi il 100% dei nodi con un margine di variazione davvero piccolo già con probabilità non molto alte.
Nella Figura 6.9 possiamo invece osservare la percentuale di nodi che si riescono a raggiungere con almeno due messaggi. Come era facile prevedere c'è un abbassamento della curva dovuto all'aumento del

numero di messaggi da ricevere e quindi una riduzione delle percentuali ottenute rispetto alla ricezione di almeno un messaggio.
Per avere una percentuale di nodi che ricevono almeno due messaggi superiore al 90% del totale bisogna spostarsi verso la soglia di probabilità 0,6. Comunque già ad una probabilità di 0,5 i dati significativi, visto che nei casi in cui la finestra ci trasmissione è maggiore di 2 raggiungiamo almeno l'80% dei nodi, con un guadagno sui pacchetti risparmiati di circa il 50%.

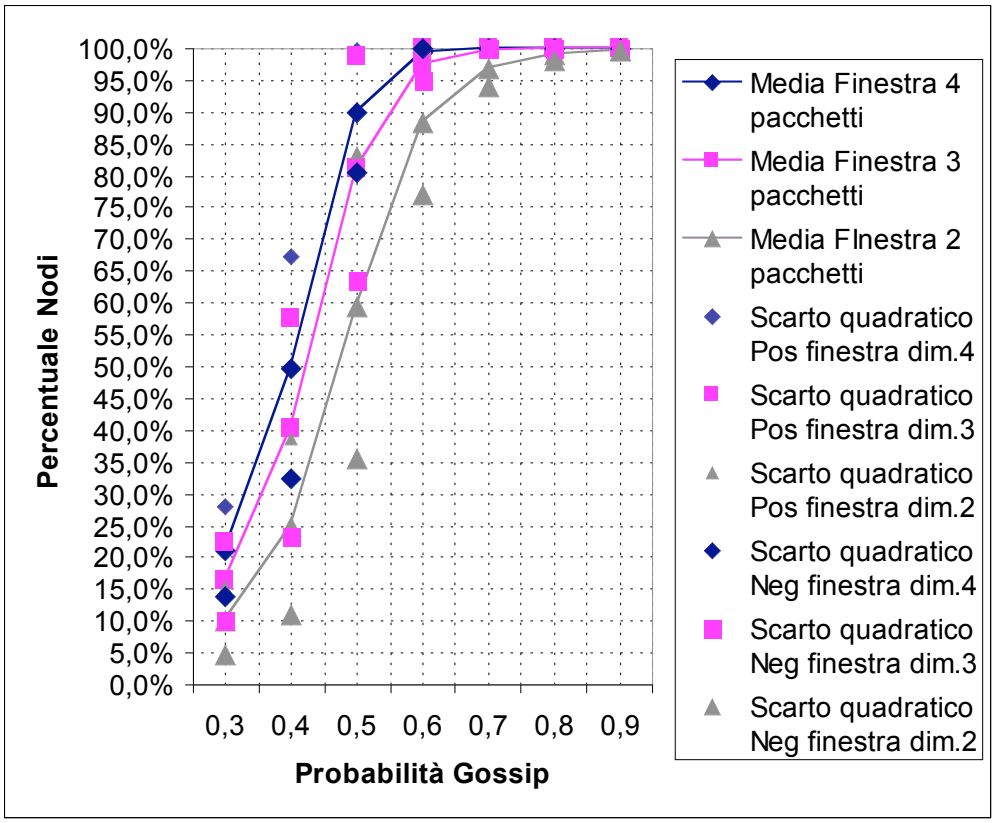

Figura 6.9 Media e Dispersione Percentuale dei nodi
che ricevono almeno 2 messaggi

Possiamo notare come il punto critico di dispersione dei nodi che hanno ricevuto almeno due pacchetti, trasla sempre più a destra, cioè verso probabilità più alte, man mano che la finestra di trasmissione si riduce (Figura 6.9). Questo è facilmente intuibile, visto che riducendosi la finestra diminuisce il numero di pacchetti che si possono ricevere e sui quali si può calcolare questa metrica. Per quanto riguarda le altre probabilità, la dispersione è discretamente bassa sempre al di sotto del 10% fornendoci un ottimo margine sulla percentuale di nodi raggiunta da almeno 2 messaggi.

Passiamo ora ad analizzare il caso della percentuale di nodi che ricevono almeno 3 messaggi. Ovviamente riguarderà solamente i casi relativi alle simulazioni con finestra di dimensioni 3 e 4. Possiamo commentare i dati ottenuti osservando i grafici di Figura 6.10.

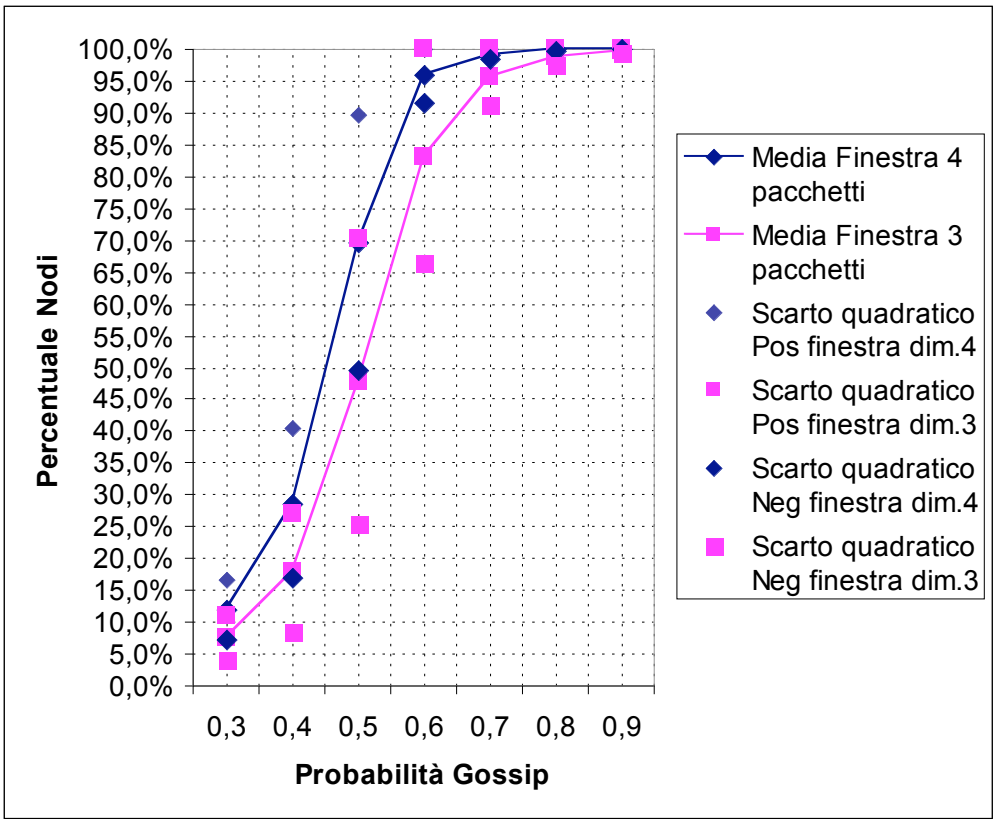

Figura 6.10 Media e Dispersione Percentuale dei nodi che ricevono almeno 3 messaggi

Possiamo notare come già alla probabilità 0,6 la percentuale di nodi raggiunta da almeno 3 pacchetti sia superiore all'80% sia nel caso di invio di una finestra di dimensione 4 sia nel caso di dimensione 3. Per avvicinarsi ad una copertura del 100% dobbiamo salire alla probabilità 0,7 che presenta una dispersione inferiore al 5%. Questo ci assicura un ottimo margine sui nodi che ricevono almeno 3 pacchetti. Passiamo ora al caso della ricezione di 4 pacchetti, in cui ovviamente viene analizzata solamente la trasmissione della finestra di dimensione 4. Come è possibile notare in Figura 6.11 la curva si abbassa notevolmente rispetto ai casi precedentemente analizzati, infatti aumenta il numero di messaggi da ricevere e quindi vengono considerati solo quei nodi che ricevono tutti i pacchetti spediti dalla stazione trasmettitrice.

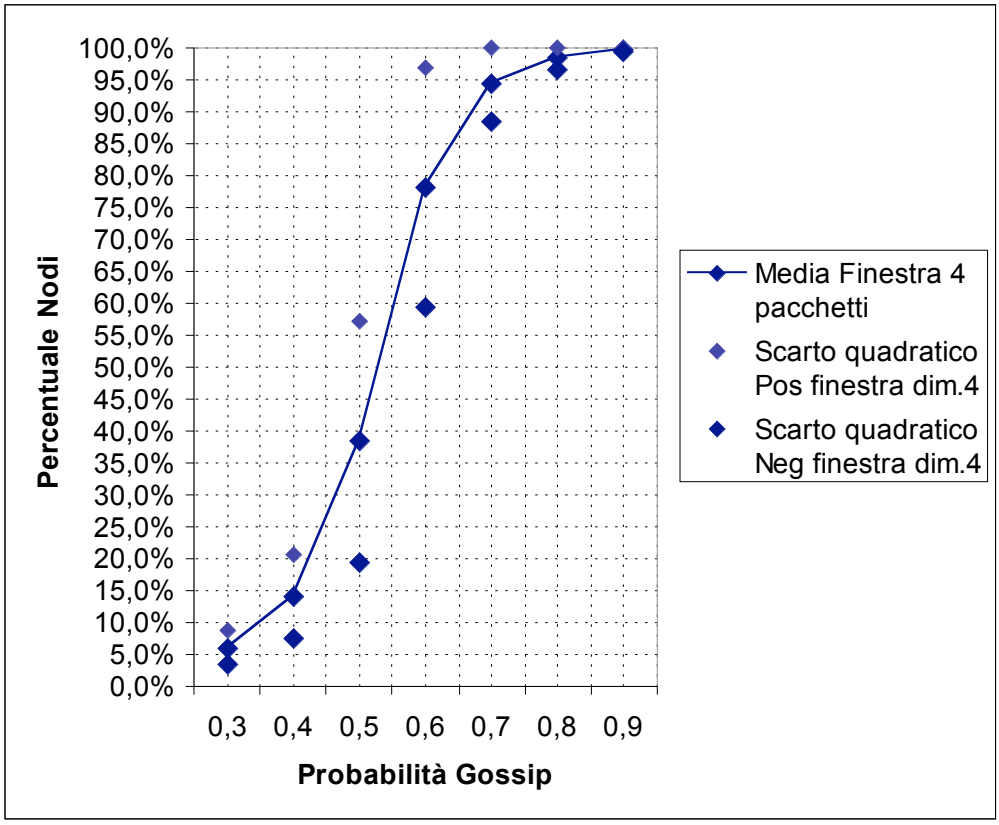

Figura 6.11 Media e Dispersione Percentuale dei nodi che ricevono 4 messaggi

Infatti mentre nella ricezione di almeno 3 pacchetti, la curva riguardante la finestra di dimensione 4 in corrispondenza della probabilità 0,6 si trovava oltre il 90%, adesso per la soglia 0,6 è di poco sotto all'80%. Rimane comunque dei dati estremamente significativi, infatti a partire da una soglia di 0,7 la percentuale di nodi che riceve tutti i 4 messaggi spediti dalla stazione trasmettitrice sono oltre il 90% con una dispersione intorno alla media di poco superiore al 5% (Figura 6.11).
Possiamo inoltre osservare il numero di pacchetti risparmiati in tutti i casi precedentemente analizzati relativi alle 3 diverse finestre di trasmissione (Figura 6.12). L'andamento dei grafici rispecchierà molto quello del caso di trasmissione del singolo pacchetto, ovviamente con valori molto più alti come conseguenza del maggior numero di pacchetti trasmessi.

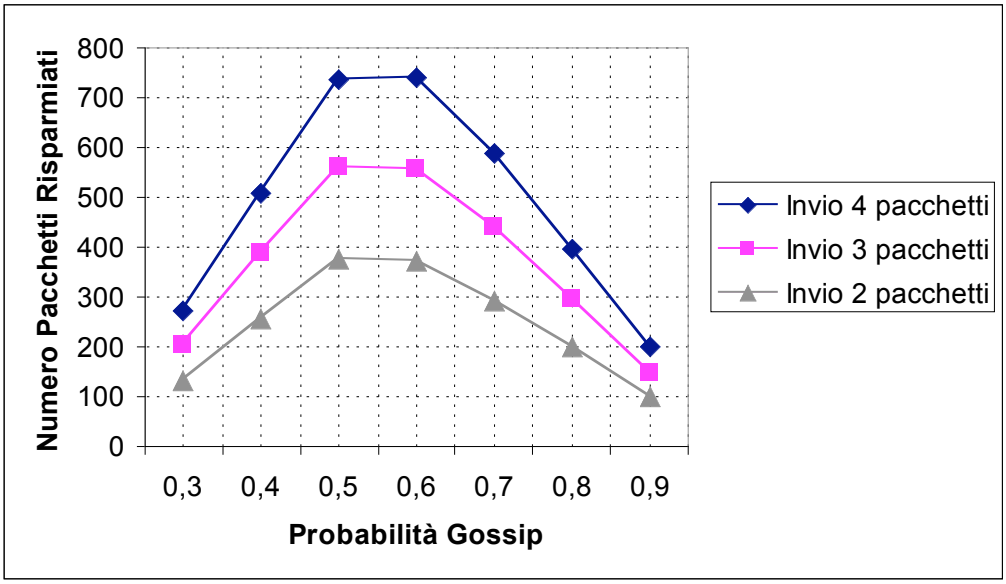

Figura 6.12 Messaggi risparmiati con finestra variabile

Per quanto riguarda la simulazione dei casi relativi alla finestra di dimensione 4 c'è un ulteriore metrica da analizzare e cioè l'energia consumata dai dispositivi nelle loro trasmissioni (Figura 6.13 e Figura 6.14). Per quanto riguarda lo scenario casuale i dati ottenuti mostrano, similmente all'invio di 1 solo pacchetto un andamento quasi lineare dell'energia. Anche nel caso di invio della finestra di 4 pacchetti il risparmio di messaggi inviati e conseguentemente di energia è stato confermato.

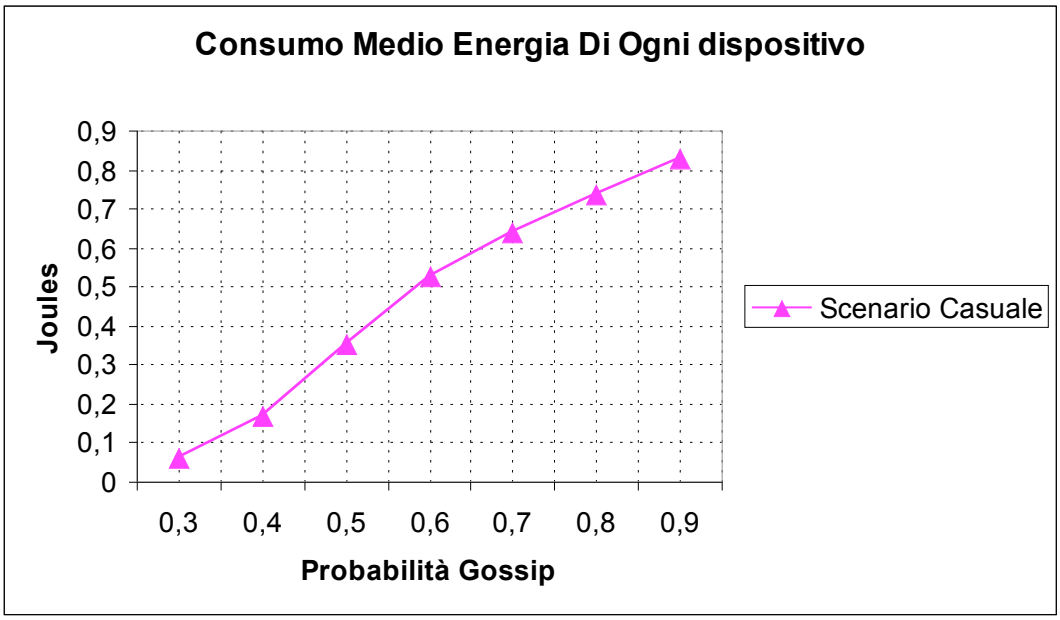

Figura 6.13 Scenario casuale: invio finestra di 4 pacchetti

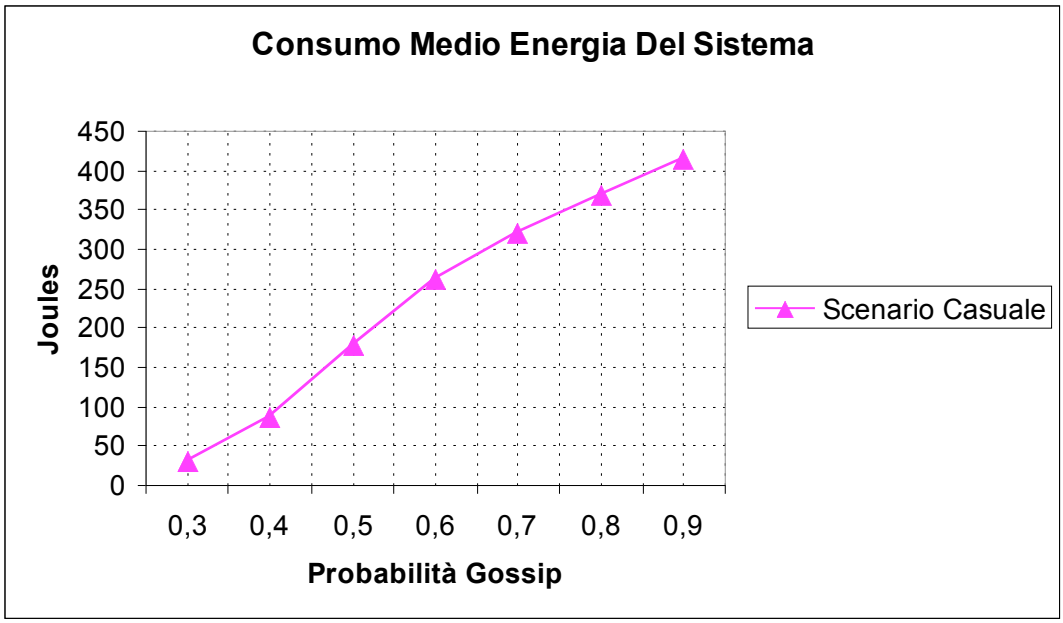

Figura 6.14 Scenario Casuale: invio finestra di 4 pacchetti

Le prestazioni del protocollo di Gossip sviluppato, in tipologie di reti casuali come quella utilizzata nelle nostre simulazioni, ci permettono di affermare che queste sono senza dubbio migliori delle prestazioni di un generico protocollo di flooding utilizzato negli stessi scenari. Infatti come abbiamo potuto notare dai dati estrapolati dalle simulazioni la percentuale di nodi che si riesce a raggiungere grazie all'uso del Gossip è davvero molto alta. Prendiamo per esempio il caso dei nodi che ricevono almeno 1 pacchetto (Figura 6.9) nella trasmissione della finestra di messaggi variabile; possiamo notare come già per probabilità di gossip pari a 0,5 si riesca a raggiungere con almeno 1 pacchetto quasi il 100% dei nodi risparmiando ben il 50% dei pacchetti inviati rispetto ad un generico flooding. Risparmiando sui messaggi trasmessi di conseguenza si effettua un corrispondente risparmio anche nel consumo di energia dei singoli dispositivi. Il protocollo di Gossip ci permette quindi rispetto ad un generico flooding, di minimizzare il numero di messaggi necessari per poter raggiungere tutti i nodi della rete riducendo quindi il consumo di energia necessaria a far comunicare i dispositivi per tutta la durata delle trasmissioni.

Conclusioni

La crescente proliferazione dei dispositivi portabili e l'emergenza delle tecnologie MANETs richiedono lo studio di soluzioni innovative che consentano la disseminazione delle informazioni.
In questa tesi ci siamo focalizzati sullo studio dei protocolli di Gossip. In particolare, è stata presentata una analisi delle caratteristiche del Gossip(p,0). La nostra analisi è stata focalizzata da un lato sulla determinazione della efficacia nella disseminazione di messaggi garantita del protocollo e, dall'altro, dalla valutazione del costo di questa operazione. In particolare, abbiamo perciò determinato la frazione dei nodi della rete che ricevono un messaggio disseminato, il numero di trasmissioni ed il degrado delle batterie dei dispositivi che la disseminazione di un messaggio comporta. Inoltre, al fine di effettuare una valutazione delle caratteristiche del protcollo flooding prob studiato, abbiamo effettuato confronti delle metriche individuate rispetto al protocollo di flooding.
Dalle nostre analisi è emerso che i protocolli di Gossip possono essere proficuamente impiegati per la disseminazione di informazioni in scenari MANET. In particolare, questi protocolli risultano particolarmente adatti a tutti gli scenari applicativi in cui è sufficiente una disseminazione di informazioni di tipo best effort. I dati sperimentali ottenuti mostrano inoltre come i protocolli di Gossip possano ridurre notevolmente l'overhead di comunicazione ed il degrado delle batterie dei dispositivi se confrontati con protocolli tradizionali quali il flooding.
Questi primi risultati possono stimolare ulteriori attività di ricerca al fine di confrontare i risultati ottenuti tramite diversi protocolli di Gossip. Inoltre il confronto fra protocolli potrebbe essere esteso ad includere protocolli di multicast quali MAODV o FGMP.

Bibliografia

[Kev03] Kevin Fall, Kannan Varadhan: “The ns Manual”, 2003.

[Mar] Marc Greis: “Tutorial for the UCB/LBNL/VINT Network Simulator ns"

[Bal] Roberto Baldoni, Ravi Prakash: “Architecture for Group Communication in Mobile Systems”

[Ami03] Amitava Mukherjee, Somprakash Bandyopadhyay, Debashis Saha: “Location Manangement and Routing in Mobile Wireless Networks”, 2003

[Moh03] Mohammad Ilyas: “The Handbook of Ad Hoc Wireless Networks”, 2003

[Dav] David Kempe, Jon Kleinberg, Alan Demers: “Spatial Gossip and Resource Location Protocols”

[And03] Andrea Zuccherelli: “Middleware per Applicazioni Peer-to-Peer in Ambito Mobile Ad Hoc Network”, 2003