UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA TRIENNALE IN INFORMATICA

ANALISI DI APPROCCI DI STOP CONDITION PER ALGORITMI EPIDEMICI USATI IN RETI OPPORTUNISTICHE

Relatore: Prof. Gian Paolo ROSSI
Correlatore: Prof.sa Elena PAGANI
Prof. Federico PEDERSINI

Tesi di Laurea di:
Francesco ALBERTI
Matricola 676857

Anno Accademico 2006–07

Indice

Capitolo 1

Introduzione

Oggigiorno sempre più utenti hanno la possibilità di comunicare fra loro via Internet tramite reti wireless.

L'utilizzo di reti wireless rispetto alle reti cablate gode di due grandi vantaggi: i costi di installazione e manutenzione sono inferiori e gli utenti possono muoversi all'interno dell'ambiente raggiunto dalla rete wireless.

Lo standard IEEE 802.11 (noto più comunemente come WiFi *(Wireless Fidelity)* supporta due tipi di mezzi di trasmissione wireless: le onde radio e i segnali ottici all'infrarosso. Le onde radio vengono trasmesse nella banda ISM (Industrial Scientific Medical) attorno ai $2,45$ GHz, dal momento che questa frequenza è disponibile in tutto il mondo ed il suo impiego non richiede alcuna licenza.

Lo standard definisce due modalità operative alternative: una che usa una stazione radio di base - che appartiene ad un'unità chiamata *access point* - e l'altra senza punto d'accesso. Nel primo caso si parla di LAN wireless d'infrastruttura, mentre nel secondo di LAN wireless *ad hoc*.

La principale applicazione delle LAN wireless è la sostituzione del cablaggio fisso: uno dei motivi per i quali si sono sviluppate le LAN wireless è la grande diffusione di unità portatili - PDA, PC tascabili, laptop, notebook, ecc. Ogni utente può utilizzare questi dispositivi per accedere ad Internet grazie ad una LAN wireless installata da provider pubblici di telecomunicazioni nei terminal degli aeroporti, stazioni ferroviarie, grandi magazzini, ecc.

Vi sono però alcune applicazioni che beneficiano dell'utilizzo di reti ad hoc: la costituzione di team in condizioni d'emergenza o in situazioni di guerra, la necessità di una continua interazione tra l'utente e l'ambiente circostante per il recupero delle informazioni, l'utilizzo in ambienti veicolari per il monitoraggio del traffico, la notifica di incidenti o la ricerca di soccorso.

In queste situazioni l'istallazione di reti cablate o wireless con infrastruttura è insostenibile per quanto riguarda costi e/o tempi.

1.1 DTN

Internet ha avuto (e sta tutt'ora avendo) un grande successo in tutto il mondo in quanto permette una connettività globale attraverso l'inter-connessione di più sottoreti di comunicazione, offrendo collegamenti punto a punto continui e bidirezionali. Anche se spesso non è esplicitato, Internet opera grazie ad alcune assunzioni: esiste un collegamento *end-to-end* tra la sorgente e la/le destinazione/i del traffico, il *round trip time* tra due nodi qualsiasi dell'inter-rete non è eccessivo e la probabilità che un pacchetto venga "droppato" durante il suo invio è molto bassa. Non a caso, la connettività di Internet è basata su collegamenti fisici[1] che interconnettono dispositivi chiamati *router* deputati all'instradamento (*routing*) dei pacchetti. Questa configurazione sostiene a pieno le tre assunzioni appena esposte.

Oltre ad Internet, altre categorie di reti differenti stanno diventando sempre più importanti [5]. Queste tipologie di reti non si basano sulle assunzioni supposte da Internet: il modello TCP/IP non può funzionare con esse. Esempi di queste reti sono:

Reti mobili terrestri: alcune di queste reti potrebbero partizionarsi improvvisamente a causa dell'elevata mobilità dei nodi o del cambiamento della potenza del segnale, mentre altre potrebbero partizionarsi in maniera predicibile. Ad esempio un autobus cittadino potrebbe essere utilizzato come switch "store and forward" a range di comunicazione limitato: quando passa da un posto all'altro potrebbe

[1] Caratterizzati da un *error rate* basso e da un *bit rate* sempre più elevato grazie all'avvento delle fibre ottiche.

offrire un servizio di comunicazione ai suoi attuali vicini con altri dispositivi più distanti che raggiungerà in futuro.

Military Ad-Hoc Networks: questi sistemi dovrebbero lavorare in ambienti ostili, dove mobilità, fattori ambientali, o spegnimenti intenzionali dei dispositivi potrebbero essere causa di disconnessioni. Inoltre, i dati che vengono trasmessi su queste reti potrebbero dover attendere del tempo prima di essere trasmessi per permettere la trasmissione di traffico con maggior priorità. Ad esempio, il traffico di dati potrebbe dover attendere alcuni secondi prima di essere trasmesso a causa dell'occupazione del canale da parte di traffico audio.

Reti di sensori: gli apparati che compongono queste reti sono caratterizzati da potenza di calcolo bassa e memoria ridotta, oltre ad una durata limitata delle batterie di alimentazione. Le comunicazioni tra questi apparecchi sono spesso schedulate per poter risparmiare batteria, e vi sono spesso dei nodi collegati alla rete ai quali vengono inviate le informazioni per poterle immettere nella rete.

Comunicazioni interplanetarie: questo tipo di comunicazione soffre di lunghe latenze, spesso misurate in ore o addirittura giorni!

Una DTN, acronimo di *Delay Tolerant Network*, è [5] una tipologia di rete progettata per poter operare efficientemente nonostante lunghi periodi di latenza. Le DTN richiedono dell'hardware necessario ad archiviare una gran quantità di dati. Queste attrezzature hardware devono essere capaci di mantenere le informazioni nonostante periodi prolungati di mancanza di energia e improvvisi riavvii del sistema. Dovrebbero anche essere accessibili in breve tempo in qualsiasi momento. La tecnologia ideale per questo fine presuppone dischi rigidi e grandi volumi di memorie flash. I dati archiviati dovrebbero essere ordinati per priorità da software in grado di garantire funzionalità accurate e affidabili di archiviazione e inoltro.

Nelle DTN vengono usati dispositivi di archiviazione poiché

- il link su cui inoltrare potrebbe non essere disponibile per un tempo indefinito;
- in una comunicazione fra due nodi uno potrebbe inviare o ricevere dati più velocemente o con meno errori dell'altro;
- un messaggio, una volta trasmesso, potrebbe dover essere ritrasmesso in caso di

errori dovuti al collegamento (frequenti sui link wireless) oppure in caso che il destinatario rifiuti il messaggio inoltrato.

1.2 MANET

Un'altra tipologia di rete wireless è composta dalle cosidette *MANET*, acronimo di *Mobile Ad hoc Network*. Anche in questa tipologia di rete i nodi, che godono di alta mobilità, sono connessi tra loro mediante collegamenti wireless instaurati grazie a reti ad hoc costituite nei momenti di contatto tra due o più dispositivi. A causa dell'alta mobilità dei nodi la topologia della rete può cambiare drasticamente in ogni momento.

Ogni nodo riesce a sapere quando ha altri nodi nel suo range di comunicazione poiché ognuno di essi trasmette in broadcast ad intervalli regolari b un pacchetto chiamato *beacon*, contenente l'identificativo del mittente. Ogni nodo mantiene una tabella dei vicini aggiornata tramite beaconing; ogni entry della tabella dei vicini ha associato un timer pari a $k \cdot b$, con $k > 2$. Quando viene ricevuto un beacon, il nodo controlla se l'identificativo contenuto nel pacchetto è già presente nella tabella. In caso positivo viene fatto ripartire il timer; in caso negativo viene creata la entry con l'identificativo del vicino e viene fatto partire il timer. Quando un timer scade viene rimossa la entry associata dalla tabella.

Una caratteristica fondamentale delle MANET è rappresentata dalla loro capacità di autocostituirsi ed autoconfigurarsi attraverso beaconing. Grazie a questa loro caratteristica esse possono essere utilizzate negli ambienti sopra descritti di emergenza, dove l'assenza di un'infrastruttura fissa comprometterebbe la connettività.

La consegna dei messaggi in queste reti è effettuata adottando un approccio *store and forward*, simile al meccanismo adottato dalle DTN e dai router IP. La differenza sostanziale risiede nel fatto che i router IP salvano per una frazione piccolissima di tempo il pacchetto da inoltrare, nelle DTN e nelle MANET il pacchetto viene salvato all'interno dei nodi. Nelle DTN il pacchetto viene salvato all'interno dei nodi perché potrebbe non essere disponibile alcun vicino al quale inoltrarlo, mentre nelle MANET viene eseguito il caching dell'informazione perché ogni nodo, sfruttando la mobilità, possa ritrasmettere il pacchetto in un secondo momento in una nuova posizione.

Un'altra differenza tra l'inoltro da parte dei router IP e l'inoltro effettuato dai nodi di una MANET è determinata dalla modalità di trasmissione: i nodi di una MANET trasmettono in broadcast il pacchetto, e quindi esso viene ricevuto da tutti i nodi nel range di comunicazione, al contrario di un router IP che esegue l'inoltro su un unico link fisico.

Nelle MANET gli algoritmi di routing tradizionali utilizzati da IP per inoltrare i pacchetti non vanno bene: la rete ha una topologia variabile e non si hanno informazioni di instradamento.

1.2.1 Reti opportunistiche

l mio lavoro tratterà protocolli di rete utilizzati all'interno di reti opportunistiche.

Esse possono essere definite [3] come reti occasionalmente connesse soggette a frequenti partizionamenti e che possono comprendere più di un insieme di protocolli - o famiglie di protocolli - divergenti fra loro. Una rete opportunistica può essere anche una rete costruita "al volo" da piccoli apparati wireless[2] sempre più diffusi nelle tasche delle persone al fine di reperire informazioni di vario genere.

Dato che gli apparati che costituiscono queste tipologie di reti hanno risorse limitate, bisogna definire un protocollo che allochi la memoria strettamente necessaria al suo funzionamento e che sia in grado di diffondere il messaggio all'interno della rete utilizzando il minor numero possibile di pacchetti inviati. La potenza del segnale wireless dovrà essere mantenuta a livelli piuttosto bassi per non sprecare batteria inutilmente, ciò minimizza quindi il raggio di comunicazione degli apparati aumentando conseguentemente la probabilità di partizionare la rete stessa; i nodi inoltre potrebbero ciclicamente spegnersi e riaccendersi per cercare di mantenere il più alto possibile il tempo di autonomia.

[2] Ad esempio palmari, smartphone e cellulari evoluti.

1.3 Algoritmi epidemici

Ovviamente le MANET non sono reti adatte al trasferimento di traffico real time, ma potrebbero essere utilizzate per la diffusione di feed, news, email, dati ottenuti da sensori distribuiti in un ambiente, ...

Gli approcci tipici che sono stati finora utilizzati all'interno di reti delay tolerant per il trasferimento di questa tipologia di dati sono principalmente basati sulla ridondanza: una stessa copia del dato da inviare viene instradata lungo cammini diversi per aumentare la probabilità che essi giungano a destinazione. I vari algoritmi di instradamento esistenti basati su questo tipo di approccio possono essere classificati in quattro categorie principali [9] in base al numero di copie dello stesso dato prodotte e al metodo utilizzato per decidere verso quale direzione instradare:

Flooding: ogni nodo inoltra tutti i messaggi non duplicati - sia quelli da lui generati che quelli ricevuti da altri nodi - verso tutti i nodi che ha in range.

Direct contact: il nodo sorgente tiene in memoria i dati finché non giunge in contatto con la destinazione.

Simple replication: *r* copie identiche del messaggio vengono inviate ai primi *r* nodi scelti a caso tra i vicini. Solo la sorgente invia copie multiple dei messaggi, i nodi "ponte" possono trasmettere solo alla destinazione finale. Algoritmi di questo tipo sono conosciuti con il nome di *two-hop relay*.

History-based: è simile al precedente ma si utilizza la storia passata come indicatore nella scelta dei vicini a cui inoltrare. Ogni nodo tiene traccia della probabilità che ogni altro host ha di consegnare il suo messaggio alla destinazione e quindi sceglie gli *r* vicini di grado più alto ed invia a loro i dati.

Non vi è un algoritmo ottimale: ognuno presenta dei trade-off differenti tra latenza di consegna del messaggio e overhead delle risorse di rete e di sistema. Il flooding, ad esempio, minimizza la latenza, ma sfrutta al massimo le risorse di rete e di sistema. Al contrario, con direct contact si ottiene il minimo overhead, ma il messaggio potrebbe subire una latenza enorme.

Algoritmo	**chi**	**quando**	**verso chi**
Flooding	tutti i nodi	nuovo contatto	tutti i nuovi
Direct contact	solo la sorgente	destinazione	solo la destinazione
Simple replication	solo la sorgente	nuovo contatto	*r* primi contatti
History-based	tutti i nodi	nuovo contatto	*r* di grado più alto

Tabella 1.1: Vari algoritmi di instradamento

Tutti gli algoritmi che rientrano in una di queste quattro categorie fanno parte di un approccio ancora più generale definito di *gossiping* o *epidemic*. La differenza tra i due è molto lieve, quasi inesistente, qui ci riferiremo indistintamente all'uno o all'altro. La definizione di approccio epidemico parte dal problema di avere un nodo che possiede un messaggio che deve far arrivare a tutti gli altri nodi della rete il più velocemente possibile, per fare ciò ognuno di essi sfrutta solo interazioni locali tra se stesso ed i suoi vicini, proprio come accade in un'epidemia. Ogni nodo inoltra il messaggio ai suoi vicini o ad un sottoinsieme di essi, iterando questa operazione hop per hop fino a raggiungere i confini della rete o un sottoinsieme prefissato di essa. Gli algoritmi epidemici mirano a raggiungere il loro obiettivo di diffusione anche in reti - come le opportunistiche - dove si hanno poche o nessuna informazione circa la topologia, mantenendo ridotto il carico di dati a patto di tollerare un aumento della latenza.

Di varianti ne esistono parecchie: un nodo potrebbe inviare i messaggi in broadcast oppure optare per una comunicazione unicast, si potrebbe decidere il vicino a cui inviare in base ad un history o in base ad altre scelte prese al momento, o ancora si può mandare il messaggio in broadcast ogni *x* secondi, si potrebbe scegliere il "verso" della trasmissione: *pull-based* se il nodo iniziatore della comunicazione chiede ai suoi vicini se possiedono nuovi messaggi per lui; *push-based* se è invece il nodo iniziatore ad inviare ai propri vicini i nuovi messaggi che ha ricevuto; *pull-push* se il nodo iniziatore invia i propri messaggi e attende che i vicini gli mandino in risposta i loro. Queste scelte partecipano a determinare il trade-off tra latenza di consegna e overhead sulle risorse di sistema e di rete.

Approcci epidemici sono già stati sfruttati in letteratura per risolvere diverse tipologie di problemi; una di queste è, ad esempio, l'allineamento di database. Allo *Xerox*

Palo Alto Research Center sono stati effettuati degli studi [4] per valutare se l'adozione di algoritmi epidemici potesse risolvere il problema di mantenere la consistenza tra i contenuti dei vari siti di un database replicato, a fronte di aggiornamenti che venivano effettuati su un singolo sito e dovevano essere propagati anche a tutti gli altri. I fattori che dovevano essere tenuti in considerazione erano proprio la latenza con cui l'aggiornamento veniva propagato a tutti i siti della rete e la quantità di traffico che veniva generata dalla propagazione di ogni singolo aggiornamento.

Il primissimo metodo utilizzato per diffondere gli aggiornamenti era del tipo *direct mail*: ogni nuovo aggiornamento veniva immediatamente inviato dal sito dove era stato applicato a tutti gli altri siti della rete. Ciò si rivelava sufficientemente efficiente sia in termini di latenza che di sovraccarico di rete ma non sempre risultava affidabile; poteva infatti accadere che dei messaggi si perdessero durante il tragitto o che alcuni siti non inviassero l'aggiornamento ad altri in quanto non erano a conoscenza della loro esistenza. Questo metodo è stato confrontato tramite alcune simulazioni con altri due algoritmi epidemici: *anti-entropy* e *rumor*, algoritmi che possiedono caratteristiche sostanzialmente diverse tra loro.

Le simulazioni effettuate hanno provato come, affiancando all'algoritmo direct-mail un algoritmo anti-entropy periodico e impostando adeguatamente i parametri di periodicità e ridondanza di quest'ultimo, le performance del sistema crescano significativamente riducendo in modo consistente il carico di traffico sull'intera rete. L'algoritmo rumor, inoltre, si è dimostrato un buon candidato a rimpiazzare, in futuro, l'algoritmo direct-mail ed, affiancato ad anti-entropy, aumentare ulteriormente le prestazioni del sistema.

Un approccio epidemico potrebbe essere applicato anche nella risoluzione di problematiche relative al *data aggregation* all'interno di reti di sensori: tramite l'utilizzo di un algoritmo anti-entropy pull-push si possono far cooperare i sensori al fine si calcolare un risultato comune. L'approccio anti-entropy potrebbe essere utilizzato anche per la ricerca di risorse o la distribuzione di informazioni in reti delay tolerant con frequente cambiamento della topologia e basso livello di connettività oppure, sempre in queste reti, si possono adottare approcci epidemici per effettuare il routing dei dati tra sorgente e destinazione in movimento.

Il lavoro di questa tesi mira a progettare un protocollo di routing da utilizzare nelle reti opportunistiche per diffondere messaggi da una sorgente a tutti i dispositivi dell'area. La progettazione del protocollo mirerà a trovare un corretto trade-off tra consumo di risorse di sistema e di rete e nodi raggiunti.

Capitolo 2

Approccio anti-entropy

I protocolli che operano all'interno di reti opportunistiche non possono basare le scelte di instradamento su informazioni distribuite quali il numero di nodi che hanno già l'informazione da distribuire, la direzione da far seguire al pacchetto per farlo giungere a destinazione, ecc..., ma devono basare le loro scelte su informazioni locali. Queste informazioni possono essere di qualsiasi tipo: il numero di vicini incontrati, il numero delle volte che è stato incontrato un vicino, il numero di duplicati ricevuti, ecc...

Assume quindi fondamentale importanza, nell'ambito di questa tipologia di reti, sviluppare un metodo efficace ed efficiente per la distribuzione di informazioni di vario genere all'interno della rete. Il protocollo utilizzato a tale scopo dovrà riuscire a distribuire i propri messaggi al maggior numero di nodi - eventualmente a tutti - all'interno della rete mantenendo, al tempo stesso, latenza e overhead a livelli accettabili.

Le MANET sono spesso e volentieri composte da apparati wireless stile PDA, cellulari evoluti o simili che supportano lo standard WiFi. Questi apparecchi portatili si stanno diffondendo molto in quanto permettono l'accesso ad Internet in aree dove è stata installata una rete wireless. Di contro però hanno il problema che la loro batteria e la loro memoria sono limitate. Un algoritmo che esegue flooding o che alloca una grande quantità di memoria non offrirebbe delle performance soddisfacenti: gli apparecchi si scaricherebbero in poco tempo oppure diventerebbero inutilizzabili.

Le ricerche fatte finora in letteratura riguardanti gli approcci epidemici, suggeriscono che un protocollo basato su algoritmi di questo tipo potrebbe fornire una soluzione

adeguata al problema. Essi infatti, come già visto in precedenza, mirano a distribuire messaggi all'interno della rete il più velocemente possibile cercando un trade-off tra overhead e latenza; tali problematiche sono proprio quelle che si devono affrontare per la distribuzione di informazioni all'interno delle MANET.

L'approccio anti-entropy è già stato utilizzato precedentemente in letteratura [8] per risolvere con successo problematiche di routing in reti ad hoc parzialmente connesse. Per garantire la consegna dei dati tale approccio si basa sull'assunzione che, grazie alla mobilità dei nodi, coppie di host entrino in range di comunicazione periodicamente e casualmente; tale casualità fa sí che il processo di distribuzione non segua un cammino particolare e che quindi le informazioni risultino meglio disperse all'interno della rete.

Anti-entropy ha uno schema di funzionamento abbastanza semplice: il nodo che genera l'informazione la invia ad altri host nel suo range di comunicazione. Questi host ripeteranno a loro volta tale processo. Siccome gli host si muovono, ognuno di essi potrebbe raggiungere una partizione diversa della rete considerata, diffondendo anche lí l'informazione. Lo scambio tra due host è bidirezionale: ognuno invia all'altro una lista di messaggi che possiede, e quest'ultimo può decidere quali non possiede e chiederne l'invio.

Le varianti su questo schema generale possono poi essere diverse in base allo scopo particolare che si intende raggiungere o al livello di performance che si vuole mantenere. Per prima cosa la frequenza con cui vengono iniziate le sessioni di anti-entropy può variare al fine di controllare la quantità di traffico sulla rete; si può decidere di iniziarne una solo quando viene generata localmente un'informazione oppure continuare periodicamente ad avviare sessioni. È possibile decidere di inviare, ad ogni sessione, solo l'ultima informazione generata oppure tutte quelle conosciute fino a quel momento; anche questo permette un controllo sulla quantità di traffico presente nella rete. Si può pensare di avviare le sessioni con un solo nodo oppure con un sottoinsieme di essi al fine di diminuire il tempo di distribuzione; i nodi con cui effettuare lo scambio possono inoltre essere scelti a caso oppure secondo diversi criteri - ad esempio mantenendo una storia degli ultimi nodi che ho incontrato posso di volta in volta scegliere quello che non vedo da più tempo. Studi riguardanti i vari metodi possibili per la scelta dei vicini

sono già stati svolti in letteratura [6] al fine di determinare le prestazioni di ognuno. È possibile infine porre un limite alla quantità di informazioni che ogni nodo può memorizzare per evitare sprechi di risorse - sia in termini di memoria che di energia dissipata - e decidere ad ogni sessione quale tra le informazioni vecchie eliminare - ad esempio adottando un metodo FIFO, LIFO oppure priority-based.

In teoria - e sembrerebbe anche in pratica [8] - dato uno scambio casuale di messaggi tra i vari host, tutti i messaggi vengono visti da tutti gli host entro un limitato ammontare di tempo.

Supporrò che ogni messaggio ha associato un lifetime L_m. Un obiettivo dell'algoritmo è quello di far pervenire m alla stragrande maggioranza degli host in rete prima che L_m scada.

Ogni nodo n mantiene al suo interno una tabella dei vicini e una tabella dei messaggi. La tabella dei vicini viene aggiornata tramite beaconing; la tabella dei messaggi è formata da tuple che contengono un identificativo univoco del messaggio, il lifetime dello stesso e una serie di dati relativi al messaggio che vengono utilizzati per determinare il comportamento del nodo[1] (*e.g.* ogni quanto tempo inviare il messaggio, ecc...). Quando L_m scade la tupla viene cancellata dalla tabella dei messaggi.

2.1 Gossip

L'algoritmo *Gossip* è stato sviluppato in uno studio precedente [7]. Esso è un tipico algoritmo epidemico caratterizzato da un round epidemico R: quando un nodo viene contagiato, fa partire un timer pari a R allo scadere del quale, il nodo invierà a sua volta il messaggio. Una volta inviato, il timer viene avviato di nuovo per inviare ancora un messaggio trascorso un altro periodo di tempo lungo sempre R. Questo procedimento viene ripetuto fino allo scadere di L_m.

La dimensione di R determina l'aggressività dell'algoritmo: un valore piccolo farà inviare più pacchetti rispetto ad un valore grande. A seconda del lifetime del dato si può impostare R per permettere un contagio più o meno veloce: più R è piccolo,

[1] Ad esempio nella tabella di messaggi potrebbe esserci un contatore per mantenere il numero di duplicati ricevuti per un dato messaggio m, il numero di volte che è stato trasmesso m, ecc...

maggiore sarà la velocità di invio dei pacchetti, e quindi maggiore sarà il numero di pacchetti inviati entro la scadenza del lifetime. Se vengono inviati più pacchetti aumenta la probabilità di raggiungere ogni nodo della rete in meno tempo.

Sebbene questo algoritmo potrebbe raggiungere, con buona probabilità, la totalità dei nodi della rete, esso non salvaguardia affatto né le risorse di rete né quelle di sistema. Ogni nodo continua a riavviare un timer per ogni messaggio m anche se l'informazione è già stata sufficientemente diffusa.

L'algoritmo deve essere migliorato: bisogna cercare di stimare il grado di diffusione del messaggio utilizzando le sole informazioni locali che ogni nodo possiede. Una stop condition diventa fondamentale in questo algoritmo di gossipping, in quanto permetterebbe di rallentare o addirittura arrestare la trasmissione dei messaggi una volta che questi siano sufficientemente diffusi, riducendo il consumo delle risorse di rete e di sistema.

Capitolo 3

Progetto di un algoritmo per il broadcast

L'algoritmo proposto, per poter rallentare la trasmissione mano a mano che il numero di nodi contagiati cresce, deve riuscire a stimare il grado di diffusione dei messaggi da diffondere. Dal momento che si opera su reti ad-hoc, l'intera rete potrebbe essere partizionata ed ogni nodo potrebbe passare da partizioni dove l'informazione è già stata diffusa ad altre nelle quali i nodi non sono mai stati raggiunti dall'informazione. L'algoritmo dovrebbe essere in grado di rilevare questi cambiamenti, permettendo al nodo di essere più aggressivo nel diffondere l'informazione quando la partizione non è stata ancora contagiata, e meno aggressivo quando l'informazione è già stata sufficientemente diffusa.

3.1 Strategia adattiva in base ai duplicati - DySEDA

Ho implementato una nuova versione di Gossip, tenendo conto degli accorgimenti appena esposti. Questa nuova versione è stata denominata *DySEDA*: *Dynamic Session Epidemic Diffusion Algorithm*.

Un modo per stimare il grado di diffusione di un dato messaggio m è contare il numero D_m di duplicati di m che vengono ricevuti dalla precedente trasmissione. Un numero alto di duplicati è sintomo di una diffusione alta di m, mentre un valore D_m piccolo indica un contagio appena iniziato.

In questo algoritmo non vi è più un round epidemico R fisso, ma ogni nodo man-

tiene per ogni messaggio una finestra di sessione SW_m, di dimensione fissa, all'interno della quale viene scelto il nuovo round epidemico ogni volta che viene mandato un pacchetto.

Il numero D_m di duplicati ricevuti per m influisce sul limite superiore (*upperbound*, b_U) ed inferiore (*lowerbound*, b_L) della finestra, la cui dimensione è mantenuta costante ($b_U - b_L = k$): ogni nodo conta i duplicati ricevuti durante un round epidemico. Al termine del round il pacchetto viene inviato, e si controlla il numero di duplicati. Se $D_m > UpperTh$ i limiti della finestra dovrebbero essere ingranditi, mentre se $D_m < LowerTh$ i limiti dovrebbero essere diminuiti.

Indipendentemente dalla funzione che modifica i limiti della finestra, permettere ad essa di "shiftare" sia in positivo (i limiti aumentano) sia in negativo (i limiti vengono diminuiti) potrebbe introdurre un problema: un nodo riceve meno duplicati solo se cambia partizione - da una già contagiata ad una non contagiata - oppure se i suoi vicini rallentano la trasmissione aumentando sempre più i limiti della finestra. Contando solo i duplicati non si è in grado di discernere i due casi: se alcuni nodi si stanno muovendo a sciame (ovvero sono tutti nel loro range di comunicazione) e sono stati appena contagiati, essi invierebbero i pacchetto ad intervalli brevi, in quanto il round è al minimo. I pacchetti inviati da ogni nodo verranno ricevuti da tutto il gruppo. Mano a mano che ogni componente del gruppo riceve duplicati, innalzerà le due soglie, per poter rallentare. Queste soglie cresceranno parallelamente in tutti i nodi, che dopo qualche tempo si ritroveranno tutti con round lunghi. Se tutti i nodi hanno un round lungo, ognuno di essi registrerà sempre meno duplicati. Quando il numero di duplicati sarà diventato minore della soglia $LowerTh$, i limiti della finestra di sessione cominceranno a diminuire, rendendo il nodo in questione via via più aggressivo.

Se la diminuzione del numero di duplicati è dovuta ad un allontanamento del nodo dallo sciame verso una nuova partizione non ancora contagiata, l'abbassamento delle soglie ha senso. Se però la diminuzione di D_m è dovuta ad un innalzamento comune a tutti gli elemento del gruppo delle soglie b_U e b_L, allora si sta andando incontro ad un effetto a "fisarmonica": ogni elemento del gruppo ha ingrandito le soglie in quanto il numero di duplicati era troppo alto. Ad un certo punto le soglie diventano abbastanza grandi tali da far sì che ogni nodo riceva un numero di duplicati minore di $LowerTh$.

Quindi le soglie vengono diminuite, ed ogni nodo riprende ad inviare pacchetti in maniera aggressiva. Tuttavia i vicini sono sempre gli stessi, quindi il numero di duplicati ricevuti ritornerà ad essere maggiore di $UpperTh$: le soglie cresceranno e cosi' via.

Per ovviare a questo inconveniente, ho imposto che b_U e b_L possono solo aumentare.

La funzione utilizzata per aumentare i limiti è un'esponenziale, precisamente 5^x: al termine di ogni round se $Diff_m > UpperTh$, x viene incrementato di uno e $b_L = 5^x$. Il limite superiore, dato che la finestra è costante, viene calcolato come $b_U = b_L + |SW|$.

Ho inserito anche un limite superiore per i valori b_U e b_L per evitare che diventino troppo grandi. Se questi valori crescono troppo, diminuisce la probabilità che un nodo, passando da una partizione ad un'altra, possa inviare un pacchetto nella nuova partizione, avviando anche lì il contagio.

Nel caso che un nodo riceva parecchi duplicati, potrebbe essere opportuno evitare che esso trasmetta allo scadere del round. Si potrebbe quindi inserire un'altra soglia $JSession > UpperTh$: se al termine di un round si verifica la relazione $D_m > JSession$, non viene trasmesso alcun messaggio e, siccome $JSession > UpperTh$, i limiti della finestra verranno incrementati ulteriormente.

L'introduzione della soglia *JSession* potrebbe avere anche dei riscontri negativi: siccome ogni nodo regola i limiti della finestra di sessione contando i duplicati ricevuti, introdurre un meccanismo che permette la soppressione della trasmissione potrebbe "ingannare" i vicini del nodo che non ha trasmesso. In un caso limite si potrebbe supporre di avere un nodo non ancora contagiato nel range di altri *n* nodi già contagiati da tempo. Il round di questi nodi potrebbe essere già molto dilatato, quindi dopo qualche tempo, anche il nodo non contagiato verrà raggiunto dal messaggio. Questo nodo comincerebbe a trasmettere con un round molto piccolo. In questo caso, il round degli altri *n* nodi in range potrebbe essere abbastanza grande da permettere ad essi di ricevere sempre un numero di duplicati $D_m > JSession$. Essi sopprimerebbero sempre la trasmissione, e il nodo appena contagiato non rallenterebbe più.

A seguito di questa considerazione, la soglia *JSession* è stata tolta. L'algoritmo può essere rappresentato dalla Figura 3.1

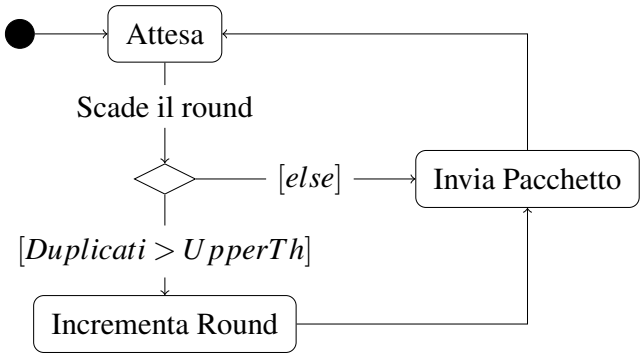


Figura 3.1: Schema dell'algoritmo DySEDA

In definitiva, l'algoritmo DySEDA permette ai nodi di rallentare la diffusione del messaggio quando esso è stato sufficientemente diffuso nell'area considerata.

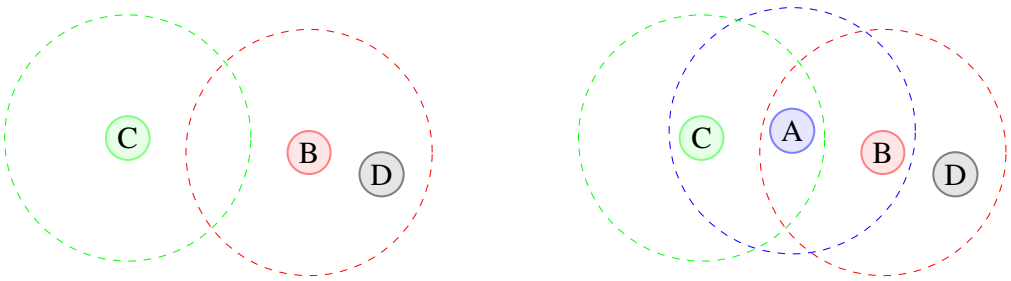
Nonostante ciò l'algoritmo presenta ancora dei problemi: allo scadere del round epidemico il pacchetto viene inviato in ogni caso, anche se non ci sono destinatari in range. Questa è una situazione che si verifica facilmente in ambienti con densità bassa di nodi. È necessario quindi inserire nell'algoritmo un controllo per sapere se ci sono vicini in range, per evitare di inviare pacchetti che non vengono ricevuti da nessuno.

3.2 Strategia reattiva in base ai vicini - NEMO

Il contatto fra i nodi è utile per sapere quando conviene inviare *m*. Ogni nodo conosce il proprio vicinato grazie ai beacon che riceve.

Con questo nuovo algoritmo intendo abbandonare la strada intrapresa con DySEDA riguardante l'analisi dei duplicati, per definire una politica di stop condition basata sull'analisi del vicinato. Il nuovo algoritmo è stato nominato *NEMO*: *NEighbors' Meeting Opportunities*.

Per stimare la variazione del vicinato, ogni volta che un nuovo nodo entra in range viene aggiornata una bitmap *BM*. Per aggiornare la bitmap, viene calcolata una funzione hash sull'identificativo del nuovo vicino; la funzione ritornerà un indice *i* compreso tra zero e la lunghezza della bitmap. La bitmap verrà aggiornata nella posizione *i* ponendo 1 se nella posizione c'è 0 o viceversa.



(a) Prima (b) Dopo

Figura 3.2: Il nodo *A* dovrebbe essere avvantaggiato rispetto al nodo *B*

Quando un nodo trasmette un messaggio m salva una copia della bitmap nella entry della tabella dei messaggi relativa ad ogni messaggio m presente nella tabella dei messaggi. Ad ogni aggiornamento della tabella dei vicini viene calcolata la distanza di Hamming HD tra la bitmap salvata nella entry relativa ad m e quella che viene continuamente aggiornata. Il valore HD assieme al numero NV dei vicini attualmente in range, ricavato dalla grandezza della tabella dei vicini, viene utilizzato per calcolare un valore $Perc = \frac{HD \cdot 100}{NV}$.

$Perc$ offre una stima di cambio del vicinato. Per dare priorità nella trasmissione ai nodi che hanno cambiato più vicini rispetto ad altri che hanno cambiato di poco il vicinato rispetto alla loro ultima trasmissione, $Perc$ viene confrontato con un'altra soglia: NTh. Se $Perc \geq NTh$ viene inviato il pacchetto, altrimenti la trasmissione viene soppressa.

Ovviamente un nodo che cambia più vicini ha maggior probabilità di avere in range nodi non ancora infettati, quindi è vantaggioso che trasmetta esso prima di altri. Supponendo $NTh = 1$ (cioè: basta un vicino per trasmettere), due nodi che si incontrano si contenderebbero subito la trasmissione. Ad esempio, in una situazione come quella in Figura 3.2, il nodo C non è ancora stato contagiato, al contrario di A, B e D. Quando A entra nel range di B (e C), B calcola una variazione di vicinato pari al 50%, in quanto D era già presente alla precedente trasmissione; A calcola una variazione del vicinato pari al 100%. Ponendo $NTh = 100$, solo A potrebbe trasmettere. Il valore NTh determina quindi il minimo cambio di vicinato tale per cui è conveniente trasmettere, avvantaggiando quei nodi che hanno registrato un maggior cambio di vicinato e

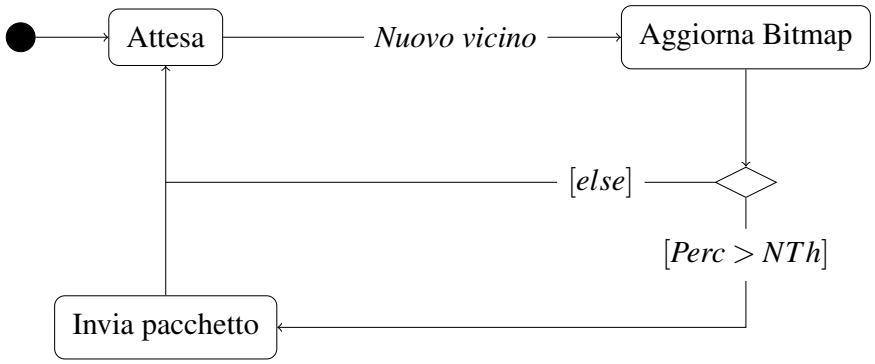


Figura 3.3: Schema dell'algoritmo NEMO

sopprimendo la trasmissione dei nodi che, in percentuale, hanno contato un cambio minore.

L'algoritmo NEMO (rappresentato nella Figura 3.3) riesce a sfruttare le opportunità di contatto tra i nodi per diffondere l'informazione. Tuttavia non ha alcun "freno" che permetta di rallentare la diffusione del messaggio. Col passare del tempo, sempre più nodi verranno contagiati e continuare a trasmettere ad ogni contatto potrebbe risultare inutile, in quanto diminuisce la probabilità di incontrare nodi che non sono ancora stati raggiunti dal messaggio.

3.3 Strategia ibrida - JIG

L'algoritmo DySEDA permette di sapere *se* conviene mandare il messaggio, mentre l'algoritmo NEMO indica *quando* mandare il messaggio. L'unione dei due algoritmi potrebbe fornire un algoritmo che implementa una buona stop condition.

Ho scelto di partire da NEMO ed applicare ad esso alcune tecniche usate anche in DySEDA, costruendo un nuovo algoritmo di nome *JIG*: *JIG Improves Gossip*.

L'algoritmo NEMO permette la trasmissione a tutti i nodi che calcolano un valore $Perc \geq NTh$: se un nodo rileva un cambiamento del vicinato tale per cui $Perc \geq NTh$, invia immediatamente un pacchetto. Nel caso che più nodi, tutti nel loro stesso range di comunicazione, rilevino un cambiamento del vicinato tale per cui $Perc \geq NTh$,

trasmetterebbero tutti il messaggio. Questo comportamento sarebbe corretto se ogni nodo avesse dei nuovi vicini che non sono in range con nessun altro nodo che sta per trasmettere. Nel caso che alcuni nuovi vicini siano nel range di più nodi che stanno per inviare m non c'è bisogno che tutti quelli che hanno rilevato $Perc \geq NTh$ trasmettano per contagiare i nuovi vicini, ne basta un sottogruppo.

In JIG, se un nodo rileva $Perc \geq NTh$, prima di trasmettere attende un tempo random "piccolo"[1] W_r. Durante questo piccolo intervallo di tempo il nodo conta il numero di duplicati D'_m che riceve da quando ha rilevato $Perc \geq NTh$ allo scadere di W_r.

Introduco una nuova soglia $DupAll$: un nodo potrà trasmettere dopo che è trascorso il tempo W_r se e solo se $D'_m \leq DupAll$. Se più nodi rilevano $Perc \geq NTh$, non è affatto detto che ogni nodo abbia rilevato un nuovo vicino raggiungibile solo da lui. Soprattutto in ambienti molto popolati, i nuovi vicini potrebbero essere in un'area raggiunta da più dispositivi. La soglia $DupAll$ permette la trasmissione solo di alcuni di essi, per risparmiare banda e risorse di sistema dei dispositivi che non trasmettono.

Il valore di $DupAll$ influisce sulle prestazioni dell'algoritmo, in particolare sul tempo di consegna dei pacchetti all'interno della rete. Esso dovrebbe essere maggiore di zero per evitare che possa trasmettere un solo nodo tra tutti quelli che contendono per la trasmissione: potrebbero esserci nodi *ponte* tra due partizioni, l'una già contagiata e l'altra no. Ponendo $DupAll = 0$, più sono i nodi in contesa, minore è la probabilità che ciascuno ha di trasmettere; oltretutto bisognerebbe avvantaggiare i nodi ponte, perché tramite essi l'informazione potrebbe essere diffusa anche in partizioni non ancora contagiate. Siccome un nodo non può sapere se è ponte tra due partizioni o no, l'unico modo per aumentare la probabilità che anche essi trasmettano è quella di impostare $DupAll$ ad un valore maggiore di zero. $DupAll$ inoltre dovrebbe essere adattato dinamicamente in base alla densità dei nodi della partizione nella quale ci si trova, mantenendolo comunque sempre all'interno di un valore massimo ed un valore minimo per evitare rispettivamente sprechi di canale oppure ritardi nella diffusione dei messaggi.

L'algoritmo JIG potrebbe già risparmiare risorse rispetto a NEMO, in quanto qual-

[1] Per piccolo si intende minore dell'intervallo di beaconing, in modo da non trasmettere troppo avanti nel tempo e incorrere nel problema che i vicini siano usciti dal range di comunicazione.

che trasmissione viene soppressa da *DupAll*, ma non ha ancora un "freno" che, col passare del tempo, faccia rallentare la trasmissione. Il punto di forza di DySEDA è proprio il meccanismo di rallentamento, che non fa crescere linearmente il numero di pacchetti inviati col passare del tempo. Questo meccanismo sfrutta il numero di duplicati ricevuti dal nodo nel round appena terminato.

D'altro canto, in DySEDA l'utilizzo di un round introduce due problemi: allo scadere del round il nodo trasmette un pacchetto anche se non ha alcun possibile destinatario ed inoltre la conta dei duplicati implica che un nodo può solo rallentare la trasmissione, e mai velocizzarla.

Tuttavia, se ogni nodo potesse sapere quanto tempo fa i mittenti dei duplicati ricevuti sono stati contagiati, potrebbe adattare il proprio round in funzione di questa informazione: se il mittente è stato contagiato *poco* tempo fa, probabilmente il messaggio deve ancora essere diffuso all'interno della partizione nella quale ci si trova e il nodo che ha ricevuto il duplicato dovrebbe aiutare a diffondere il messaggio. Maggiore è il tempo trascorso dall'istante nel quale il nodo mittente del duplicato è stato contagiato, maggiore è la probabilità che tutti i nodi della partizione abbiano ricevuto una copia del messaggio, quindi la reazione migliore da parte del nodo sarebbe quella di sopprimere gran parte delle trasmissioni per risparmiare banda e risorse di sistema.

Inserendo due campi nei pacchetti che vengono inviati rappresentati il primo - t_i - il tempo trascorso da quando il messaggio è stato creato a quando il mittente è stato contagiato ed il secondo - t_x - il tempo passato da quando è stato creato il messaggio a quando è stato inviato il pacchetto, il destinatario può sapere quanto tempo fa il nodo mittente è stato contagiato per la prima volta.

La Figura 3.4 mostra come avviene lo scambio dei messaggi con all'interno i due valori t_i e t_x. Utilizzando degli intervalli di tempo relativi non c'è bisogno di avere la sincronizzazione dei timer, problema molto difficile da risolvere in un ambiente distribuito; ogni nodo inoltre riesce a stimare il grado di diffusione nella propria partizione calcolando $Diff_m = t_x - t_i$: più il valore $Diff_m$ è piccolo, meno tempo è passato da quando il mittente è stato contagiato.

Quando un nodo riceve un nuovo messaggio m' crea la struttura dati per contenere dei valori utili per poter stimare il grado di diffusione del messaggio. In particolare,

il nodo salverà nella entry relativa a m' della tabella dei messaggi un numero t'_i pari al tempo t_x contenuto nel pacchetto appena ricevuto. Quando questo nodo ritrasmetterà m' invierà un pacchetto formato da $< m, t'_i, t'_x >$, dove t'_x è il tempo che è passato da quando il messaggio è stato creato al momento dell'invio.

In JIG viene stabilito *quando* trasmettere in base al cambio del vicinato e non in base alla scadenza di un round come accadeva in DySEDA. Per riuscire a sfruttare il valore $Diff_m$ e rallentare la trasmissione ho introdotto un parametro P_n che rappresenta la probabilità che ogni nodo ha di inviare un pacchetto. La probabilità P_n entrerà in gioco se un nodo, dopo aver registrato un valore $Perc \geq NTh$, riceve un numero di duplicati $D'_m \leq DupAll$ nell'intervallo di attesa W_r.

P_n può oscillare tra due valori massimi e minimi: rispettivamente MAX_p e MIN_p. Come in DySEDA c'è un valore limite per il round, anche in JIG c'è un valore minimo per la probabilità, ed il motivo è il medesimo: un nodo con $P_n = 0$ che attraversa una partizione non ancora contagiata non ha alcuna probabilità di trasmettere un pacchetto in quanto la probabilità potrebbe essere aumentata solo dalla ricezione di un duplicato. Un valore $MIN_p \neq 0$ potrebbe permettere l'invio di un pacchetto.

Inoltre è stato inserto nell'algoritmo un valore $TIME\_TH$ che indica il tempo massimo che può trascorrere tra due invii: quando un nodo invia un pacchetto fa partire un timer T di valore $TIME\_TH$. Ogni volta che il nodo trasmette azzera il timer. Quando questo timer scade, il nodo trasmette appena rileva un nuovo vicino: in questo caso quindi non viene controllata né *Perc* né viene atteso un tempo W_r.

Il range dei valori assumibili da P_n è cruciale per determinare le prestazioni dell'algoritmo: a seconda di come vengono configurati i limiti inferiore e superiore del range, l'algoritmo si comporta in maniera più o meno aggressiva, modificando il trade-off tra tempi di consegna e consumo di risorse.

Anche il valore di $TIME\_TH$ influisce sulle performance dell'algoritmo, e dovrebbe essere mantenuto abbastanza grande (nell'ordine delle decine di minuti): $TIME\_TH$ è stato inserito esclusivamente per far fronte a situazioni "estreme", nelle quali i nodi possono vagare per lunghi periodi senza incontrare e senza trasmettere nessun pacchetto, oppure per garantire una trasmissione ogni $TIME\_TH$ secondi in caso che MIN_p sia posto ad un valore troppo basso.

P_n vale MAX_p appena un nodo viene contagiato. Il valore della probabilità viene aggiornato ad ogni duplicato ricevuto: l'algoritmo aggiornerà P_n in funzione del valore $Diff_m$; la funzione utilizzata per aggiornare P_n è la seguente:

$$update(x) = \begin{cases} \frac{100-x}{1000}, & \text{se } x \leq \text{JIG\_TH;} \\ -\frac{x}{10000}, & \text{se } x > \text{JIG\_TH;} \end{cases}, JIG\_TH \geq 100$$

dove, al posto di x verrà passato il valore $Diff_m$.

Il valore di P_n aggiornato viene calcolato sommando a P_n il risultato di $update(Diff_m)$.

JIG_TH influisce sulle performance dell'algoritmo: più il valore di questa costante è grande, più verrà eseguita l'equazione che divide per 1000 il valore di $Diff_m$. Al contrario, un assegnamento piccolo a JIG_TH renderà l'aggiornamento della probabilità più lieve, in quanto il valore $Diff_m$ viene diviso per 10000.

Se un nodo A entra in una nuova partizione appena contagiata, riceverà duplicati con $Diff_m$ piccolo, Se $Diff_m$ è minore di 100 P_n aumenterà, dando l'opportunità al nodo di dare il proprio contributo alla diffusione del messaggio anche nella nuova partizione. Non è necessario che aumenti fino a tornare ad 1: è sufficiente che venga trasmesso un pacchetto: i nuovi contagiati cominceranno a diffondere con probabilità 1, e quindi con molta aggressività.

In caso che il nodo A entri in una partizione contagiata parecchio tempo fa, i valori $Diff_m$ saranno molto grandi: la funzione $update(Diff_m)$ assumerà un valore negativo, che diminuirà P_n. Più $Diff_m$ è grande, più il valore P_n viene diminuito.

La Figura 3.5 rappresenta un diagramma dell'algoritmo JIG. Questo algoritmo, unione dei due precedenti, reagisce in base al cambiamento di vicinato per riuscire ad inviare i pacchetti solo quando si è certi di avere dei destinatari, ed inoltre, tramite i timestamp all'interno dei pacchetti riesce ad aggiornare il campo P_n evitando l'effetto "fisarmonica".

Nel prossimo capitolo introdurrò il simulatore usato per implementare gli algoritmi e per misurarne le prestazioni.

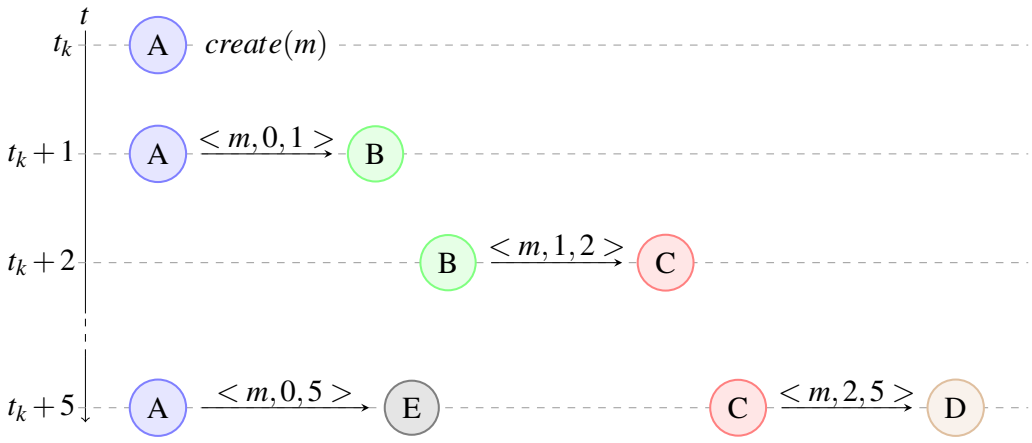


Figura 3.4: Scambio di messaggi fra diversi nodi

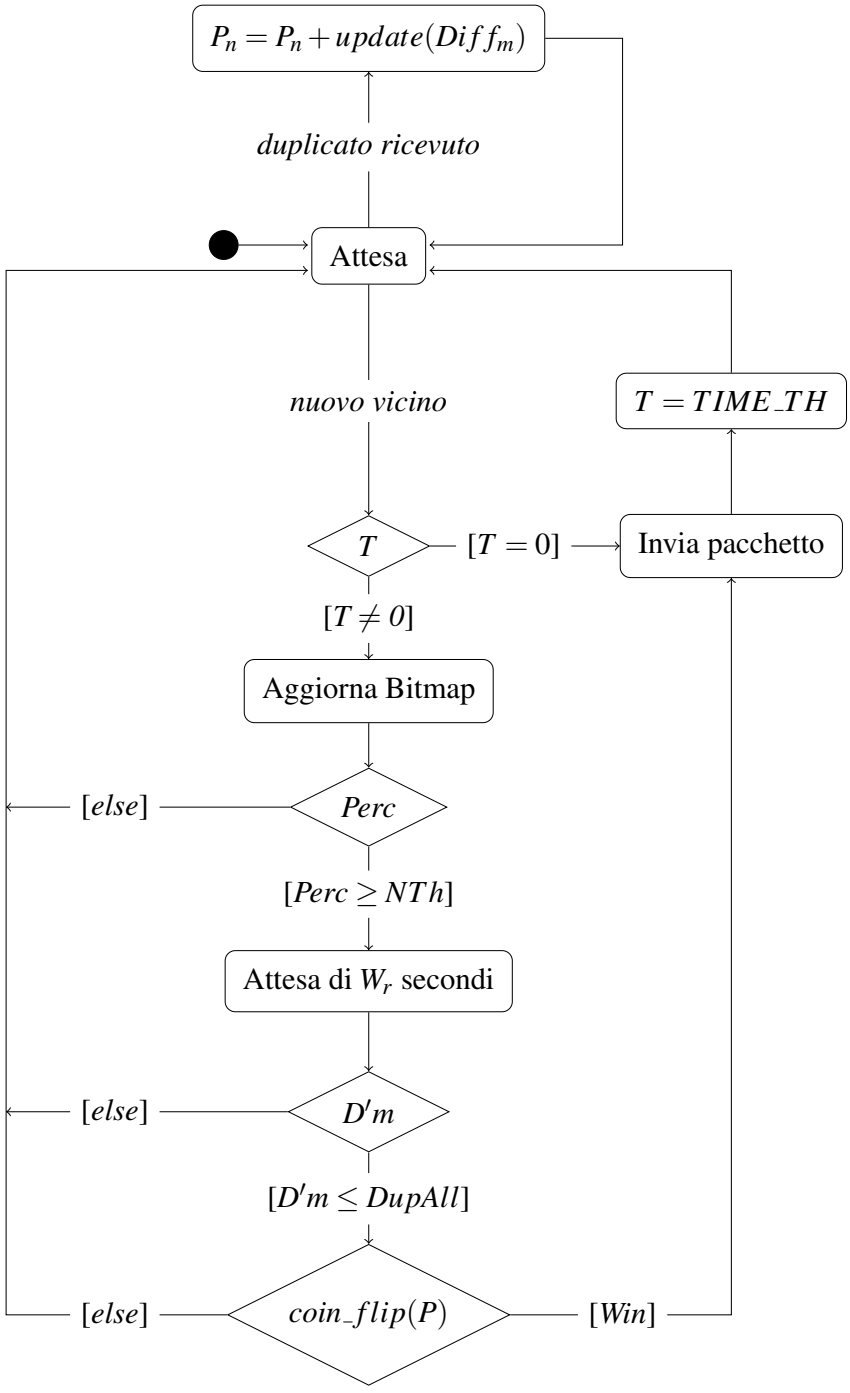


Figura 3.5: Schema dell'algoritmo JIG

Capitolo 4

Simulazioni

Le analisi degli algoritmi proposti nel Capitolo 3 sono state condotte implementando gli algoritmi nel simulatore GloMoSim [1] misurando le prestazioni degli stessi.

GloMoSim è un simulatore di apparati di rete wireless che implementa i cinque livelli della pila protocollare TCP-IP. Per definire i parametri di simulazione bisogna editare il file *config.in*. Al suo interno si possono indicare diversi parametri di simulazione quali la durata della simulazione, la grandezza dell'area all'interno della quale si muovono i dispositivi wireless e il numero di nodi da simulare. Inoltre, è possibile definire la strategia di posizionamento dei nodi scegliendo tra diverse opzioni quali random, a griglia oppure definendo un file all'interno del quale vengono indicate le posizioni di ogni nodo. Oltre al posizionamento è possibile definire il tipo di movimento dei nodi partendo da *NONE*, che indica l'assenza di movimento, ad un modello Random Waypoint[1], fino a definire un file che indica per ogni nodo il movimento che deve essere seguito da esso.

All'interno del file di configurazione si trovano anche parametri per controllare il funzionamento dell'antenna dei dispositivi: è possibile indicare il limite di propagazione, modelli di propagazione del segnale, temperatura dell'ambiente, e scegliere fra tre diversi modelli di propagazione del segnale: free-space, two-ray o pathloss-matrix.

Infine è possibile definire, sempre editando il file *config.in* il protocollo di routing da utilizzare per le simulazioni e quali statistiche esportare dalla simulazione.

[1] in tal caso è possibile definire la velocità massima e minima dei nodi e il tempo di pausa.

Il codice del simulatore è organizzato in directory: ce ne sono cinque relative ai cinque livelli dello stack TCP-IP: *radio*, *mac*, *network*, *transport*, *application*.

Per sviluppare un nuovo protocollo di routing è necessario editare alcuni file, per integrare completamente il protocollo all'interno del simulatore.

Innanzitutto il protocollo deve offrire delle funzioni di interfaccia per essere utilizzato con Glomosim[2]. Le funzioni sono:

Routing[Protocol_name]Init: è la funzione di avvio: viene chiamata da ogni nodo prima dell'avvio della simulazione. Devono essere inserite istruzioni di inizializzazione del protocollo, come il reset delle statistiche di rete o l'eventuale avvio di protocolli a supporto (quali ad esempio il beaconing);

Routing[Protocol_name]HandleProtocolPacket: è la funzione che viene richiamata quando dal livello MAC arriva un pacchetto per il proprio protocollo di rete. Deve quindi gestire il pacchetto del protocollo;

Routing[Protocol_name]HandleProtocolEvent: viene richiamata quando scade un evento generato dal nodo stesso. Utilizzata per gestire i timer.

Routing[Protocol_name]Finalize: richiamata alla fine della simulazione, in genere si occupa solo di provvedere alla stampa delle statistiche raccolte.

Il protocollo definito deve essere reso disponibile a tutti i livelli del sistema necessari e all'utente. Deve essere definito in diversi file:

include/network.h: al tipo di dato "NetworkProtocolType" deve essere aggiunto il nuovo protocollo `ROUTING_PROTOCOL_[Protocol_Name];`

include/nwcommon.h: è necessario specificare il numero del nuovo protocollo da associare all'header IP: `#define IPPROTO_[Protocol_name];`

application/application.pc: a livello applicazione è possibile specificare la funzione di inizializzazione (`Routing[protocol_name]Init`), sempre che il protocollo di routing debba essere inizializzato a livello applicazione (strutture dati condivise). Altrimenti, il protocollo di routing deve comunque essere specificato senza alcuna funzione di `Init` (che verrà specificata in seguito, nelle funzioni di interfaccia a livello network).

[2]Il bind di queste funzioni deve essere effettuato nel file `nwip.pc`, all'interno della sottodirectory `network`

A questo punto è necessario definire i nuovi messaggi che verranno utilizzati dai timer impostati dal protocollo (messaggi associati ai timer del nodo) all'interno del file `include/structmsg.h`, inserendo `MSG_[protocol_name]_[msg_type]`. Per permettere il corretto funzionamento è inoltre necessario inserire il tipo di pacchetto nel file `network/diffusionState.h`, struttura `DiffusionPacketType`.

Nel caso vengano sviluppati nuovi file `.pc` oppure `.h` sarà necessario inserirli nel `makefile` per permetterne la compilazione, al termine della quale sarà possibile utilizzare il nuovo protocollo configurando opportunamente il file config.in.

4.1 Codice sviluppato

Ho implementato nel simulatore il tre algoritmi esposti nel Capitolo 3, per testarne le prestazioni. Ogni protocollo è stato sviluppato in due file: `< protocol_name.pc >` e `< protocol_name.h >`. Tutti e tre gli algoritmi presuppongono un algoritmo di beaconing che permetta di gestire correttamente la tabella dei vicini.

4.1.1 DySEDA

Il codice dell'algoritmo DySEDA è stato il primo ad essere sviluppato: esso deriva principalmente dal codice di antientropy [7]. A questo codice sono state applicate delle modifiche, creando così il codice di DySEDA, contenuto in due file: `dyseda.pc` e `dyseda.h`.

Nel file `dyseda.h` vengono definite le strutture dati di ogni nodo: a quelle "standard" già presenti ne sono state aggiunte 3:

numDuplicates un intero necessario per contare i duplicati ricevuti all'interno del round epidemico;

lowerBound un intero utilizzato per definire il lowerbound della finestra di sessione. La dimensione della finestra è fissata ad un valore tramite la costante simbolica `SESSION_WINDOW`, dichiarata nel medesimo file;

x un altro intero necessario a contare il numero di round all'interno dei quali il nodo ha

ricevuto un numero di duplicati maggiori alla soglia $UpperTh$, definita tramite la costante simbolica `DUP_THRESHOLD`.

Nel file `dyseda.pc` vengono definite tutte le funzioni dell'algoritmo. È stata definita una funzione `DysedaManageSessionWindow` deputata all'aggiornamento di limiti della Session Window:

```
1 void DysedaManageSessionWindow(GlomoNode *node) {
2  GlomoRoutingDyseda *structures = GetDysedaStructures(node);
3  if (structures->numDuplicates > DUP_THRESHOLD) {
4   structures->x++;
5   DysedaIncrementLowerbound(node);
6   structures->numDuplicates = 0;
7  }
8 }
```

Il codice della procedura non fa altro che controllare il valore dei duplicati ricevuti dal nodo, richiamando `DysedaIncrementLowerbound` quando il numero dei duplicati ricevuti è maggiore della soglia `DUP_THRESHOLD`

La procedura `DysedaIncrementLowerbound(node)` implementa la funzione di incremento dei limiti della finestra di sessione. Il limite inferiore della finestra viene incrementato esponenzialmente:

```
structures->lowerBound = pow(BASE, structures->x);
```

dove il valore di `BASE` viene definito nell'header file `dyseda.h`.

La funzione che incrementa la finestra è esponenziale: una funzione lineare introdurrebbe un rallentamento, ma permetterebbe ai nodi di inviare comunque un numero troppo alto di pacchetti. Una funzione esponenziale invece cresce abbastanza velocemente, e dovrebbe mostrare un reale risparmio nei pacchetti inviati, contagiando comunque la totalità dei nodi nell'area.

4.1.2 NEMO

Come per DySEDA, anche per quanto riguarda il codice del protocollo NEMO sono partito da antientropy.

Anche per NEMO, sono stati sviluppati due file: `nemo.pc` e `nemo.h`. All'interno dell'header file è stata definita una costante simbolica: `NTh`. Essa definisce la percentuale minima di cambio di vicinato per inviare un pacchetto.

La struttura dati dei nodi è stata modificata rispetto ad antientropy aggiungendo:

neighbors mantiene il numero dei nodi in range di comunicazione. Questo valore è stato aggiunto per alleggerire l'esecuzione del codice: sebbene il numero dei nodi in range sia sempre disponibile contando il numero di entry presenti nella lista dei vicini, percorrere tutta la lista dei vicini ogni volta che un nodo rileva un nuovo vicino potrebbe diventare oneroso, soprattutto in caso di ambienti molto popolati, dove la tabella dei vicini potrebbe contenere parecchi elementi.

bitmapN la bitmap aggiornata ogni volta che viene rilevato un nuovo vicino;

bitmapO la bitmap relativa all'ultimo invio del nodo[3];

active serve per evitare che un nodo faccia partire dei timer inutilmente quando non conosce ancora alcuna informazione da inviare. Quando il nodo viene contagiato questo valore viene posto a `TRUE`, abilitando il nodo ad inviare pacchetti.

Le procedure all'interno del file `dyseda.pc` sono state modificate annullando la ritrasmissione secondo un round epidemico e inserendo le nuove funzioni di aggiornamento e di salvataggio delle bitmap.

Il round epidemico, nel file `gossip.pc` veniva impostato nella procedura chiamata `RoutingGossipHandleProtocolPacket`, quando veniva gestito il pacchetto di tipo `GOSSIP`. La procedura `RoutingNemoHandleProtocolPacket`, all'interno di `nemo.pc`, è stata modificata cancellando la linea che richiamava la procedura `NemoSetNextMessageTimer`, in modo da non settare mai alcun timer.

Inoltre, nella medesima funzione è stato aggiunto il seguente codice:

```
1 case NEMO:
2   if (RoutingNemoProcessData(node, msg)) {
3     GlomoRoutingNemo *structures = GetNemoStructures(node);
4     structures->active = 1;
```

[3]Questa bitmap dovrebbe essere replicata per ogni messaggio. Tuttavia, siccome il codice per ora gestisce un unico evento, è inutile replicare la bitmap.

```
5   }
6 break;
```

La procedura `RoutingNemoProcessData` ritorna un valore `TRUE` se non è mai stato ricevuto un messaggio del tipo di `msg`, `FALSE` in caso contrario.

Le funzioni per aggiornare e salvare le bitmap sono le seguenti:

void NemoUpdateBitmap (GlomoNode *node, int nodeAddr) prende in input il nodo su cui effettuare l'aggiornamento della bitmap e l'indirizzo del nuovo vicino;

int NemoGetIndex (int nodeAddr) è la *hash function* usata per mappare gli indirizzi dei nodi in un identificatore numerico compreso tra 0 e la lunghezza della bitmap (espressa nell'header file tramite una costante simbolica `BITMAP_LENGTH`);

int NemoGetHD (unsigned a, unsigned b) ritorna la distanza di Hamming usata per calcolare *Perc*;

void NemoSaveBitmap (GlomoNode* node) salva la bitmap. Per ora la funzione è alquanto banale, poiché il modello è stato pensato per funzionare su un solo evento. Un'implementazione futura della procedura presenterà in input un identificatore del messaggio, per salvare la bitmap nella corretta entry della tabella dei messaggi;

int NemoGetPerc (GlomoNode* node) ritorna il valore di *Perc*. Come per la precedente funzione, in un'implementazione futura dovrà essere indicato il messaggio, per ritornare il valore di *Perc* relativo al messaggio indicato.

La procedura `NemoUpdateBitmap` viene eseguita ogni volta che il nodo rileva nuovi vicini, ovvero nella procedura `RoutingNemoHandleProtocolPacket`, quando viene rilevato un pacchetto di tipo `BEACONING` inviato da un nodo il cui identificativo non è dentro la tabella dei vicini:

```
1 case BEACONING: {
2   BeaconingPacket* beaconingPacket =
3           (BeaconingPacket*)GLOMO_MsgReturnPacket(msg);
4   if(!BeaconingNeighborOld(beaconingPacket->sourceAddr,
5         nemoLayer->beaconingStructures.neighborsTable)) {
6     //this is a new neighbor
```

```
7     GlomoRoutingNemo* structures = GetNemoStructures(node);
8     NemoUpdateBitmap (node, beaconingPacket->sourceAddr);
9     structures->neighbors++;
10
11    //control percentage
12    if (NemoGetPerc(node) >= NTh &&
13              structures->active ) {
14      NemoSendMessage(node, SESSION);
15    }
16  }
17 [...]
18 break;
```

La procedura alla riga #4 cerca all'interno della tabella dei vicini se il nodo identificato da `beaconingPacket->sourceAddr` è già presente nella tabella; se non lo è aggiorna la bitmap, aumenta il numero dei vicini e, se le condizioni sono verificate, invia il messaggio.

4.1.3 JIG

Per sviluppare il codice del protocollo JIG sono partito dal codice di NEMO creando i due file `jig.pc` e `jig.h`. Innanzitutto, ho aggiunto un campo all'interno dei pacchetti inviati modificando il file `epidemic.h`, dove è definita la struttura dei pacchetti:

```
typedef struct epidemic_packet_str {
    DiffusionPacketType pkt_type;
    EpidemicMsgType type;
    NODE_ADDR sourceAddr;
    EpidemicEventListEntry* eventList;
    clocktype first;
} EpidemicPacket;
```

Il campo `first` è stato aggiunto per permettere ad ogni nodo di inviare nel pacchetto il timestamp della sua prima infezione[4].

Il codice dell'algoritmo è nei file `jig.pc` e `jig.h`.

Nell'header file `jig.h` sono state definite le seguenti costanti simboliche (oltre a quelle definite per NEMO):

JIG_RATE il tempo che passa da quando un nodo rileva la variazione di vicinato necessaria per inviare all'istante nel quale dovrebbe inviare il pacchetto;

MAX_PROB e MIN_PROB rispettivamente valore massimo e minimo che può assumere il campo `probability`. Sebbene matematicamente la probabilità è un valore compreso tra 0 e 1, in questo caso risulta più semplice trattarla come un intero che può assumere tutti i valori compresi tra `MAX_PROB` e `MIN_PROB` inclusi.

Alla struttura dati dei nodi sono stati aggiunti i seguenti campi:

duplicates serve per contare i duplicati ricevuti da quando viene settato il timer a quando dovrebbe essere inviato il pacchetto;

setTimer se impostato a `TRUE` significa che il nodo ha già impostato un timer per inviare i pacchetti, quindi non ne verrà impostato un altro;

first contiene il timestamp di quando è stato infettato il nodo. L'algoritmo dovrebbe basarsi su tempi relativi, tuttavia, dato che tramite il simulatore è possibile avere la sincronizzazione degli orologi, viene utilizzato tranquillamente un timestamp assoluto;

probability la probabilità che il nodo ha di inviare il pacchetto o no.

Il codice è stato arricchito con nuove procedure: in particolare, nella procedura `RoutingJigHandleProtocolPacket` viene eseguito il seguente codice:

```
1 if (JigGetPerc(node) >= NTh && !structures->setTimer) {
2   JigSetNextMessageTimer(node);
3 }
```

[4] Il protocollo JIG, così come viene esposto nel Capitolo 3 necessiterebbe di due campi nel pacchetto per inserire i due intervalli di tempo t_i e t_x. Tuttavia tramite il simulatore Glomosim è possibile avere un orologio condiviso da tutti i nodi: nel pacchetto viene inviato solamente il timestamp di prima infezione; il tempo t_x è il timestamp al momento della ricezione, dato che l'evento viene creato all'inizio dell'esecuzione.

Così facendo, invece di inviare subito un pacchetto viene avviato un timer del tipo `MSG_JigSendMessageTimeout` lungo `JIG_RATE`.
La procedura `JigSetNextMessageTimer` si occupa anche di settare il valore `setTimer` a `TRUE`.

La procedura deputata alla gestione dei timer che scadono è sempre all'interno di `jig.pc`: `RoutingJigHandleprotocolEvent`. Quando scade un timer del tipo `MSG_JigSendMessageTimeout`, la procedura esegue il seguente codice:

```
1 [...]
2 if ( (structures->duplicates <= DUPALLOWED) &&
3             JigCoinFlip(node) )
4   JigSendMessage(node, SESSION);
5 structures->duplicates = 0;
6 structures->setTimer = 0;
7 JigSaveBitmap(node);
8 [...]
```

Si controlla che il numero di duplicati ricevuti da quando è partito il timer sia inferiore alla soglia `DUPALLOWED`. La procedura `JigCoinFlip` estrae un intero random compreso tra 0 e `MAX_PROB`. Se l'intero estratto è inferiore o uguale al valore `probability` la funzione ritorna `TRUE`; se entrambe le condizioni sono verificate viene inviato il pacchetto dopo di che viene posto a zero `setTimer`, per permettere al nodo di settare il prossimo timer.

La procedura `JigUpdateProbability` aggiorna il valore della probabilità in funzione dei tempi contenuti nei pacchetti ricevuti.

Capitolo 5

Risultati

Ho eseguito delle simulazioni con il simulatore GloMoSim per testare l'algoritmo.

Ogni simulazione è stata effettuata con i seguenti parametri[1]:

TERRAIN-DIMENSIONS 1000x1000 (m)

NODE-PLACEMENT RANDOM

MOBILITY RANDOM WAYPOINT, pausa: 0 s, velocità massima: 2 m/s, velocità minima 0 m/s.

PROPAGATION-LIMIT -90 dBm

PROPAGATION-PATHLOSS TWO-RAY

NOISE-FIGURE 10.0

TEMPERATURE 290.0 K

RADIO-TYPE RADIO-ACCNOISE

RADIO-FREQUENCY 2.4e9 Hz

RADIO-BANDWIDTH 2000000 (bps)

RADIO-RX-TYPE SNR-BOUNDED[2]

RADIO-RX-SNR-THRESHOLD 10.0 (dB)

RADIO-TX-POWER 2.0 (dBm)

RADIO-ANTENNA-GAIN 0.0 (dB)

[1] I valori di `RADIO-TX-POWER`, `RADIO-ANTENNA-GAIN`, `RADIO-RX-SENSITIVITY` e `RADIO-RX-THRESHOLD` impostati in questa maniera offrono un range di comunicazione di 30 metri.

[2] Questo parametro indica che se il SNR (*Signal to Noise Ratio*) è maggiore di `RADIO-RX-SNR-THRESHOLD`, il pacchetto viene ricevuto senza errori.

RADIO-RX-SENSITIVITY -77.0 (dBm)
RADIO-RX-THRESHOLD -67.7 (dBm)
MAC-PROTOCOL 802.11
PROMISCUOUS-MODE NO

Gli istogrammi riguardanti il tempo di contagio presenteranno due barre sovrapposte. Esse rappresentano il tempo medio di consegna ed il tempo massimo, ovvero il tempo nel quale è stato raggiunto l'ultimo nodo dell'area.

5.1 Simulazioni

5.1.1 k statico

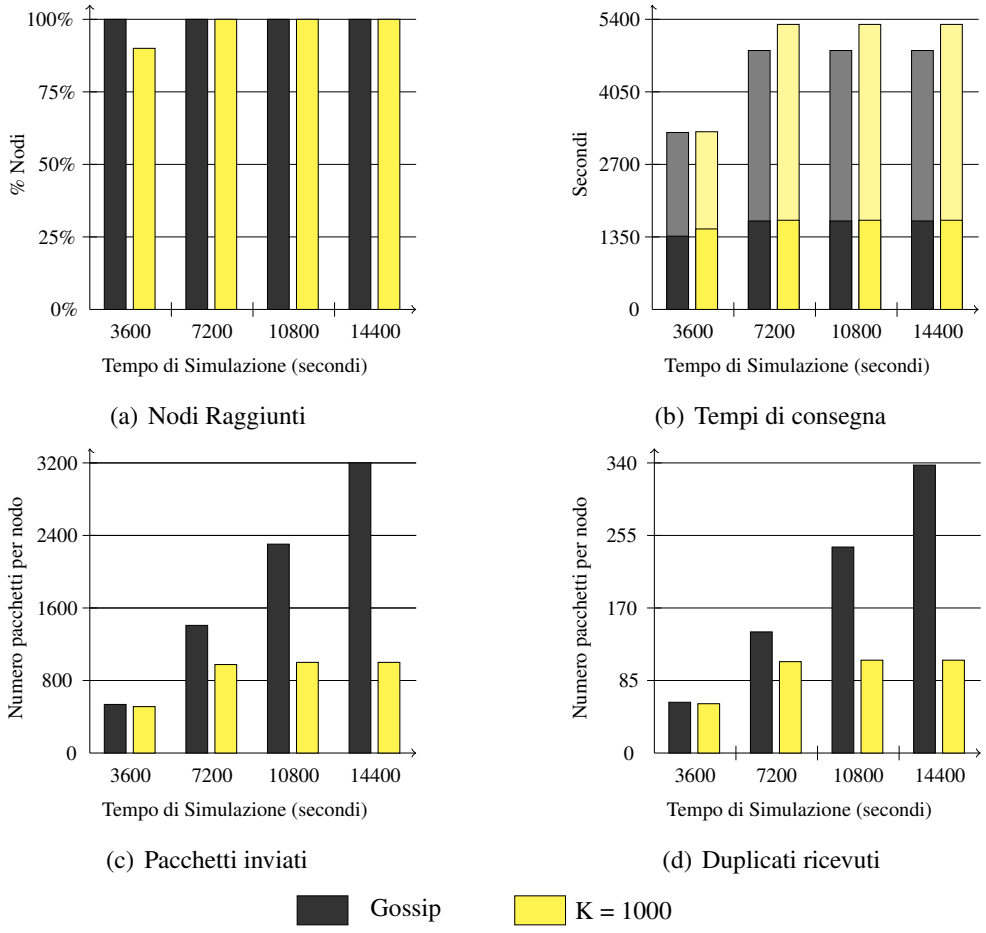
Le prime simulazioni sono state condotte per confrontare l'andamento dell'algoritmo Gossip con round pari a 4 secondi senza alcuna stop condition con il medesimo algoritmo al quale è stata applicata una stop condition abbastanza banale: un intero k prefissato indicante il numero di pacchetti che ogni nodo può inviare.

Per un ambiente con 25 nodi in un'area di 1000x1000m (Figura 5.1), il valore ottimale per k è 1000[3], mentre per un ambiente con 200 nodi in un'area grande uguale alla precedente (Figura 5.2) un valore ottimale risulta essere 320, che riesce a raggiungere tutti i nodi dell'area entro 1550 secondi.

Come si nota dai grafici in Figura 5.1(a) e Figura 5.2(a), l'algoritmo con stop condition riesce a far pervenire l'informazione a tutti i nodi dell'area entro due ore, nonostante ognuno di essi invii un numero inferiore di pacchetti. Questi grafici indicano quindi che le supposizioni fatte nel Capitolo 2 sono valide: una stop condition permette di salvaguadardare le risorse di rete e di sistema di ogni dispositivo, permettendo tuttavia una diffusione dell'informazione a tutti i nodi dell'area entro tempi ragionevoli.

Una soluzione senza stop condition non ferma mai la trasmissione: col passare del tempo le due barre dell'istogramma saranno sempre più distanti in quanto l'algoritmo

[3]Questi valori sono stati calcolati empiricamente svolgendo delle simulazioni per determinare quale fosse il valore "migliore". Per "migliore" si intende, nell'insieme dei k che permettono di raggiungere il 100% dei nodi dell'area entro 2 ore, quello che colleziona meno duplicati.



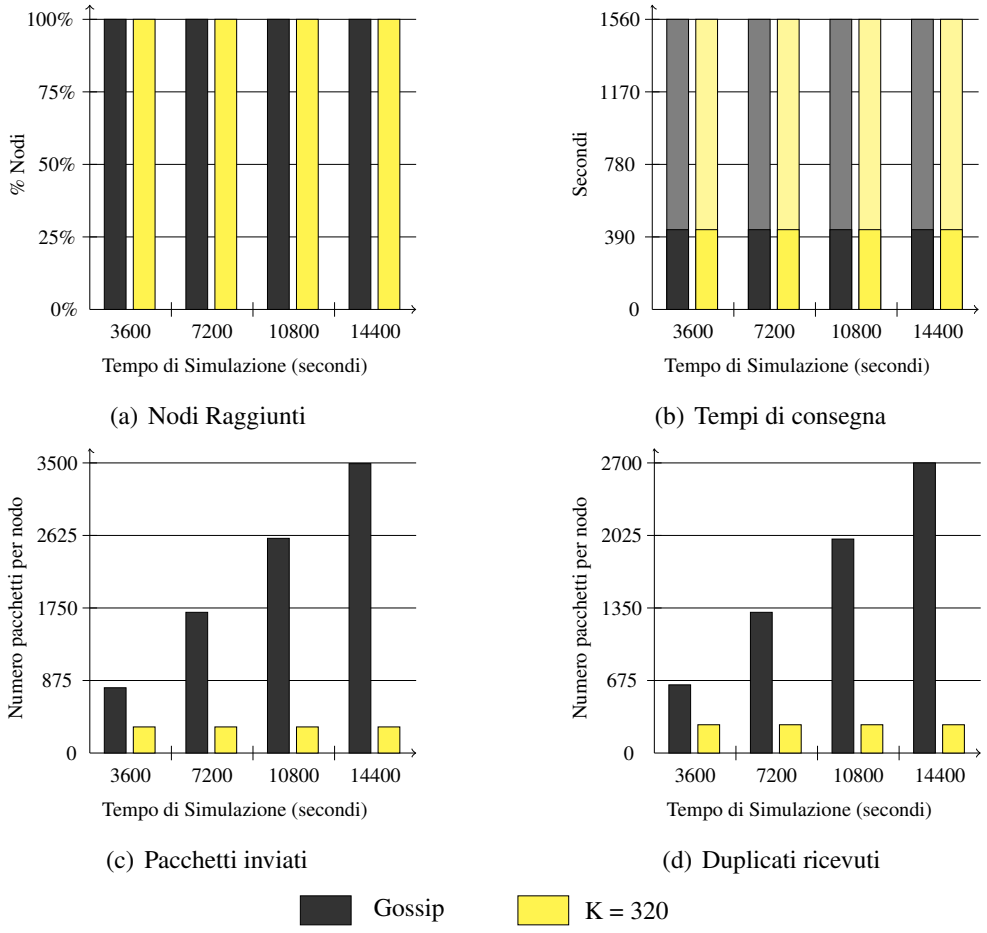
(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.1: Grafici per confrontare il protocollo Gossip originale ed il protocollo Gossip con l'aggiunta del valore K in un ambiente a bassa densità di nodi.



(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

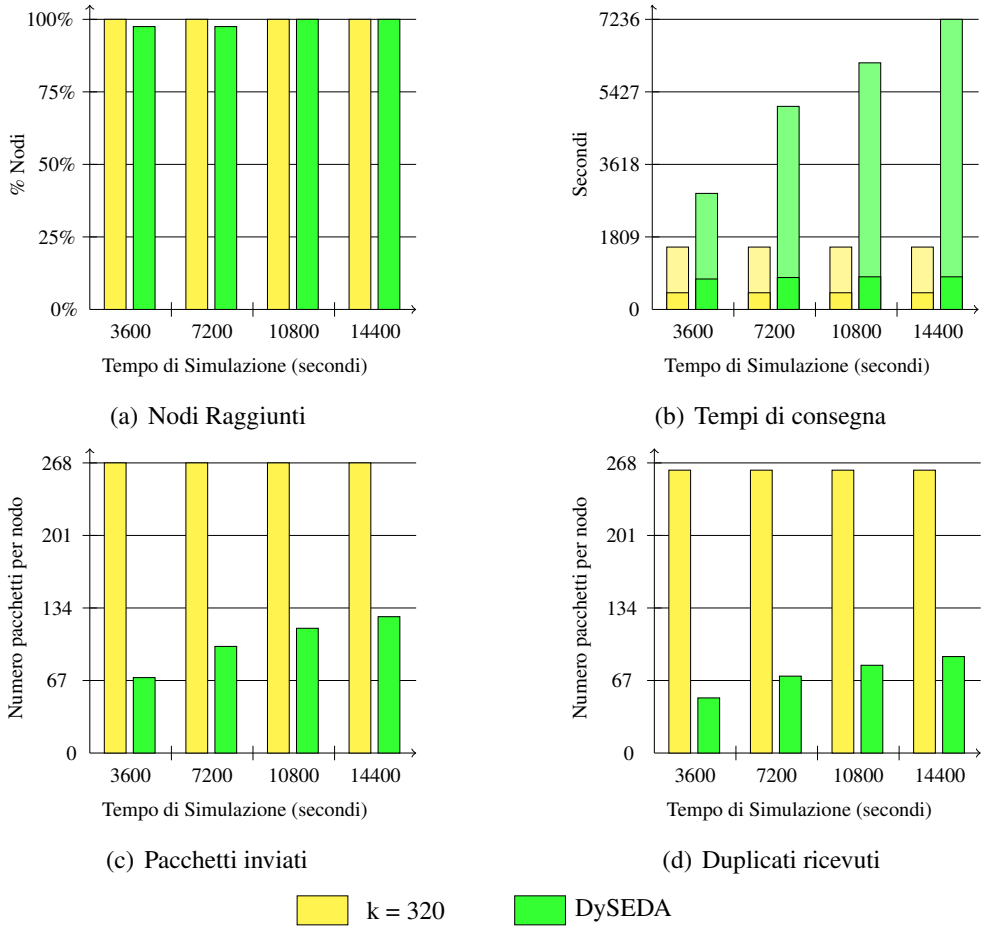
(d) Duplicati ricevuti

Figura 5.2: Grafici per confrontare il protocollo Gossip originale ed il protocollo Gossip con l'aggiunta del valore K in un ambiente ad alta densità di nodi.

con stop condition non trasmetterà più, mentre l'altro continuerà a trasmettere ogni 4 secondi.

Sebbene introduca un risparmio di risorse, la stop condition implementata non è sicuramente una delle migliori: essa non scala, in quanto dipende dalla densità dei nodi: in un ambiente scarsamente popolato bisogna permettere ad ogni nodo di inviare un numero di pacchetti molto alto (ad esempio 1000) per permettere la diffusione del messaggio: ogni nodo invia un pacchetto ogni 4 secondi, anche se non ci sono ipotetici destinatari. Se il numero di invii a disposizione è troppo basso, ogni nodo sprecherebbe i suoi pacchetti senza riuscire a contagiare altri nodi.

Ogni nodo contagia i suoi vicini con una certa frequenza: quest'ultima è determinata dal periodo della sessione epidemica (indicata con S), vale a dire, quanti secondi passano tra un invio e l'altro. Preso atto che l'unica vera stop condition per la quale un nodo smetterà di inviare un'informazione è costituita dal *lifetime* del dato[4], è utile dunque lavorare sul periodo di sessione: si può controllare il numero di invii aumentandolo o diminuendolo in base a dei parametri. Così facendo ci sarà un decremento del numero medio di pacchetti che vengono inviati da ogni nodo rispetto ad una soluzione senza stop ed ogni nodo potrà adattarsi in base alla densità dell'ambiente in cui si trova per controllare i propri invii. Inoltre, in un ambiente partizionato e poco denso (come può essere quello con 25 nodi), un valore di S statico (e relativamente piccolo) non è vantaggioso: si contagerebbero tutti i nodi confinanti (probabilmente se la velocità è bassa, per più sessioni contigue i vicini saranno sempre gli stessi, che collezioneranno oltretutto un gran numero di duplicati), e, se la stop condition adottata fosse un semplice k statico, nel caso che un nodo arrivasse verso la fine del tempo di simulazione in un area del piano con nodi ancora immuni, non potrebbe più mandare pacchetti perché avrebbe già esaurito i k invii a disposizione.



(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.3: Grafici per confrontare il protocollo Gossip con l'aggiunta del valore k ed il protocollo DySEDA in un ambiente ad alta densità di nodi.

5.1.2 DySEDA

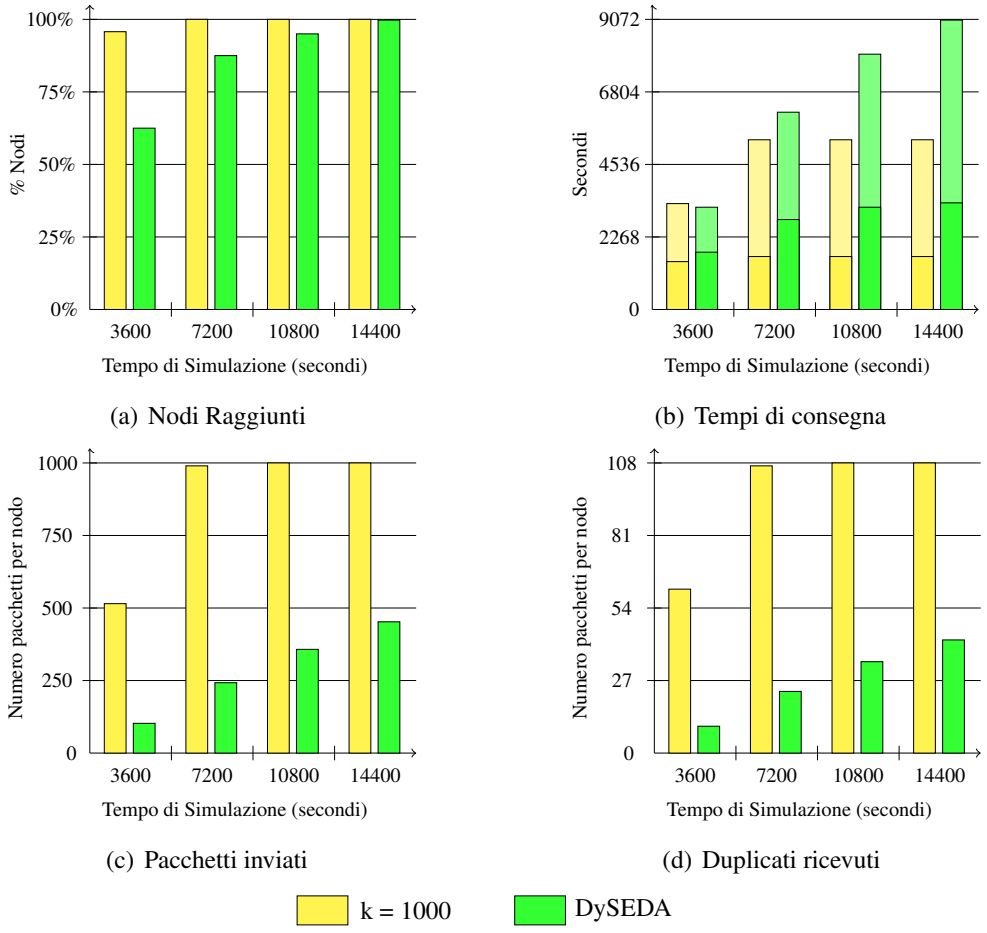
Anche l'algoritmo DySEDA è stato simulato con GloMoSim. I parametri adottati per le simulazioni sono i seguenti:

- `numDuplicates = 2`
- `SESSION_WINDOW = 2`
- `lowerBound = 2`

La Figura 5.3 mostra come l'algoritmo sia in grado di raggiungere buoni risultati (98% dei nodi contagiati entro la prima ora, con un invio di 70 pacchetti) rispetto al Gossip modificato con l'aggiunta del k in un ambiente ad alta densità di nodi. Dal grafico in Figura 5.3(b) si evince che l'algoritmo non gode più di una velocità di diffusione del messaggio pari a quella dell'algoritmo Gossip con un k, in quanto i tempi medi e massimi sono sempre maggiori di quest'ultimo algoritmo. Tuttavia il valor medio dei tempi di contagio risulta basso, il che è positivo: significa che la maggior parte dei nodi viene contagiata in fretta, mentre per l'ultima piccola parte viene impiegato più tempo, probabilmente perché questi ultimi nodi sono completamente sconnessi da partizioni già contagiate. Questi dati manifestano anche l'"aggressività" dell'algoritmo: ogni nodo, una volta contagiato, comincia a diffondere con un round epidemico molto veloce, come se fosse un Gossip tradizionale. Ecco quindi che un nodo è aggressivo appena contagiato, per poi rallentare mano a mano che riceve duplicati.

La Figura 5.3(c) permette di capire il guadagno offerto dall'algoritmo: a prima vista si osserva che i pacchetti inviati sono molti meno rispetto a quelli inviati dall'algoritmo Gossip con k. Un'analisi più scrupolosa porta alla luce un altro aspetto: il numero dei pacchetti inviati non aumenta linearmente col passare del tempo: si passa da 70 a 99, 115 per terminare a 126 pacchetti inviati in 4 ore: anche quest'aspetto è positivo! L'algoritmo effettivamente rallenta la trasmissione, permettendo di risparmiare risorse di rete e di sistema. Una diretta conseguenza del grafico in Figura 5.3(c) è la ricezione da parte dei nodi di meno duplicati, come mostra il grafico in Figura 5.3(d).

[4]Ogni nodo potrebbe entrare in qualsiasi momento in partizioni non ancora contagiate. Porre una stop condition che ferma in maniera assoluta la trasmissione potrebbe impedire il contagio di nuove partizioni.



(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.4: Grafici per confrontare il protocollo Gossip con l'aggiunta del valore k ed il protocollo DySEDA in un ambiente con densità di nodi bassa.

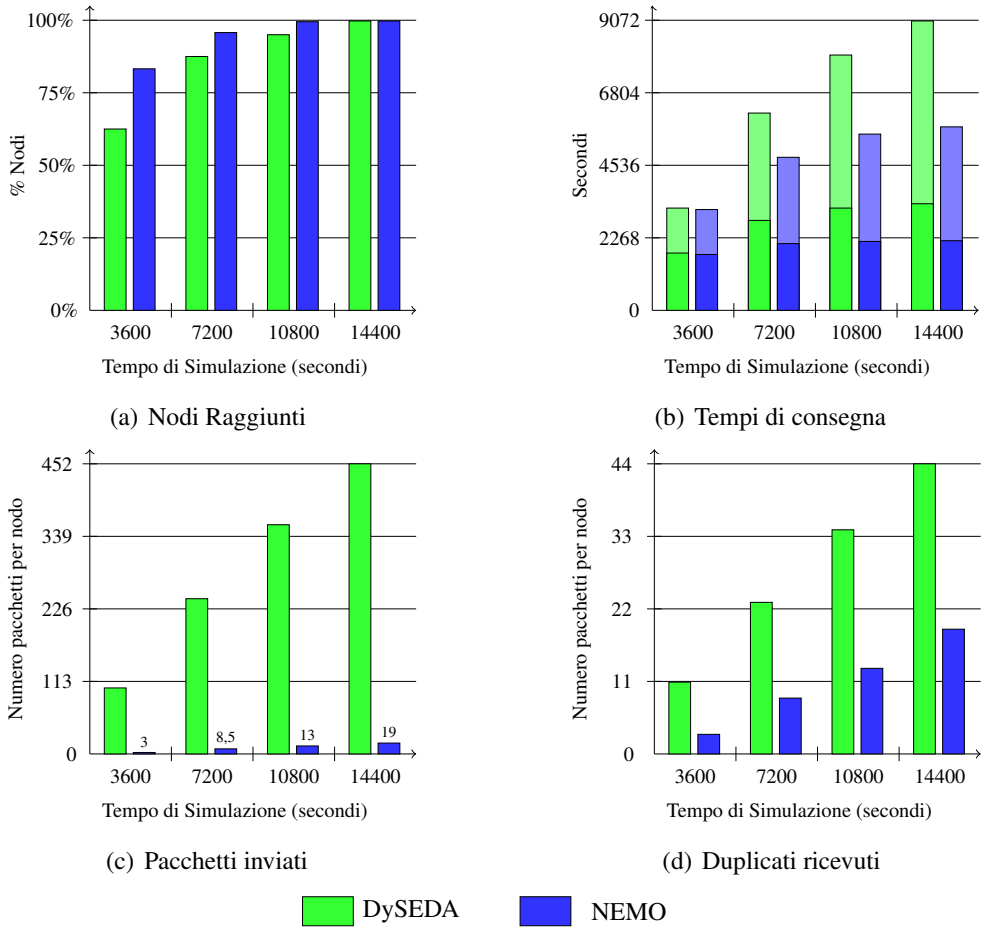
Sebbene in un ambiente molto popolato l'algoritmo di buoni risultati anche in ore, in uno scarsamente popolato necessita più tempo per far pervenire il messaggio alla totalità dei nodi nell'area. In Figura 5.4 vengono confrontati i risultati delle simulazioni in un ambiente scarsamente popolato degli algoritmi DySEDA e Gossip con l'aggiunta di un k pari a 1000. L'algoritmo DySEDA riesce a raggiungere soltanto il 58% dei nodi nella prima ora. In due ore raggiunge l'87%, in tre riesce a contagiare il 95% mentre in 4 ore raggiunge la totalità dei nodi dell'area. In un ambiente poco popolato l'algoritmo non da' delle buone performance: per come è stato ideato, necessita di ricevere dei duplicati per rallentare: dato che l'area è scarsamente popolata, ciò accade di rado: in Figura 5.4(c) si nota un miglioramento in fatto di numero di pacchetti inviati rispetto al Gossip con k, ma la crescita dei pacchetti inviati col passare del tempo viene rallentata molto meno rispetto ad un ambiente denso: si passa da 103 pacchetti inviati in un ora, 242 in due ore, 357 in tre e 453 in quattro. Il numero di duplicati ricevuti è basso: si passa da 10 in un'ora per arrivare ad un massimo di 43 in quattro ore. Questi valori indicano che la maggior parte dei pacchetti inviati è inutile, in quanto non vengono ricevuti da nessuno.

L'algoritmo si comporta sicuramente meglio del Gossip con k: in un ambiente molto popolato offre delle buone performance e un discreto risparmio delle risorse di rete e di sistema, mentre in un ambiente scarsamente popolato è in ogni caso meglio del Gossip con k che occupa parecchio canale inviando fino a 1000 pacchetti, ma non rallenta a sufficienza la sua trasmissione. Per tempi di simulazione lunghi il numero di pacchetti inviati da Dyseda potrebbe essere addirittura superiore a quello di Gossip con k, in quanto continua a crescere in maniera quasi lineare.

Un aspetto positivo è il seguente: l'algoritmo Gossip con k è stato adattato alle due densità di nodi cambiando il valore del k, mentre l'algoritmo DySEDA è rimasto lo stesso in entrambi i casi: ciò indica che l'algoritmo non necessita di parametrizzazione in funzione della densità di nodi.

5.1.3 NEMO

L'unico parametro dell'algoritmo NEMO - `neighbors` - è stato posto pari a 50.



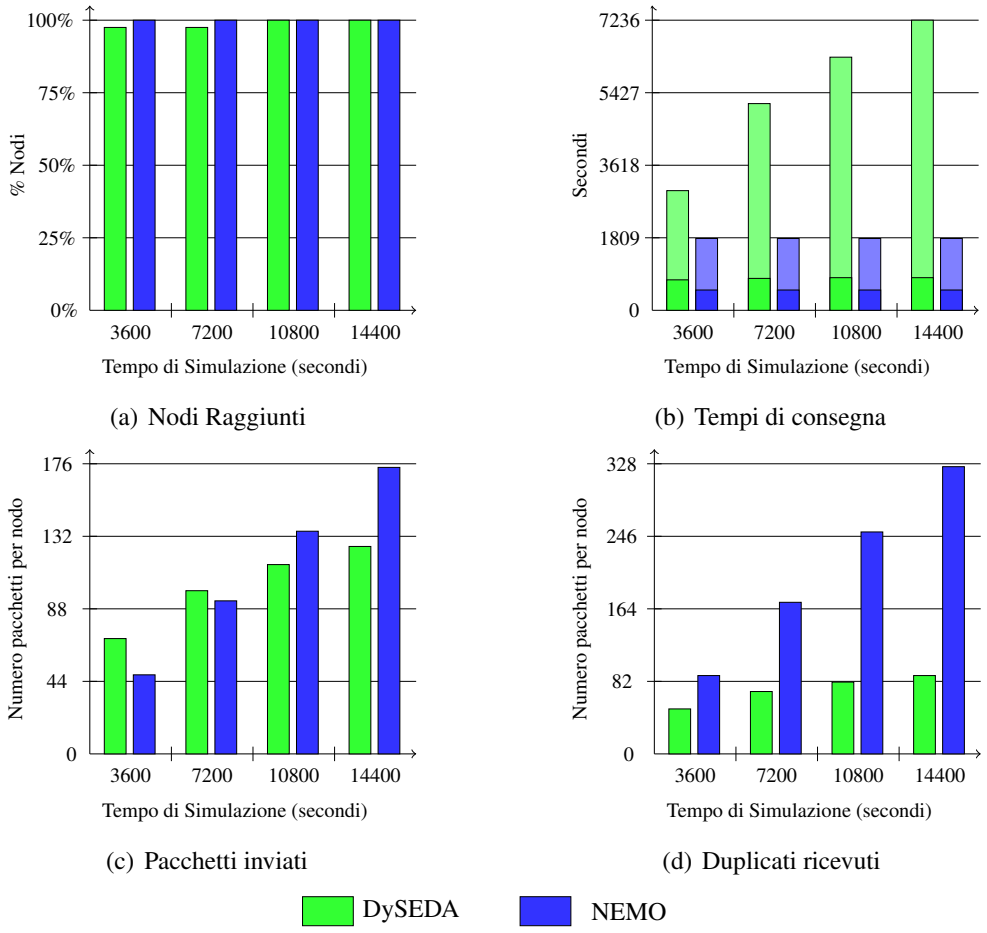
(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.5: Grafici per confrontare il protocollo DySEDA con il protocollo NEMO in un ambiente a bassa densità di nodi.



(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.6: Grafici per confrontare il protocollo DySEDA con il protocollo NEMO in un ambiente ad alta densità di nodi.

I risultati delle simulazioni dell'algoritmo NEMO sono stati confrontati con quelli di DySEDA. I dati sono stati riportati in Figura 5.5 per quando riguarda le simulazioni in ambiente scarsamente popolato e in Figura 5.6 per le simulazioni in ambiente ad alta densità di nodi.

Analizzando i grafici in Figura 5.5(a) e in Figura 5.6(a) si nota che l'algoritmo raggiunge più nodi di DySEDA in entrambi i casi. Inoltre, un altro punto a favore di NEMO è rappresentato dai grafici riguardanti i tempi di contagio (Figura 5.5(b) e Figura 5.6(b)): in entrambi i casi l'algoritmo è più veloce di DySEDA. I grafici in Figura 5.5(c) e in Figura 5.6(c) mostrano un andamento ottimo per quando riguarda un ambiente scarsamente popolato: l'algoritmo invia solamente 3 pacchetti in un'ora riuscendo a contagiare 83% dei nodi; in due ore vengono contagiati il 98,63% dei nodi con un invio medio di 8,5 pacchetti per nodo. Anche in quattro ore il numero di pacchetti inviati risulta molto basso: 19 pacchetti per nodo con il 100% dei nodi contagiati. In ambiente ad alta densità di nodi, l'algoritmo mostra un andamento "ambiguo": il numero di pacchetti inviati da NEMO continua ad aumentare in maniera quasi lineare, mentre DySEDA rallenta. Analizzando la Figura 5.6(c) si vede chiaramente che NEMO non rallenta la trasmissione: si passa da 48 pacchetti inviati per nodo in un'ora, a 93 in due ore, in tre ore ogni nodo invia 135 pacchetti mentre in quattro ore ne vengono inviati 174. DySEDA invece invia in un'ora 70 pacchetti, 99 in due ore, 115 in tre e 126 in quattro. Risulta evidente quanto sia importante unire le due soluzioni: l'algoritmo NEMO ha dalla sua parte la velocità (come mostra il grafico in Figura 5.6(b)), conseguenza diretta dal fatto che ogni nodo sfrutta le opportunità di contatto per inviare i messaggi. Tuttavia non rallenta mai, la trasmissione! In un ambiente scarsamente popolato ciò non influisce sulle performance, in quanto le opportunità di contatto sono poche, ma in un ambiente ad alta densità di nodi il rallentamento della trasmissione è cruciale per poter salvaguardare le risorse di rete e di sistema, dato che le opportunità di contatto sono tante! In Figura 5.6(d) si vede chiaramente la differenza tra le due strategie: il numero di duplicati massimo collezionato con DySEDA è pari a 89 pacchetti, mentre con NEMO si arriva fino a 325!

5.1.4 JIG

Ecco le impostazioni dei parametri per l'algoritmo JIG[5]:

- `JIG_RATE = 1`
- `MAX_PROB = 1000`
- `MIN_PROB = 100`
- `JIGTH = 400`
- `TIME_TH = 900`
- `NTh = 50`
- `DUPALLOWED = 2`

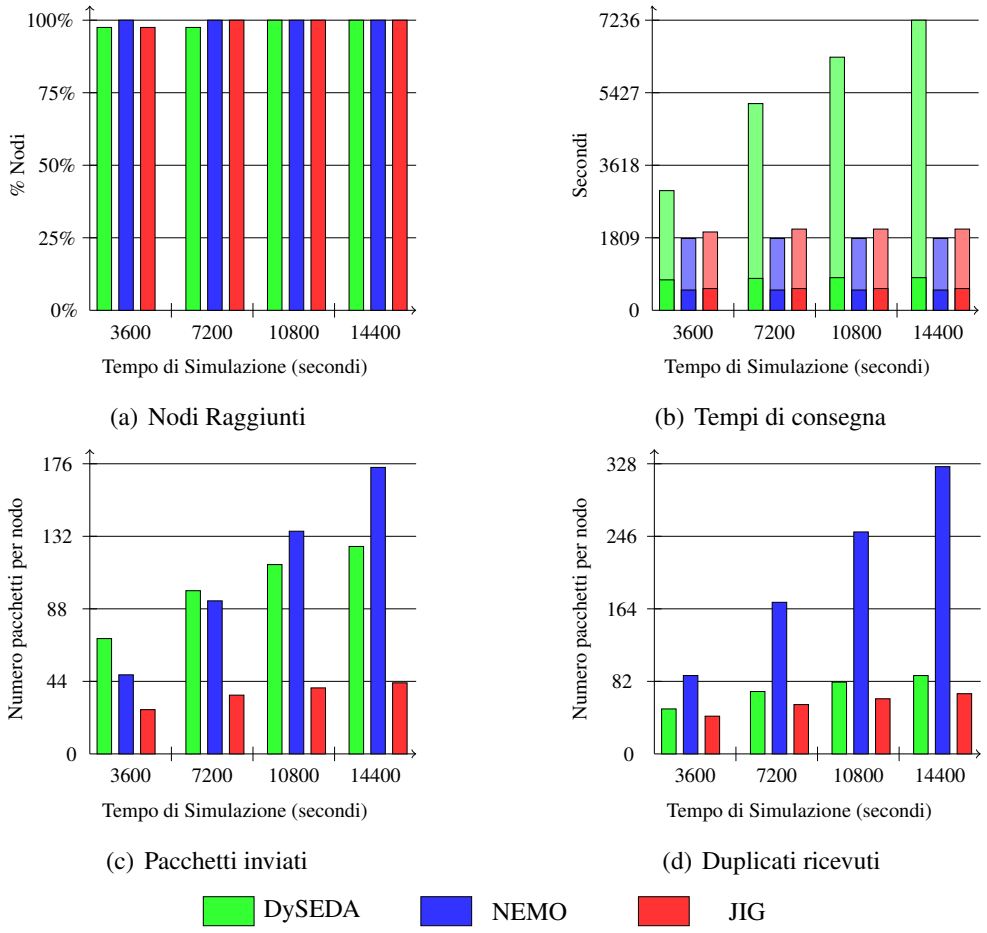
L'algoritmo JIG dovrebbe aver ereditato i punti di forza degli algoritmi DySEDA e NEMO.

I risultati delle simulazioni sono riportati in Figura 5.7 ed in Figura 5.8. Dai grafici si vede immediatamente che l'algoritmo è, globalmente, il migliore tra i tre: ereditando da NEMO la capacità di sfruttare i contatti fra i nodi non vengono inviati messaggi che non hanno destinatari. L'invio dei messaggi è controllato tramite l'inserimento nel pacchetto dei due campi t_i e t_x, per cui l'algoritmo può diminuire o aumentare il valore di P a seconda che la partizione nella quale si trova il nodo in esame sia stata appena contagiata oppure no, rallentando effettivamente l'invio dei pacchetti.

I grafici attestano che l'algoritmo permette, in un ambiente ad alta densità di nodi, un invio di 27 pacchetti in un'ora, 36 in due ore, 40 durante le tre ore e un massimo di 44 pacchetti in 4 ore. La crescita del numero di pacchetti inviati non è approssimabile ad un andamento lineare: JIG è quindi in grado di rallentare la trasmissione dei messaggi.

Per un ambiente poco popolato, il numero decresce di molto: 2 pacchetti in un'ora, 7 in due ore, 10 in tre e 12 in quattro, mostrando anche in questo caso una capacità

[5] I valori di `JIG_RATE`, `JIGTH` e `TIME_TH` sono espressi in secondi.



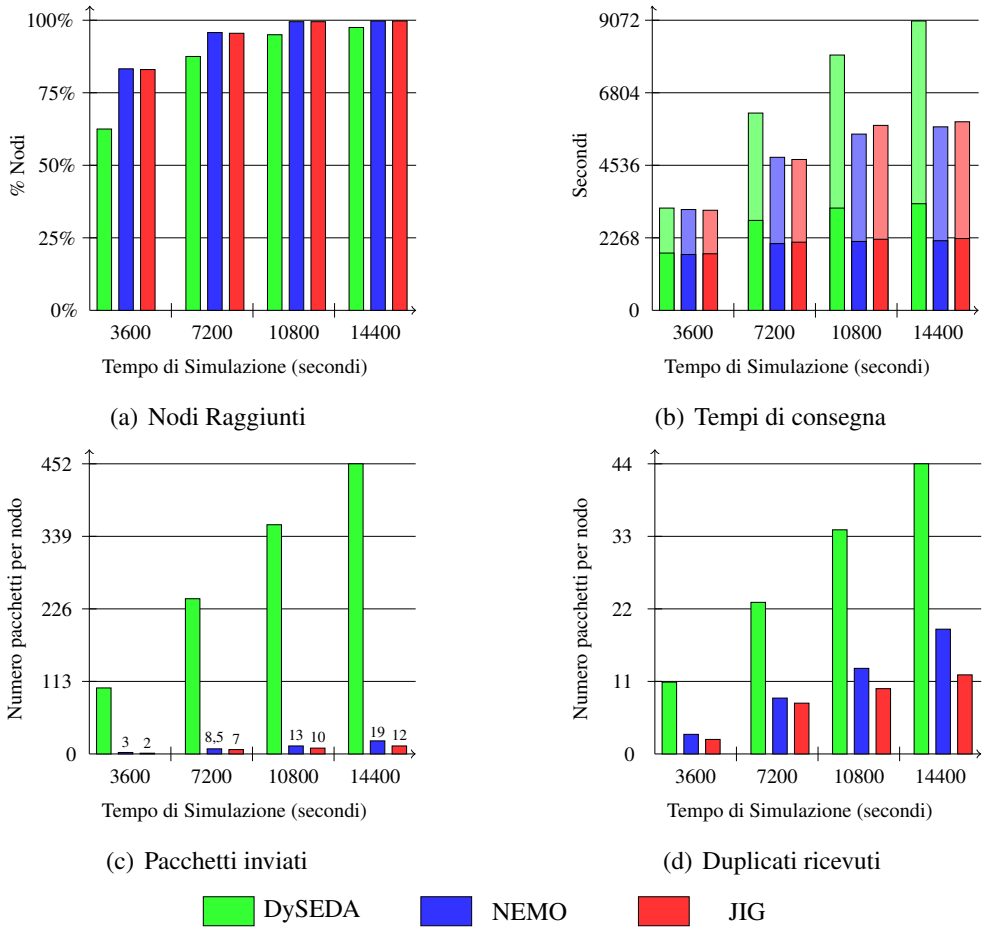
(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.7: Grafici per confrontare il protocollo JIG con i protocolli DySEDA e NEMO in un ambiente ad alta densità di nodi.



(a) Nodi Raggiunti

(b) Tempi di consegna

(c) Pacchetti inviati

(d) Duplicati ricevuti

Figura 5.8: Grafici per confrontare il protocollo JIG con i protocolli DySEDA e NEMO in un ambiente a bassa densità di nodi.

di rallentamento. Tuttavia il numero di nodi contagiati in un'ora non è soddisfacente (83%); nella seconda ora la percentuale dei nodi contagiati sale a 97%, ottimo valore se si pensa che è raggiunto con soltanto 7 pacchetti per nodo; mentre dalla terza ora in poi le performance dell'algoritmo sono ottime in quanto viene contagiata la totalità dei nodi con pochi pacchetti.

L'algoritmo JIG offre ottime performance in ambiente denso già con messaggi con lifetime pari ad un'ora. Con lifetime più lunghi la performance non peggiora; inoltre l'algoritmo è efficiente anche in ambienti poco popolati, ottenendo ottimi risultati dalle due ore in su.

5.2 Validazione con modelli di mobilità

Per verificare quanto il modello di mobilità influisce sulle performance dell'algoritmo JIG ho effettuato delle simulazioni con un modello di mobilità diverso da Random Way-Point, sviluppato in un altro lavoro di tesi ([2]).

Descriverò ora brevemente il modello di mobilità sviluppato in [2], descrivendone le caratteristiche principali e i parametri da passare in input all'eseguibile `incontro`[6] per generare i file di traccia.

L'applicazione `incontro` prende in input prende in input diversi parametri:

base la lunghezza della base dell'area espressa in metri;

altezza la lunghezza dell'altezza dell'area espressa in metri;

duratasim durata della simulazione in secondi;

numero_nodi numero dei nodi nella simulazione;

velocita m/s velocità media dei nodi;

MAXDIST_contatto raggio del range di copertura dei dispositivi;

num_setpoint numero di punti di aggregazione;

alfa_smooth parametro usato per creare un effetto di smoothing nei cambi di direzione;

[6]Per utilizzare il programma bisogna installare la libreria "GNU Scientific Library", distribuita liberamente sotto licenza GPL

beta_tanh parametro per fissare la crescita più o meno ripida della funzione tanh;
n_change_setpoint numero di cambio di set point durante la simulazione;
file_prob_massime
file_pstay file contenente le probabilità che ogni nodo ha di rimanere presso il luogo di aggregazione avendolo già raggiunto o di continuare ad inseguire un nodo avendolo già raggiunto[7];
file_intra_singoli file dove riporre le statistiche dei tempi di intracontatto fra i nodi estrapolate dal modello;
file_intercontatti file dove riporre le statistiche di intercontatto fra i nodi estrapolate dal modello;
file_output_traccia file di output dove salvare la traccia di mobilità.

GloMoSim permette di definire il movimento dei nodi dall'esterno, indicando in due file la posizione inziale dei nodi e le coordinate di ogni nodo al passare del tempo. Questi due file sono rispettivamente `nodes.input` e `mobility.in`.

Entrambi i file devono contenere delle linee che indicano rispettivamente:

```
N T (x, y, z)
```

dove `N` è l'identificativo del nodo, `T` è l'istante temporale che si sta considerando mentre la destinazione è la posizione che il nodo dovrà assumere.

Nel file `nodes.input` i tempi saranno messi tutti a 0 in quanto in questo file bisogna indicare solo le posizioni inziali di tutti i nodi.

Il modello di mobilità posiziona i nodi secondo una distribuzione uniforme nell'area.

La struttura interna di funzionamento del programma è la seguente. Innanzitutto il tempo di simulazione viene discretizzato in istanti di 200 ms. Per ognuno di questi istanti di 200 ms si calcola la prossima posizione di ogni nodo. Ciò può avvenire secondo 3 modalità

1. *movimento casuale Browniano*: vengono estratti due numeri casuali e sommati alle coordinate attuali del nodo tenendo conto conto dell'inerzia del movimento tramite il parametro $alfa$ passato in input al programma che crea un effetto

[7] queste probabilità sono costanti per tutto il periodo della simulazione

di smoothing nei cambi di direzione, per rendere i movimenti maggiormente realistici;

2. *movimento di inseguimento*: viene estratto un numero casuale che rappresenta l'ID del nodo da inseguire.

3. *movimento verso luogo aggregazione*: viene estratto un numero casuale che rappresenta l'ID del luogo di aggregazione da raggiungere.

I calcoli per gli ultimi due tipi di movimento sono i seguenti:

- viene estratto un numero casuale che rappresenta il movimento del nodo lungo la retta che collega il nodo al luogo di aggregazione o nodo da inseguire.

- viene estratto un numero casuale che rappresenta uno scostamento da questa retta per rendere più casuale il movimento e non sempre e solo lungo la retta congiungente il nodo sorgente con il nodo o luogo di destinazione

Tutti i numeri casuali coinvolti nei calcoli dello spostamento sono estratti da una distribuzione normale standard N con media 0 e deviazione standard che è calcolata in relazione alla velocità media dei nodi (infatti intuitivamente con un varianza più grande si avranno spostamenti maggiori che corrispondono a velocità medie dei nodi più alte).

È da notare che quando un nodo sta inseguendo un altro nodo o si sta muovendo verso un setpoint non interromperà il suo movimento finché non l'avrà raggiunto.

I nodi cambiano comportamento a seconda della probabilità che hanno di inseguire o aggregarsi. Queste due probabilità sono calcolate in base alle soglie di probabilità massime moltiplicate per la funzione tangente hyperbolica $tanh()$. Riporto la formula per chiarezza:

$$p[j] = pmax[j] \cdot tanh(beta \cdot (i - im[j]))$$

dove

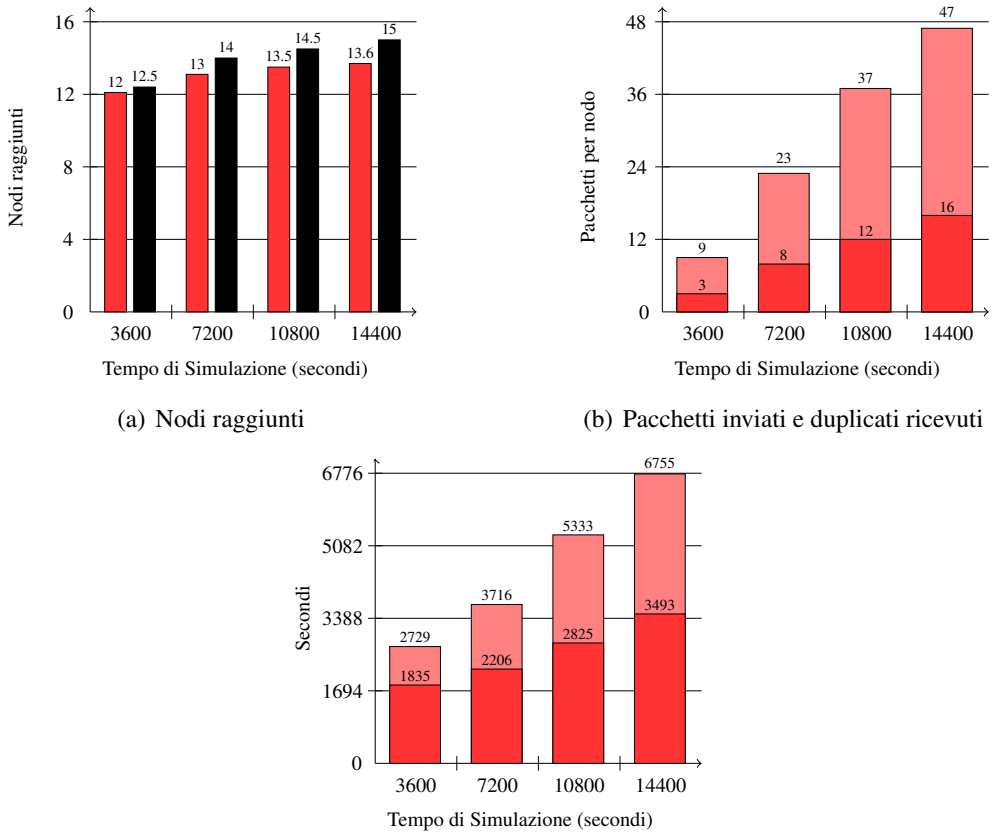
- $p[j]$ è la probabilità attuale del nodo di andare verso un luogo di incontro

- $pmax[j]$ è la soglia massima che può raggiungere la probabilità di andare verso un luogo di incontro
- $beta$ parametro per fissare la crescita più o meno ripida della funzione $tanh()$
- i è numero di iterazione corrente (l'istante attuale della simulazione si trova calcolato $(i * 0,2s)$)
- $im[j]$ è il parametro che crea il periodico "crollo" della probabilità: ogni nodo parte da un dato valore di probabilità. Questo valore cresce col tempo (la rapidità nella crescita della funzione è data dal parametro $beta$ passato in input al programma), ma senza mai poter superare la sua soglia massima (definita dal parametro $pmax$ per i luoghi di aggregazione e da $pmaxfollow$ per gli inseguimenti). Quando il nodo raggiunge il luogo di aggregazione o il nodo che inseguiva allora tale probabilità viene fatta "crollare" riportandola ad un valore molto basso. Se il nodo "decide" di abbandonare il luogo di aggregazione raggiunto o il nodo inseguito, il "crollo" della probabilità evita che venga subito ricatturato.

Dato che i luoghi di aggregazione cambiano periodicamente posizione (dipende dal parametro $n\_change\_setpoint$ passato in input), c'è una variabile che viene decrementata ad ogni iterazione che indica per quante iterazioni il luogo di aggregazione rimane fisso. Quando questa variabile raggiunge lo 0 allora si estraggono due numeri casuali da una distribuzione uniforme. Tali numeri sono le nuove coordinate del punto di aggregazione. Naturalmente esiste un vettore di luoghi di aggregazione e i calcoli precedenti sono effettuati per tutti i punti di aggregazione. Se un luogo di aggregazione cambia posizione, i nodi che lo stavano inseguendo ritornano a muoversi di moto casuale.

Per ulteriori approfondimenti sul modello di mobilità si rimanda a [2].

Attraverso un visualizzatore si è visto che, con un modello random way-point, ogni nodo ha almeno un'opportunità di contatto per ricevere l'informazione, sostenendo a pieno il presupposto introdotto da Vahdat e Becker in [8]: grazie alla mobilità dei nodi,



(a) Nodi raggiunti

(b) Pacchetti inviati e duplicati ricevuti

(c) Tempi di consegna

Figura 5.9: Nodi raggiunti, pacchetti inviati e duplicati ricevuti in un ambiente poco popolato (25 nodi in un'area di 1000x1000m).

coppie di essi entreranno in range di comunicazione periodicamente e casualmente, favorendo la distribuzione del messaggio.

Il nuovo modello di mobilità tende ad effettuare partizioni molto nette nell'area. Con questo nuovo modello alcuni nodi non hanno opportunità di contatto entro quattro ore (tempo massimo di simulazione), quindi sono praticamente irraggiungibili. Il grafico in Figura 5.9(a) rappresenta i nodi raggiunti. Il grafico presenta due istogrammi: le barre rosse rappresentano l'andamento di JIG, mentre le barre nere rappresentano i nodi realmente raggiungibili.

Per determinare quanti fossero i nodi effettivamente raggiungibili, ho eseguito delle

simulazioni con l'algoritmo Gossip con round epidemico pari a 2 secondi, utilizzando i medesimi file di mobilità utilizzati per testare l'algoritmo JIG. Il totale dei nodi raggiunti da Gossip è stato preso come massimo dei nodi effettivamente raggiungibili.

Come si vede dal grafico 5.9(a), l'algoritmo riesce sempre a raggiungere almeno il 90% dei nodi raggiungibili. Questo risultato è decisamente buono.

Il grafico in Figura 5.9(b) mostra il numero di pacchetti inviati e il numero di pacchetti ricevuti per ogni nodo. Come si nota, il numero di duplicati è circa tre volte il numero dei pacchetti inviati. Ciò sta ad indicare che in media ogni pacchetto inviato è stato ricevuto da altri tre nodi, sintomo di un'alta aggregazione all'interno delle partizioni e uno scarso cambiamento di partizione da parte dei nodi stessi - altrimenti il rapporto non sarebbe stato così preciso anche al variare del tempo di simulazione.

Infine, il grafico in Figura 5.9(c) mostra i tempi di consegna che sono stati registrati nelle simulazioni.

In definitiva, l'algoritmo JIG ha fatto registrare delle prestazioni molto buone anche con un modello di mobilità diverso da random way-point.

Capitolo 6

Conclusioni e sviluppi futuri

L'algoritmo JIG offre delle buone performance, salvaguardando risorse di rete e di sistema: esso adotta una politica reattiva in base al vicinato per determinare *quando* inviare il messaggio, mentre adotta una politica adattiva in base alle informazioni inserite nei pacchetti per sapere *se* conviene inviare il messaggio.

Il prossimo passo potrebbe essere lo studio di come variano le prestazioni dell'algoritmo variando i parametri.

Un punto fondamentale da osservare è che l'algoritmo JIG, come del resto tutti gli algoritmi epidemici, diffonde l'informazione tanto meglio quanto più i nodi si "mescolano" fra loro casualmente: è sfruttando le opportunità di contatto fra i nodi che l'informazione può essere diffusa a tutti gli host. Se il modello di mobilità crea delle partizioni nette, e i nodi stessi hanno poca probabilità di mischiarsi tra loro, la diffusione viene rallentata di molto - fino a quando un nodo si "stacca" da una partizione contagiata ed entra in una non ancora contagiata.

La politica adattiva di JIG è migliore di quella di DySEDA in quanto permette di variare l'aggressività del nodo in positivo o negativo, mentre DySEDA poteva solo aumentare il round epidemico. Questa politica ha un punto debole: presuppone che i nodi abbiano già l'informazione. Infatti ogni nodo si basa sui duplicati ricevuti per poter aggiornare la propria probabilità. JIG, oltre al controllo dei timestamp contenuti nei pacchetti duplicati, non ha altri mezzi per determinare il grado di diffusione dei messaggi, e questa è una limitazione: un nodo con probabilità minima che attraversa

una partizione non ancora contagiata non riceverà alcun pacchetto e la sua probabilità non verrà aggiornata, permettendogli di oltrepassare la partizione senza avere inviato alcun pacchetto.

Una soluzione al problema potrebbe prevedere l'inserimento nei beacon di una bitmap: essa indicherà quali messaggi possiede il mittente del beacon. In questo modo la "sincronizzazione" dei messaggi[1] potrebbe avvenire più velocemente in quanto ogni nodo conosce deterministicamente quali messaggi possiede ogni suo vicino.

L'inserimento di informazioni nei beacon ha dei pro e dei contro: sebbene permette di sincronizzare due nodi immediatamente, aumentare le dimensioni di un pacchetto beacon aumenta anche i consumi delle risorse di rete e di sistema. È necessario quindi individuare una dimensione tale da ottenere vantaggi che giustifichino l'overhead introdotto.

[1] Nel mio lavoro di tesi ho lavorato con un solo messaggio, ma potrebbero essercene tranquillamente di più. Per "sincronizzazione" dei messaggi significa che ogni coppia di host che viene in contatto determina quali messaggi in suo possesso mancano al vicino, e successivamente glieli invia.

Bibliografia

[1] Glomosim. http://pcl.cs.ucla.edu/projects/glomosim/, on line, last visit 22 aug 2007.

[2] P. Barberis. Modelli di aggregazione per la simulazione di reti opportunistiche. Tesi di laurea, Università degli Studi di Milano, 2007.

[3] V. Cerf, S. Burleigh, A. Hooke, L. Torgersonand, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay-tolerant network architecture. Internet-draft, DTN Research Group, March 2006. Expires September 2006.

[4] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[5] K. Fall. A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, New York, NY, USA, 2003. ACM Press.

[6] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[7] G. Rabbiosi. Analisi e valutazione delle prestazioni di un protocollo di gossip per reti wireless. Tesi di laurea, Università degli Studi di Milano, 2006.

[8] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks, 2000.

[9] Y. Wang, S. Jain, M. Martonosi, and K. Fall. Erasure-coding based routing for opportunistic networks. In *WDTN '05: Proceeding of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 229–236, New York, NY, USA, 2005. ACM Press.