GOSSIPICO - Un approccio *gossip-based* per la stima del numero dei nodi nelle reti dinamiche

Nicola Corti - 454413
Corso di Laurea Magistrale in Informatica - Università di Pisa

22 Aprile 2014

Sommario

Questa relazione ha lo scopo di illustrare l'algoritmo GOSSIPICO, un algoritmo gossip per il conteggio del numero dei nodi (o per altre funzioni di aggregazione) all'interno di una rete. GOSSIPICO ha in più, rispetto ad altri algoritmi di conteggio, un'elevata robustezza che lo rende adatto ad operare all'interno di reti che presentino un elevato grado di dinamicità.

Indice

Introduzione

Il conteggio del numero dei nodi di una rete è sempre stato uno dei problemi più affrontati quando si prendono in considerazione le reti decentralizzate, dove non è presente un nodo server che raccoglie le connessioni di tutti i client. In una rete di questo genere difficilmente ogni singolo nodo avrà la visione globale di tutti gli altri nodi della rete, in particolare se il numero dei nodi della rete diventa elevato.

Conoscere però questo valore potrebbe essere comunque fondamentale per molte applicazioni che potrebbero ottimizzare i loro parametri di esecuzione (memoria da allocare, numero e frequenza dei messaggi da inviare, etc...).

Per effettuare questo genere di calcolo sono stati realizzati vari modelli con punti di forza e di debolezza differenti. Fra questi uno che merita di essere menzionato è il modello *gossip*. Gli algoritmi *gossip* effettuano fondamentalmente un continuo scambio di informazioni fra nodi "vicini"[1] al fine di approssimare sempre più il

[1]Per vicini non si intendono i nodi fisicamente vicini, ma i nodi appartenente al sottoinsieme dei nodi della rete noti al singolo peer

numero dei nodi del sistema. I vari modelli di algoritmi, ed in particolare il modello *gossip*, sono introdotti nella sezione 1.

GOSSIPICO (presentato nella sezione 2) rappresenta un esempio di protocollo *gossip* per il calcolo dei nodi di una rete. GOSSIPICO si basa fondamentalmente su due moduli che coesistono e funzionano in armonia al fine di velocizzare il calcolo: il modulo COUNT (sezione 3) si occupa di effettuare il conteggio vero e proprio, conservando in ogni nodo le informazioni sull'approssimazione finora raggiunta dall'algoritmo, mentre il modulo BEACON (sezione 4) si occupa di individuare in modo casuale dei nodi di riferimento (detti appunti nodi *beacon*) verso cui veicolare i messaggi al fine di velocizzare il processo di conteggio.

Uno dei punti di forza di GOSSIPICO sta nel fatto che l'algoritmo si adatta molto bene a reti che sono dinamiche, con nodi che si connettono e si disconnettono nel tempo. Gli algoritmi di conteggio classici presentano infatti delle criticità nel caso in cui un nodo si disconnetta dalla rete. Un ulteriore scenario si presenta se la disconnessione di un singolo nodo porta alla disconnessione dell'intera rete in due componenti distinte. GOSSIPICO affronta queste difficoltà tramite alcune accortezze (sezione 5) che gli permettono di affrontare senza troppe difficoltà i conteggi su reti dinamiche.

Per poter valutare le performance dell'algoritmo è stato realizzata un'implementazione dell'algoritmo utilizzando il simulatore PeerSim ([3]). È possibile conoscere i comandi necessari per far funzionare la simulazione leggendo la user guide (sezione 9) mentre i risultati delle simulazioni su varie tipologie di rete sono raccolti nella sezione perfomance (sezione 8).

1 Gli algoritmi *gossip-based*

Alcune delle operazioni che potrebbero risultare banali in una rete organizzata con un paradigma *client-server* possono presentare alcune criticità se considerate all'interno di una rete *peer-to-peer*.

Il conteggio del numero dei nodi risulta essere una delle prime, ma si pensi anche alla distribuzione di un'informazione a tutti i nodi della rete (ad esempio una nuova release di un software), oppure al recupero di informazioni sullo stato globale dei nodi stessi (quanti nodi si sono disconnessi, etc...). In questo contesto risulta necessario disporre di algoritmi che siano scalabili, efficienti e che al contempo non basino il loro funzionamento su un nodo centrale che potrebbe disconnettersi improvvisamente.

È in questo contesto che sono nati gli algoritmi *gossip* anche detti algoritmi *epidemici*.

Per comprendere il principio che fonda le basi di questa classe di algoritmi si pensi al modo con cui si propagano i pettegolezzi in un gruppo di persone oppure al modo con cui si propaga un'infezione virale: casualmente un soggetto infettato incontra un altro soggetto suscettibile e lo infetta.

Risulta chiaro come questo meccanismo porti a lungo termine ad uno stato in cui tutti i soggetti sono infettati, ovvero tutti i nodi hanno raccolto l'informazione presente nella rete.

Ogni soggetto può trovarsi in una serie di stati differenti e transire verso un altro stato in base a come interagisce con gli altri nodi e con l'informazione:

susceptible Rappresenta un soggetto che non è ancora stato coinvolto dall'informazione,

infected Rappresenta un soggetto che è stato coinvolto e che sta diffondendo l'informazione,

recovered Rappresenta un soggetto che non è più interessato a diffondere l'informazione.

Inoltre in base a come vengono svolte le comunicazioni si possono individuare protocollo di tipo differente:

push Il soggetto che ha stabilito la comunicazione invia l'informazione che sta mantenendo,

pull Il soggetto che ha stabilito la comunicazione raccoglie l'informazione dal soggetto che ha contattato,

push/pull Il soggetto che ha stabilito la comunicazione scambia le proprie informazioni con il soggetto che ha contattato.

1.1 Algoritmi per il conteggio dei nodi

In passato sono stati presentati altri modelli per il conteggio dei nodi di una rete, in particolare si possono raggruppare i modelli proposti fra:

- Algoritmi basati su *probabilistic polling*, che stimano la dimensione della rete in base alla risposta ad una richiesta inviata da un nodo,
- Algoritmi basati sul *random walk*, che stimano la dimensione della rete in base al numero di archi percorsi da un messaggio che segue un percorso casuale fra i nodi,
- Algoritmi basati sul *gossip*, in cui ogni nodo possiede una valore e si procede ad approssimare il conteggio effettuando delle medie fra i valori ad iterazioni successive.

GOSSIPICO rappresenta un esempio di algoritmo appartenente a quest'ultima classe, offrendo però il meccanismo BEACON per velocizzare il calcolo della stima.

2 GOSSIPICO

L'algoritmo GOSSIPICO ([1]) nasce presso l'Università di Delft con lo scopo primario di realizzare un algoritmo *gossip-based* che permetta di realizzare il calcolo di funzioni di aggregazione su reti decentralizzate. Il protocollo originariamente presentato in [1] permette di effettuare solamente il calcolo dei nodi, ma può essere facilmente

esteso permettendo il calcolo di altre funzioni di aggregazione (in particolare della somma, del massimo, del minimo e della media).

GOSSIPICO è formato da due moduli:

COUNT Che effettua la fase di conteggio vera e propria. Si occupa di raccogliere i messaggi e di "combinarli" fra di loro al fine di portare la rete verso lo stato di convergenza.

BEACON Permette di velocizzare la fase di COUNT, in particolare organizzando la rete in eserciti (*Army*) che si combattono al fine di determinare un unico vincitore. In questo modo i messaggi della rete saranno veicolati verso il vincitore che si occuperà di svolgere il ruolo di raccolta dell'informazione globale.

Si noti che il modulo di COUNT potrebbe funzionare anche in modalità *standalone*, ma il modulo di BEACON velocizza notevolmente il calcolo (vedi sezione 8).

L'algoritmo è stato implementato tramite il simulatore Peersim utilizzando un insieme di classi che verranno presentate nel seguito e che sono sintetizzate nel seguente diagramma delle classi (immagine 1). Si noti che non sono presenti tutti gli attributi e tutti i metodi delle classi poiché sono stati rappresentati solamente i più significativi al fine di non appesantire troppo la rappresentazione.

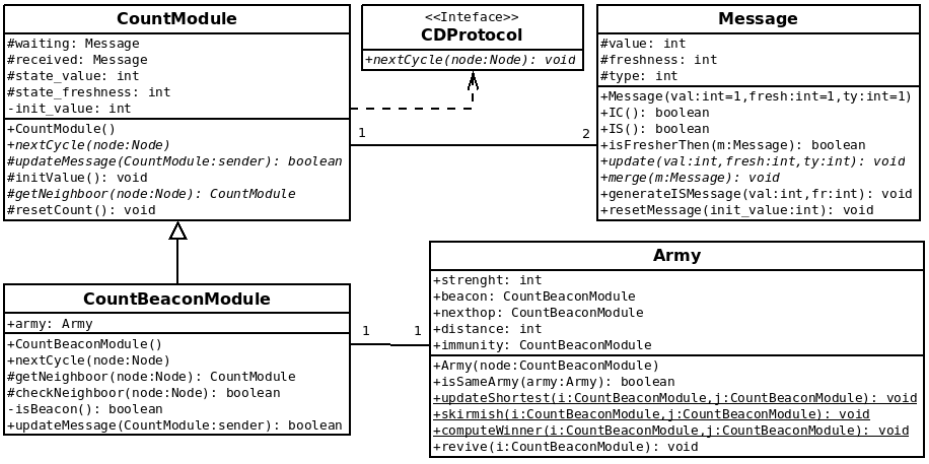


Immagine 1: Diagramma delle classi realizzate

Le classi sono state realizzate implementato l'interfaccia CDProtocol, ovvero un protocollo Peersim che procede a cicli successivi. CDProtocol rappresenta una delle due interfacce offerte da Peersim per implementare un protocollo, insieme all'interfaccia EDProtocol (protocollo che funziona tramite eventi); si è scelta la prima interfaccia in quanto risulta più naturale nell'implementare l'algoritmo GOSSIPICO poiché si dispone della descrizione del comportamento ciclo per ciclo.

3 Il modulo COUNT

Il modulo COUNT è realizzato dalla classe CountModule che esegue ad ogni ciclo la seguente computazione:

1. Contatta uno dei vicini scelto casualmente,

2. Invia il proprio messaggio in attesa,

3. Genera un nuovo messaggio contenente l'informazione finora raccolta.

Quando invece un nodo riceve un messaggio in input, esso aggiorna il proprio messaggio in attesa in funzione del messaggio appena ricevuto.

Per comprendere a fondo il funzionamento di COUNT risulta però necessario conoscere la struttura dei messaggi inviati e ricevuti da ogni nodo.

3.1 Struttura dei messaggi

Ogni nodo durante tutto il calcolo riceve ed invia costantemente messaggi che hanno la seguente struttura

$$< C, F, T >$$

$C \in \mathbb{Z}$ Rappresenta il valore attualmente contenuto del messaggio, ovvero l'approssimazione che finora è stata calcolata

$F \in \mathbb{N}$ Indica quanto è recente l'informazione contenuta nel messaggio, a freschezza maggiore corrisponde un messaggio più recente

$T \in \{0, 1\}$ Rappresenta il tipo del messaggio che può essere di *Information Spreading* (IS, $T = 0$) oppure di *Information Collecting* (IC, $T = 1$).

In particolare il tipo del messaggio rappresenta la natura dell'informazione contenuta al suo interno:

IS Indica un messaggio che sta diffondendo informazioni sulla rete. In particolare ogni nodo periodicamente genera un messaggio di tipo IS, al fine di informare la rete del dato che finora ha raccolto,

IC Indica un messaggio che contiene dell'informazione che deve essere ancora raccolta. I nodi della rete daranno maggiore priorità a questi messaggi rispetto ai messaggi IS. Quando due messaggi IC si incontrano essi verranno combinati al fine di accumulare sempre più informazione.

La computazione comincia inizializzando tutti i nodi della rete con messaggi formati nel modo seguente $\{1, 1, 1\}$[2]. I messaggi inizieranno a fluire nella rete ed i messaggi IC che si incontrano verranno combinati, fin quando non si sarà raccolta tutta l'informazione presso un singolo nodo, avendo dunque un solo messaggio IC.

[2]Questa inizializzazione vale solamente per il caso in cui si effettui il conteggio dei nodi

3.2 Regole di *message-combining*

Ogni nodo conserva al proprio interno uno stato formato da:

M_r Ovvero il messaggio appena ricevuto dal peer. I nodi utilizzano M_r per conservare il messaggio appena ricevuto e lo confrontano insieme al messaggio in attesa (M_w) per calcolare un nuovo messaggio in attesa.

M_w Ovvero il messaggio in attesa. Ogni nodo invia nel proprio ciclo di esecuzione il messaggio M_w ad un altro nodo vicino scelto in modo casuale.

C_s Rappresenta il valore attualmente conservato dallo stato del nodo.

F_s Rappresenta la freschezza più alta attualmente vista dal nodo.

Dopo ogni invio un nodo utilizza i valori C_s e F_s per generare un nuovo messaggio $\{C_s, F_s, 0\}$ generando dunque un messaggio di tipo IS per informare la rete sull'informazione finora raccolta.

Quando un nodo riceve un messaggio, confronta i messaggi M_r e M_w utilizzando le seguenti regole:

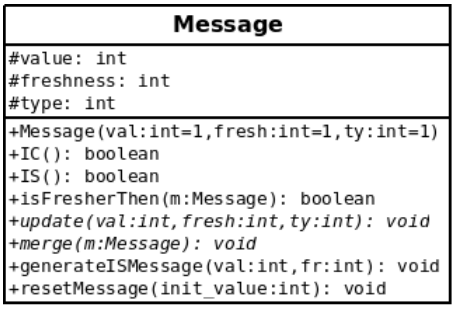
- $M_r.Tipo$ = IS e $M_w.Tipo$ = IS, si aggiorna M_w al messaggio che contiene il valore di $Freschezza$ più elevato,
- $M_r.Tipo$ = IC e $M_w.Tipo$ = IS, si da priorità all'informazione contenuta dentro il messaggio IC, per cui si imposta $M_w \leftarrow M_r$,
- $M_r.Tipo$ = IS e $M_w.Tipo$ = IC, si da sempre priorità all'informazione contenuta dentro il messaggio IC, per cui si scarta il messaggio M_r,
- $M_r.Tipo$ = IC e $M_w.Tipo$ = IC, in questo caso si effettua il *combining* fra i due messaggi IC al fine di generare un nuovo messaggio IC formato nel modo seguente

$$\{M_r.C + M_w.C, M_r.F + M_w.F, 1\}$$

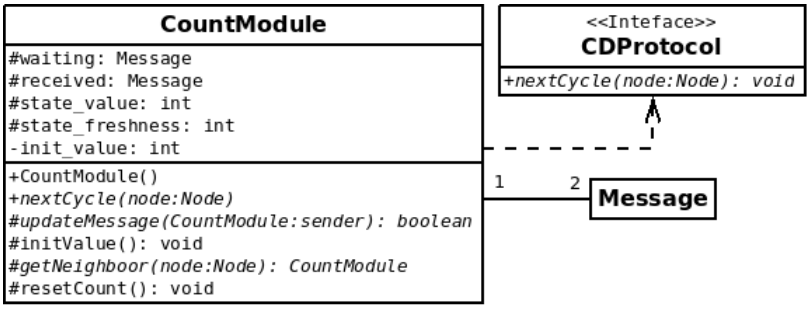
Contestualmente a queste operazioni, il nodo che riceve un messaggio provvede anche ad aggiornare i propri valori C_s e F_s in modo da mantenere le proprie informazioni le più aggiornate possibile.

Grazie a questo meccanismo il numero totale dei messaggi IC all'interno della rete tende a decrescere fino ad arrivare ad un singolo messaggio che conterrà l'informazione aggregata.

Le variabili C_s e F_s sono necessarie in quanto il nodo ad ogni ciclo di esecuzione, dopo aver inviato il proprio messaggio in attesa, genererà un nuovo messaggio in attesa di tipo IS partendo dalle informazioni sullo stato, che verrà inviato al ciclo successivo per informare la rete sulle informazioni da lui raccolte finora.



(a) Classe Message



(b) Classe CountBeacon

Immagine 2: Diagramma delle classi relative al modulo COUNT

3.3 Implementazione

I messaggi sono realizzati dalla classe Message che dispone di tutti i metodi per la gestione dei messaggi (immagine 2a), mentre il modulo COUNT è realizzato dalla classe CountModule (immagine 2b).

In particolare la classe Message, oltre ad una serie di costruttori base, offre i metodi IC() ed IS() per verificare il tipo di un messaggio, isFresherThen() per confrontare la freschezza, update(val, fresh, ty) per aggiornare tutti i campi di un messaggio (usato per l'invio di messaggi), merge(m) per calcolare la combinazione di due messaggi, generateISMessage(val, fr) per generare un nuovo messaggio IS e il metodo reset(val) per reimpostare il valore di un messaggio.

La classe CountModule dispone di due field di tipo Message che rappresentano rispettivamente i messaggi M_w e M_r, dispone inoltre delle variabili dello stato C_s e F_s.

Per quanto riguarda i metodi, la classe CountModule implementa il metodo nextCycle(node) come richiesto dall'interfaccia CDProtocol di Peersim, in cui è presente tutta la logica del singolo nodo. Sono presenti inoltre altre funzioni di comodo quali:

updateMessage(sender) Per aggiornare il messaggio M_w in base alle regole descritte nella sezione 3.2,

initValue() Per impostare il valore iniziale C_i con cui iniziare il calcolo,

getNeighboor() Per scegliere in modo casuale un vicino dalla rete offerta da Peersim e configurata nel file di configurazione,

resetCount() Per riportare il messaggio M_w allo stato iniziale.

Si faccia particolare attenzione anche al metodo clone(), override dalla classe Object, in quanto Peersim utilizza questo metodo per generare nuove istanze dei singoli protocolli.

4 Il modulo BEACON

GOSSIPICO prevede che oltre al modulo COUNT sia presente anche un modulo BEACON. Il modulo BEACON serve per "guidare" i nodi nell'invio dei messaggi IC verso un nodo unico (che si chiamerà appunto *beacon*, dall'inglese faro) in modo da velocizzare il processo di *message-combining* dei messaggi che raccolgono l'informazione.

Si pensi al caso in cui nella rete rimangono solamente due messaggi di tipo IC: la convergenza viene raggiunta quando i due messaggi si incontrano presso uno stesso nodo. Il tempo necessario per raggiungere questa situazione è approssimabile con quello necessario a due random walk che si incontrano su uno stesso nodo. Risulta evidente che su reti di dimensioni molto elevate questo meccanismo può portare a messaggi IC che si muovono sulla rete e che si incontrano con bassa probabilità.

Per questo si è deciso di considerare ogni nodo della rete come facente parte di un esercito. Ogni esercito avrà a capo un *beacon* che sarà responsabile di raccogliere i messaggi IC dei membri del proprio esercito. Ogni esercito relativo ad un nodo i dispone di:

A_i L'ID dell'esercito, necessario per identificare l'esercito di appartenenza,

S_i La forza dell'esercito,

D_i La distanza in termini di hop verso il *beacon* dell'esercito,

P_i Il riferimento al prossimo nodo (*next-hop*) verso il *beacon* dell'esercito.

La rete verrà inizializzata in modo che ogni nodo formi un proprio esercito di cui lui stesso è il *beacon*, la forza viene generata in modo casuale (assumendo che due nodi non possano avere la stessa forza, ovvero $\mathbb{P}(S_i = S_j) = 0$ se $i \neq j$), la distanza verso il *beacon* viene posta a zero, e come *next-hop* si imposta il nodo stesso:

$$\{A_i = i, S_i = rnd(), D_i = 0, P_i = i\}$$

Ad ogni ciclo dell'iterazione, con una probabilità casuale, i vari nodi contatteranno un altro nodo (in modo casuale) ed effettueranno una schermaglia (*skirmish*) da cui verrà proclamato un vincitore in base alla forza dei vari eserciti. Questo porterà a convergere verso un unico *beacon* (il nodo che in principio aveva il valore di S_i più elevato) che disporrà di tutti i nodi della rete come membri del proprio esercito.

4.1 Il meccanismo delle schermaglie

Nel momento in cui un nodo i contatta un altro nodo j possono avvenire due episodi differenti in base agli eserciti di appartenenza dei due nodi.

Se entrambi i nodi appartengono allo stesso esercito, viene semplicemente aggiornata la distanza fra i due verso il *beacon*, in modo da mantenere i percorsi verso il *beacon* i più brevi possibile.

Se i e j appartengono a due eserciti differenti, si calcola quale dei due eserciti è vincente in base ai valori S_i e S_j. Supponiamo che $S_i > S_j$ in tal caso il nodo j entra a far parte dell'esercito di j e il nodo i diventa il *next-hop* del nodo j:

$$\{A_j = A_i, S_j = S_i, D_j = D_i + 1, P_j = P_i\}$$

Quando un nodo entra a far parte di un altro esercito viene invocato il processo di *reset* del modulo COUNT su quel nodo ovvero viene reimpostato il messaggio M_w al valore $\{1, 1, 1\}$.

4.2 Variazioni sul modulo COUNT

L'utilizzo del modulo BEACON comporta alcune variazioni al modulo COUNT al fine di poter beneficiare a pieno della presenza di un nodo *beacon*.

Primo fra tutti, i nodi non invieranno più i messaggi M_w ad un nodo scelto in modo casuale (come invece descritto in sezione 3), ma effettueranno una scelta più oculata: nel caso in cui un nodo non sia il *beacon* del proprio esercito, esso invierà i messaggi di tipo IC al nodo P_i ovvero al *next-hop* nei confronti del *beacon*; in tutti gli altri casi (sia se il nodo è il *beacon* oppure se il messaggio da inviare è di tipo IS) viene scelto un vicino in modo casuale.

Inoltre si configurano i nodi per rifiutare messaggi che non provengono dal proprio esercito. Infine risulta necessario prevedere la procedura di reset del conteggio ogni volta che un nodo entra a far parte di un nuovo esercito.

4.3 Implementazione

Il modulo BEACON è realizzato tramite la classe Army per rappresentare un esercito e la classe CountBeaconModule. Come si può facilmente immagine CountBeaconModule risulta essere una sottoclasse di CountModule che ridefinisce ed amplia alcuni dei metodi di quest'ultima.

Entrambi le classi sono rappresentate nel diagramma UML nell'immagine 3.

La classe CountBeaconModule ridefinisce in particolare il metodo nextCycle, introducendo la computazione BEACON, che viene effettuata con una probabilità $\mathbb{P} = 0.50$. Inoltre il metodo getNeighboor viene ridefinito considerando il caso in cui il nodo debba inviare un nodo IC, in modo da instradarlo verso il *beacon*. Il metodo updateMessage viene ridefinito in modo da rifiutare i messaggi M_r che sono stati ricevuti da nodi che non appartengono all'esercito A_i del nodo.

Per quanto riguarda la funzione checkNeighboor si faccia riferimento alla sezione 5.

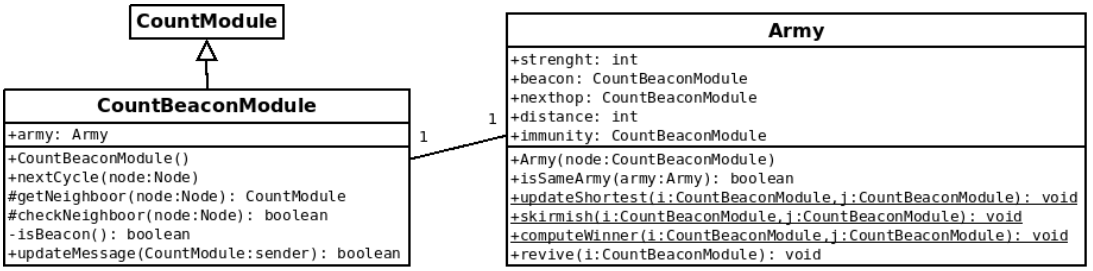


Immagine 3: Diagramma UML delle classi Army e CountBeaconModule

La classe Army offre invece i metodi per gestire gli eserciti e i combattimenti fra eserciti: in particolare skirmish per calcolare l'esito di una schermaglia, updateShortest per aggiornare la distanza fra due nodi dello stesso esercito e computeWinner per far entrare un nodo perdente all'interno di un altro esercito più forte.

Per la funzione revive vale un discorso analogo a quello di checkNeighboor (vedi sezione 5).

Per far funzionare al meglio l'algoritmo BEACON è inoltre necessario che venga eseguito CountBeaconInitializer, un inizializzatore (ovvero un'implementazione dell'interfaccia Control di Peersim) che permette di impostare i valori di forza ai nodi in modo casuale ideale, facendo in modo che non possano esistere due nodi con lo stesso valore di S_i.

5 Reti dinamiche

GOSSIPICO, come già annunciato nella sezione 2, è stato pensato per adattarsi al meglio alle situazioni di dinamicità.

Può infatti succedere che all'interno della rete un nodo si disconnetta. La disconnessione può essere volontaria, nel caso in cui un nodo decida di sua spontanea volontà di abbandonare le rete, oppure involontaria, nel caso in cui un nodo non riesca più a connettersi ad esempio per problemi sulla rete fisica.

GOSSIPICO prevede che i nodi che si rendano conto di un nodo che si è disconnesso diano luogo ad una "rivoluzione" effettuando un ricalcolo, in modo da far ripartire il calcolo alla luce del nodo appena disconnesso. Il nodo effettua quindi un reset del proprio modulo COUNT in modo analogo a quanto descritto in sezione 4.1. Viene poi reinizializzato l'esercito del nodo con nuovi valori (quindi un nuovo valore di $S_i = rnd()$) ed impostato il nodo come *beacon* dell'esercito.

Una situazione di questo genere non risulta però essere la soluzione adatta a risolvere il problema, in quanto il nodo i potrebbe essere inglobato dall'esercito attualmente al potere A_j, andando quindi ad interrompere l'operazione di ricalcolo. Per evitare che ciò accada si aggiunge un nuovo campo all'esercito di ogni nodo: il campo I_i che rappresenta l'*Immunità*:

$$I_i = j \qquad \text{se } i \text{ risulta immune a } j$$

Introducendo l'*Immunità* si deve anche aggiornare il meccanismo delle schermaglie, prevedendo che, nel momento in cui un nodo i contatta un nodo j, nel caso

in cui sia impostato $I_i = j$ il nodo i sia dichiarato automaticamente vincitore dello scontro (e viceversa).

Essendo l'*Immunità* un nuovo campo dell'esercito, esso viene trasmesso anche ai nuovi nodi che entrano a far parte dell'esercito.

5.1 Implementazione

Per implementare il supporto alle reti dinamiche, si può notare che sono presenti alcuni metodi all'interno delle classi CountBeaconModule e Army (immagine 3). In particolare checkNeighboor permette di controllare se tutti i vicini sono sempre up, oppure se qualche nodo si è disconnesso. Nel caso in cui un nodo si accorga che un nodo si è disconnesso esso invoca il metodo revive sul proprio esercito ed imposta il valore del campo immunity in modo da effettuare un ricalcolo.

6 Interazione

Come si è potuto evincere dalle sezioni 3 e 4 le interazioni fra i due moduli COUNT e BEACON sono notevoli, ogni modulo funziona grazie alle informazioni raccolte dall'altro.

Le interazioni fra i due moduli possono essere riassunte nell'immagine 4

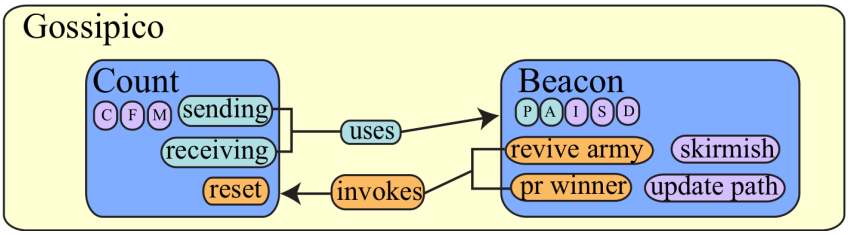


Immagine 4: Schema delle interazioni fra i due moduli, tratto da [1]

7 Ulteriori Classi

Oltre alle classi finora presentate, il software è stato corredato da altre classi di comodo per effettuare simulazioni più strutturate.

7.1 Altre funzioni di aggregazione

L'algoritmo di COUNT, come già anticipato, può essere utilizzato per effettuare il calcolo di altre funzioni di aggregazione. Sono state implementate le funzioni di somma, massimo e minimo. In particolare è possibile scegliere la funzione da usare tramite il parametro `.func` nel file di configurazione: i valori ammissibili sono `count`, `sum`, `min` e `max`.

Per realizzare queste funzioni è stato sufficiente definire delle sottoclassi di Message, in particolare le classi MinMessage e MaxMessage (immagine 5), in cui si va a fare l'override del metodo merge(m) in modo da andare a definire quale valore

deve essere conservato quando due messaggi IC si incontrano. Per quanto riguarda la funzione di somma è stato sufficiente impostare il valore iniziale C_w del primo messaggio in attesa. Tale valore può essere impostato tramite il parametro `.value` nel file di configurazione.

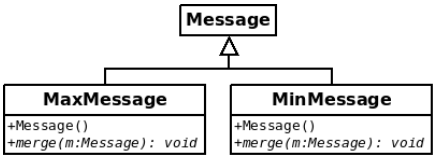


Immagine 5: Diagramma UML delle classi MinMessage e MaxMessage

Con GOSSIPICO è possibile effettuare anche il calcolo della funzione della media, risulta però necessario ampliare il messaggio con due valori V_w dove si mantiene la somma dei valori e C_w dove si mantiene il conteggio dei nodi finora visti. Per implementare questa funzione non sarebbe sufficiente una classe AvgMessage con un campo ulteriore, ma sarebbe necessario ampliare anche la classe CountModule (o CountBeaconModule) prevedendo una nuova variabile di stato e fare l'override di tutti i metodi che coinvolgono la variabile di stato.

7.2 Inizializzatori

Nel caso in cui si voglia utilizzare il calcolo tramite la funzione della somma, del massimo o del minimo è possibile dare il valore in input tramite il parametro `.value` come presentato poco fa. In questo modo però ogni nodo condivide lo stesso valore, se si volesse invece utilizzare dei valori differenti per ogni nodo si può utilizzare un inizializzatore.

Questi inizializzatori implementano l'interfaccia Control di Peersim e possono essere eseguiti prima dell'esecuzione della simulazione; in particolare sono disponibili i seguenti inizializzatori:

LinearInitializer Assegna ad ogni nodo un valore che va da 0 ad $n-1$, dove n è il numero dei nodi della rete,

PeakInitializer Assegna ad ogni nodo il valore 0, tranne al primo nodo a cui assegna il valore impostato dal parametro `.value` nel file di configurazione,

RandomInitializer Assegna valori casuali ad ogni nodo. I valori sono compresi fra gli estremi `.min` e `.max` impostati nel file di configurazione.

7.3 Raccolta di statistiche

Per la raccolta delle statistiche sull'esecuzione è possibile utilizzare la classe Statistics (implementazione di Control). Questa classe offre informazioni relative alla situazione della rete quali il numero dei messaggio IC, IS ed il numero dei *beacon* presenti nella rete.

Inoltre visualizza a quale ciclo vengono raggiunte le 3 fasi descritte nella sezione 8.

Nel caso di computazioni con molti cicli, se non si fosse interessati ad avere tutte le informazioni sui messaggi ad ogni ciclo, ma si è solamente interessati a sapere quanti cicli sono necessari all'algoritmo per convergere, si può impostare il parametro `.silent` su `true` nel file di configurazione (di default è su `false`).

7.4 Debugging

Per effettuare il debug è possibile usare la classe Debugger (implementazione di Control) che stampa ad ogni ciclo la situazione attuale di ogni nodo fornendo informazioni su tutto il suo stato (sia per quanto riguarda la parte COUNT che la parte BEACON).

8 Performance

Andando ad analizzare a fondo l'algoritmo si può notare che il calcolo procede in 3 fasi distinte.

1. Si svolge la lotta per eleggere un singolo *beacon*. Il modulo COUNT funziona correttamente, ma vengono invocati dei *reset* ogni volta che un nodo entra a far parte di un nuovo esercito.

2. È stato individuato un unico *beacon*, che adesso avrà il compito di raccogliere tutti i messaggi IC della rete e ricombinarli. È possibile che avvengano altre ricombinazioni in nodi intermedi prima di arrivare direttamente al *beacon*, ma noi assumiamo al caso pessimo che arrivino $n - 1$ messaggi IC al *beacon* dove n è la dimensione della rete.

3. Nella rete è presente un singolo messaggio di tipo IC che contiene tutta l'informazione. Adesso l'informazione deve essere condivisa presso gli altri nodi attraverso i messaggi IS.

Per poter stimare queste tre fasi introduciamo la conduttanza ϕ (presentata in [2]) come misura del grado globale di connessione della rete ($0 < \phi < 1$ e vale 1 nel caso di un grafo completamente connesso). In [2] si nota come la velocità con cui un'informazione si diffonde all'interno di una rete sia proporzionale a $O(\phi^{-1}log(\mathrm{N}))$ dove N è il numero di nodi della rete.

Per quanto riguarda la prima e la terza fase, possiamo ricondurle facilmente alla diffusione di un'informazione su una rete, per cui vale l'approssimazione $O(\phi^{-1}log(\mathrm{N}))$. Per la seconda fase consideriamo invece il caso pessimo in cui tutti i messaggi IC si incontrino presso il nodo *beacon*; anche in questo caso il percorso più lungo verso il *beacon* risulta essere dell'ordine di $O(\phi^{-1}log(\mathrm{N}))$ per cui questo limite si applica anche al secondo caso.

In media si nota che l'algoritmo GOSSIPICO impiega $O(log(\mathrm{N}))$ cicli per giungere alla convergenza, dimostrandosi dunque un ottimo algoritmo per il calcolo di funzioni di aggregazione, in grado di affrontare anche reti di dimensioni elevate.

8.1 Grafici delle performance

Per poter apprezzare a fondo GOSSIPICO ed in particolare l'utilità del modulo BEACON sono stati realizzati alcuni test sperimentali di cui si riportano i risultati nelle sezioni seguenti.

8.1.1 Evoluzione del numero di messaggi IC

Un parametro che risulta interessante analizzare è il numero di messaggi di tipo IC presenti ad ogni iterazione nella rete. Sappiamo infatti che l'informazione è stata correttamente raccolta ed aggregata quando sarà presente nella rete solamente un messaggio di tipo IC. Sono stati quindi monitorati il numero di messaggi IC in una simulazione con un grafo random formato da 1000 nodi.

La simulazione è stata realizzata utilizzando sia l'algoritmo COUNT che l'algoritmo COUNT-BEACON.

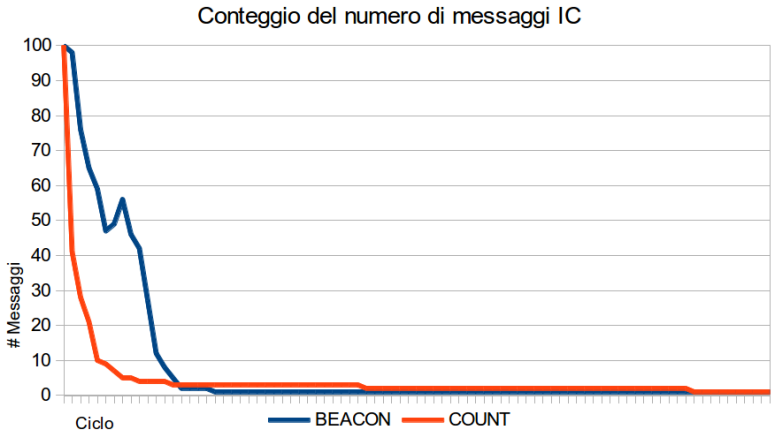


Immagine 6: Grafico che rappresenta il numero di messaggi IC presenti nella rete

I risultati sono mostrati nell'immagine 6, si può notare come il numero dei messaggi decresca in entrambi i casi, ma decresce in modo più immediato il numero nel caso dell'algoritmo COUNT.

Questo fenomeno deriva dal fatto che l'algoritmo BEACON fa effettuare dei *reset* ai nodi che sono stati appena inseriti in un nuovo esercito. Quando si effettua un *reset* il nodo viene reimpostato con il messaggio M_w su $\{1, 1, 1\}$ che contribuisce ad aumentare il conteggio dei messaggi IC.

Se si osserva attentamente il grafico si può notare che, correttamente, l'utilizzo del modulo BEACON velocizza la convergenza (che in questo caso viene raggiunta in circa 50 cicli), mentre l'algoritmo COUNT porta ad uno stato in cui sono presenti pochi messaggi IC che viaggiano in modo casuale nella rete. Questa situazione permane per molti cicli fino a quando i messaggi IC non si incontrano presso un singolo nodo (in questo caso all'incirca dopo 600 cicli).

8.1.2 Velocità di convergenza

Un altro parametro fondamentale da misurare risulta essere il numero di cicli necessari all'algoritmo per raccogliere tutta l'informazione e distribuirla a tutti i nodi.

Sono stati effettuati vari test variando algoritmo e topologia di rete:

- Algoritmo COUNT con grafi random.
- Algoritmo COUNT-BEACON con grafi random.
- Algoritmo COUNT-BEACON con grafi *small-world* di Watts and Strogatz.
- Algoritmo COUNT-BEACON con grafi *scale-free* di Barabasi-Albert.

Sono state dapprima effettuate simulazioni su reti di piccole dimensioni (immagine 7) in cui si può notare come, già su reti di dimensioni superiori a 100 nodi, l'aggiunta del modulo BEACON velocizza notevolmente il calcolo.

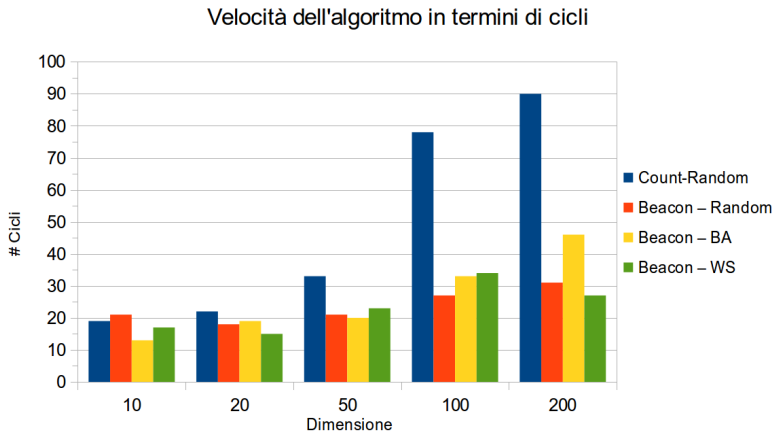


Immagine 7: Grafico che rappresenta la velocità di convergenza su reti di piccole dimensioni

I risultati davvero interessanti si ottengono però quando si effettuano simulazioni su reti di medie o grandi dimensioni (immagine 8), in cui il modulo BEACON permette un notevole miglioramento delle performance. Si noti che nell'immagine 8 la scala sull'asse y risulta essere logaritmica in modo da rappresentare al meglio il miglioramento introdotto.

8.1.3 Evoluzione in caso di disconnessione di nodi

Un'ultima simulazione che risulta molto interessante da analizzare è quella che rappresenta l'evoluzione della rete nel caso in cui un nodo si disconnetta.

In particolare si va ad analizzare la stima della dimensione della rete calcolata nel modo seguente:

$$S = \frac{\sum_{i=1}^{k} M_i.C}{k} \tag{8.1.1}$$

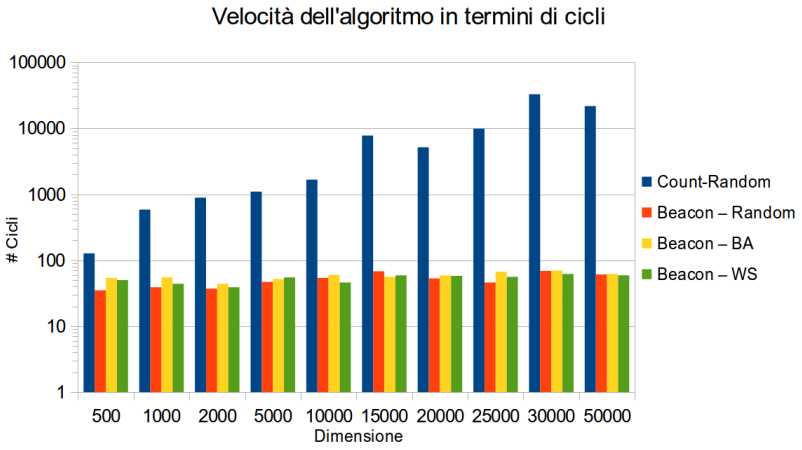


Immagine 8: Grafico che rappresenta la velocità di convergenza su reti di medie e grandi dimensioni

Dove $k = Network.size()$.

Nei grafici nell'immagine 9 si va ad analizzare la differenza fra la stima calcolata e la dimensione effettiva della rete.

Nel grafico in immagine 9a le disconnessioni sono distanziate nel tempo, per cui si nota chiaramente che l'algoritmo riesce a raggiungere senza problemi il valore di dimensione della rete effettivo.

Nel grafico in immagine 9b sono invece presenti tre disconnessioni ravvicinate. Si nota come la rete si adatti a queste disconnessioni: viene fatto partire un riconteggio e la stima tende a riallinearsi al valore effettivo; in prossimità dell'ultima disconnessione viene fatto partire un nuovo riconteggio (si noti come la linea blu nella parte centrale decresce leggermente) e l'algoritmo riesce comunque ad approssimare comunque l'effettiva dimensione della rete.

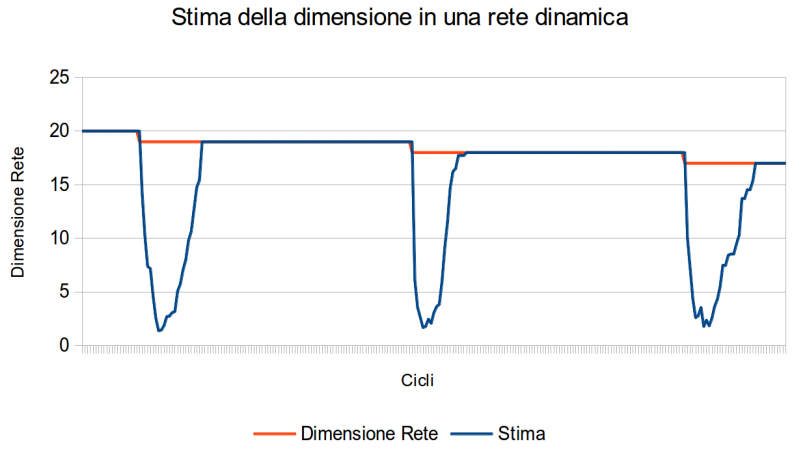
Utilizzo di una funzione logistica Si noti come in entrambi le figure 9a e 9b la stima del calcolo precipita bruscamente dopo ogni disconnessione. Per avere delle stime più precise, invece di considerare la formula 8.1.1 si potrebbe utilizzare la formula seguente:

$$X = \begin{cases} (1 - f(t))X_{old} + f(t)C_s & \text{se } C_s < X_{old} \\ C_s & \text{se } C_s \geq X_{old} \end{cases} \tag{8.1.2}$$

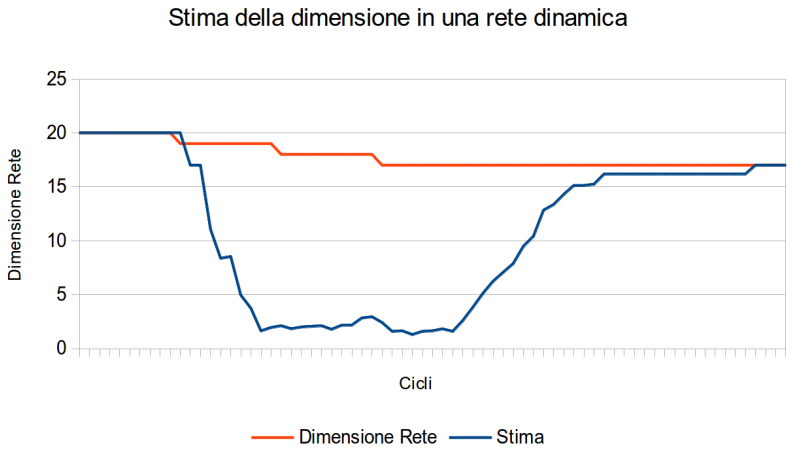
Dove il valore X_{old} rappresenta il vecchio valore conservato dal nodo prima del riconteggio e $f(t)$ risulta essere la funzione logistica seguente:

$$f(t) = \frac{1}{1 + e^{-t+2D+5}} \tag{8.1.3}$$

Dove D rappresenta la distanza verso il beacon, e il valore t rappresenta il numero di cicli in cui un nodo ha generato lo stesso messaggio IS (quindi un numero che cresce al procedere della simulazione), in modo da poter shiftare la stima da X_{old} verso il nuovo C_s.



(a) Disconnessioni distanti



(b) Disconnessioni ravvicinate

Immagine 9: Grafico che rappresenta la stima della dimensione della rete in condizioni di dinamicità

9 User guide

La simulazione è stata realizzata in Java utilizzando il simulatore Peersim ed è stata corredata di un file **ant** (il file `build.xml`) che offre dei target per automatizzare il processo di compilazione e di esecuzione del software.

Per funzionare, la simulazione ha bisogno di un file di configurazione in input da cui poter caricare la configurazione della rete (numero di nodi, numero di link, protocolli da utilizzare, etc...). Alcuni esempi di file di configurazione possono essere trovati all'interno della cartella `example/`.

Per compilare la simulazione è necessario posizionarsi all'interno della directory dove è contenuto il software ed invocare da terminale il comando

```
ant build
```

che provvederà ad invocare il compilatore `javac` per compilare i sorgenti presenti all'interno della cartella `src/`, i file `.class` generati si troveranno all'interno della cartella `bin/`.

Per pulire la cartella `bin/` al fine di avere un ambiente pulito per poter effettuare una nuova compilazione è possibile utilizzare il target

```
ant clean
```

È infine possibile generare un file `jar` contenente tutti i file compilati e tutte le librerie necessarie all'esecuzione. Per farlo è sufficiente invocare il target

```
ant jar
```

Verrà generato un file chiamato `p2p_final.jar` all'interno della cartella principale del software. Per avviare il file `jar` è necessario invocare il comando

```
java -jar p2p_final.jar [file di configurazione]
```

9.1 Documentazione

Al fine di rendere il codice sorgente più comprensibile, il software è stato corredato di documentazione. In particolare tutte le parti del codice sorgente che potrebbero risultare di difficile comprensione sono state commentate. Inoltre ogni funzione e classe del software è stata documentata con il formato javadoc, la documentazione generata può essere visionata all'interno della cartella doc/ e può essere rigenerata utilizzando il comando

```
ant javadoc
```

Per una comprensione organica del software si consiglia la lettura della seguente relazione nella sua interezza. La presente relazione viene rilasciata in Pdf ed in LaTeX e può essere visionata all'interno della cartella `doc/tex/`.

9.2 Avvio della simulazione

Una volta compilato il software è possibile invocarlo tramite il comando

```
ant Execute [-Dfile=nomefile]
```

Con `-Dfile=nomefile` è possibile impostare il file di configurazione da usare. Per effettuare alcune computazioni di esempio è quindi sufficiente impostare `-Dfile=example/<nomefile.conf>`, eseguendo alcuni dei files di esempio già predisposti all'interno della cartella `example`.

I nomi dei files contenuti nella cartella `example` permettono di comprendere facilmente quali sono le configurazioni impostate. Sono stati predisposti calcoli solamente COUNT oppure COUNT-BEACON, con grafi random, small-world e scale-free. Inoltre sono stati predisposti esempi con debugger attivato ed altri esempi in cui si calcolano altre funzioni (massimo, somma, etc...).

Se la struttura dei file di configurazione non fosse chiara è possibile visionare il file example.conf che contiene tutti i commenti necessari a comprendere le impostazioni possibili.

A Codice Sorgente

Di seguito è presente il codice sorgente dell'applicazione diviso per classi. Sono presenti i commenti prima di ogni classe, metodo e campo nello stile Javadoc. Sono inoltre presenti i commenti all'interno dei punti critici del codice per comprendere a meglio i punti più delicati.

A.1 Classi relative a COUNT

A.1.1 Classe **CountModule**

```java
package it.ncorti.p2p;

import peersim.cdsim.CDProtocol;
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.Linkable;
import peersim.core.Node;

/**
 * Classe che rappresenta il modulo che implementa l'algoritmo COUNT,
 * la classe implementa CDProtocol e puo' essere usata all'interno del simulatore
     Peersim.
 *
 * Ad ogni ciclo ogni nodo contattera' in modo casuale un altro nodo e verra'
     effettuato lo scambio
 * delle informazioni fra i due nodi
 *
 * @author Nicola Corti
 */
public class CountModule implements CDProtocol {

    /** Stringa per ottenere il valore iniziale */
    protected static final String param_value = "value";
    /** Stringa per ottenere la funzione di aggregazione da usare */
    protected static final String param_aggreg = "func";

    /** Messaggio in attesa di essere inviato */
    protected Message waiting;
    /** Messaggio appena ricevuto */
    protected Message received;

    /** Variabile di stato: valore attuale */
    protected int state_value;
    /** Variabile di stato: freschezza attuale */
    protected int state_freshness;

    /** ID dell'istanza del protocollo */
    protected int ID = -1;

    /** Valore iniziale del calcolo */
    private int init_value;
    /** Funzione di aggregazione usata */
    private String func;


    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public CountModule(String prefix) {

        init_value = 1;
        state_value = init_value;
        state_freshness = 1;
```

```
        func = Configuration.getString(prefix + "." + param_aggreg);

        if (func.contentEquals("count") || func.contentEquals("sum")){
            waiting = new Message();
            received = new Message();
        } else if (func.contentEquals("min")){
            waiting = new MinMessage(init_value);
            received = new MinMessage(init_value);
        } else if (func.contentEquals("max")){
            waiting = new MaxMessage(init_value);
            received = new MaxMessage(init_value);
        }
    }

    /* (non-Javadoc)
     * @see peersim.cdsim.CDProtocol#nextCycle(peersim.core.Node, int)
     */
    @Override
    public void nextCycle(Node node, int protocolID) {

        /* Funzione invocata ad ogni ciclo della simulazione
         * 1) Contatta un nodo vicino
         * 2) Deposita il proprio messaggio in attesa
         * 3) Invoca l'aggiornamento del messaggio sul vicino
         * 4) Genera un messaggio di IS partendo dallo stato
         */

        CountModule next = this.getNeighboor(node, protocolID);
        if (next == null) return;

        next.received.update(this.waiting);

        if (next.updateMessage(this))
            waiting.generateISMessage(state_value, state_freshness);
    }

    /**
     * Funzione per aggiornare il proprio messaggio in attesa.
     * Si aggiorna il messaggio cercando di mantenere l'informazione contenuta nel
         messaggio IC
     * (scartando quindi eventuali messaggi IS) ed effettuando un merge quando
         entrambi i messaggi
     * sono IC
     *
     * @param sender Il riferimento al mittente che ha depositato il messaggio
         presso il nodo
     * @return true se l'update e' andato a buon fine, false altrimenti.
     */
    protected boolean updateMessage(CountModule sender) {

        if (received.IS() && waiting.IS() && received.isFresherThen(waiting)){
            waiting.update(received);
        } else if (received.IC() && waiting.IS()){
            waiting.update(received);
        } else if (received.IC() && waiting.IC()){
            waiting.merge(received);
        }

        // Aggiorna le variabili di stato
        if (waiting.isFresherThen(state_freshness)){
            state_freshness = waiting.getFreshness();
            state_value = waiting.getValue();
        }
        return true;
    }

    /**
     * Funzione utilizzata dagli inizializzatori per impostare il valore iniziale
     * del messaggio in attesa
```

```
     *
     * @param val Valore iniziale
     */
    protected void initValue(int val){
        this.waiting.value = val;
        this.init_value = val;
    }

    /**
     * Funzione che ritorna un nodo vicino scelto in modo casuale
     *
     * @param node Nodo di cui si cerca un vicino
     * @param protocolID Id del protocollo Linkable che rappresenta la rete
     * @return Un'istanza di CountModule, null se non ci sono vicini.
     */
    protected CountModule getNeighboor(Node node, int protocolID){
        int linkableID = FastConfig.getLinkable(protocolID);
        Linkable link = (Linkable) node.getProtocol(linkableID);

        CountModule next = null;

        int index = (int) (Math.random() * link.degree());
        if (link.degree() == 0)
            return null;

        next = (CountModule) link.getNeighbor(index).getProtocol(protocolID);

        return next;
    }

    /**
     * Funzione che fa ripartire il conteggio di un nodo dal proprio valore
         iniziale
     */
    protected void resetCount(){
        this.state_value = init_value;
        this.state_freshness = 1;
        this.waiting.update(init_value, 1, 1);
    }

    /**
     * Ritorna l'ID dell'istanza protocollo/nodo (NB: e non del nodo), necessario
         per le stampe di debug
     * realizzate dalla classe Debugger
     *
     * @return L'ID del protocollo
     */
    public int getIndex(){
        return this.ID;
    }

    /* (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() { return ("Node " + this.getIndex() + " \t Count: r =
        " + this.received + " w = " + this.waiting + " \t state v = " +
        state_value + " f = " + state_freshness + " \t"); }

    /* (non-Javadoc)
     * @see java.lang.Object#clone()
     *
     * Il metodo e' particolarmente importante in quanto Peersim usa il metodo
         clone per effettuare
     * la generazione di nuove istanze del protocollo.
     */
    @Override
    public Object clone(){
        CountModule node = null;
```

```
        try { node=(CountModule)super.clone();

        if (func.contentEquals("count") || func.contentEquals("sum")){
            node.waiting = new Message();
            node.received = new Message();
        } else if (func.contentEquals("min")){
            node.waiting = new MinMessage(init_value);
            node.received = new MinMessage(init_value);
        } else if (func.contentEquals("max")){
            node.waiting = new MaxMessage(init_value);
            node.received = new MaxMessage(init_value);
        }

        node.received.update(this.received);
        node.waiting.update(this.waiting);
        node.state_value = this.state_value;
        node.state_freshness = this.state_freshness;
        node.init_value = this.init_value;

        } catch (CloneNotSupportedException e) { e.printStackTrace(); }
        return node;
    }
}
```

A.1.2 Classe **Message**

```
package it.ncorti.p2p;

/**
 * @author nicola
 *
 */
/**
 * @author nicola
 *
 */
public class Message {

    protected int value;
    protected int freshness;
    protected int type;

    /**
     * Costruttore utilizzato per generare un nuovo messaggio
     * con un valore di partenza
     * @param val Il valore di partenza del messaggio
     */
    public Message(int val){
        value = val;
        freshness = 1;
        type = 1;
    }
    /**
     * Costruttore utilizzato per generare un nuovo messaggio
     * con un valore di partenza uguale ad 1
     */
    public Message(){
        this(1);
    }
    /**
     * Costruttore utilizzato per generare un nuovo messaggio
     * permettendo di impostare tutti i campi
     * @param val Il valore di partenza del messaggio
     * @param fresh Il valore di freschezza del messaggio
     * @param ty Il tipo del messaggio
     */
    public Message(int val, int fresh, int ty){
```

```
        value = val;
        freshness = fresh;
        type = ty;
    }

    /**Ritorna il valore del messaggio
     * @return Il valore contenuto nel messaggio
     */
    public int getValue() { return value; }

    /**Ritorna il valore di freschezza del messaggio
     * @return Valore dei freschezza del messaggio
     */
    public int getFreshness() { return freshness; }

    /**Metodo per controllare se il messaggio e' di tipo IC
     * @return true se il messaggio e' di tipo IC, false altrimenti
     */
    public boolean IC(){ return (type == 1); }
    /**Metodo per controllare se il messaggio e' di tipo IS
     * @return true se il messaggio e' di tipo IS, false altrimenti
     */
    public boolean IS(){ return (type == 0); }


    /** Permette di confrontare quale fra due messaggi e' piu' recente
     * @param m Secondo messaggio con cui confrontare
     * @return true se il primo messaggio e' piu' recente, false altrimenti
     */
    public boolean isFresherThen(Message m){ return (freshness > m.freshness); }
    /** Permette di confrontare il messaggio con un valore di freschezza
     * @param fr Valore di freschezza con cui confrontare
     * @return true se il messaggio e' piu' recente, false altrimenti
     */
    public boolean isFresherThen(int fr){ return (freshness > fr); }

    /** Metodo per aggiornare contemporaneamente tutti i campi di un messaggio
     * @param val Nuovo valore del messaggio
     * @param fresh Nuovo valore di freschezza
     * @param ty Nuovo tipo del messaggio
     */
    public void update(int val, int fresh, int ty){
        value = val;
        freshness = fresh;
        type = ty;
    }

    /** Metodo per aggiornare i campi di un messaggio partendo da un altro
        messaggio
     * @param m Messaggio da cui copiare i valori
     */
    public void update(Message m){
        this.value = m.value;
        this.freshness = m.freshness;
        this.type = m.type;
    }

    /** Metodo per aggiornare il valore di un messaggio
     * @param val Nuovo valore del messaggio
     */
    public void update(int val){
        this.value = val;
    }

    /** Metodo per fondere due messaggi IC in uno nuovo che contenga
        l'informazione risultante
     * @param m Messaggio da unire
     */
    public void merge(Message m){
        this.value += m.value;
```

```
            this.freshness += m.freshness;
            this.type = 1;
        }

        /** Trasforma il messaggio in un messaggio IS, dati valore e freschezza
         * @param val Nuovo valore del messaggio
         * @param fr Nuovo valore di freschezza
         */
        public void generateISMessage(int val, int fr){
            this.value = val;
            this.freshness = fr;
            this.type = 0;
        }

        /** Reimposta il messaggio partendo da un valore iniziale, impostando la
         * freschezza ad 1 e il tipo ad IC
         * @param init_value Valore iniziale da dare al messaggio
         */
        public void resetMessage(int init_value){
            this.value = init_value;
            this.freshness = 1;
            this.type = 1;
        }

        /* (non-Javadoc)
         * @see java.lang.Object#toString()
         */
        @Override
        public String toString() {
            String message;
            if (type == 0)
                message = "{" + value + "|" + freshness + "|IS}";
            else
                message = "{" + value + "|" + freshness + "|IC}";
            return message;
        }


    }
```

A.2 Classi relative a BEACON

A.2.1 Classe **CountBeaconModule**

```
package it.ncorti.p2p;

import java.util.ArrayList;
import java.util.List;

import peersim.config.FastConfig;
import peersim.core.Linkable;
import peersim.core.Node;

/**
 * Classe che rappresenta il modulo che implementa l'algoritmo COUNT-BEACON,
 * la classe implementa CDProtocol e puo' essere usata all'interno del simulatore
     Peersim.
 *
 * Oltre ad eseguire la computazione di COUNT, con una probabilita' p = 0.50
     verra' effettuata
 * la computazione BEACON fra due nodi casuali
 *
 * @author Nicola Corti
 */
public class CountBeaconModule extends CountModule {

    /** Esercito del nodo */
```

```
    protected Army army;

    /** Lista dei nodi che si sono disconnessi
     *  Utile per non notificare due volte la disconessione
     */
    private List<Node> disconnected;


    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public CountBeaconModule(String prefix) {
        super(prefix);
        army = new Army(this);
        disconnected = new ArrayList<>();
    }


    /* (non-Javadoc)
     * @see it.ncorti.p2p.CountModule#
     *  nextCycle(peersim.core.Node, int)
     */
    @Override
    public void nextCycle(Node node, int protocolID) {
        super.nextCycle(node, protocolID);

        int linkableID = FastConfig.getLinkable(protocolID);
        Linkable link = (Linkable) node.getProtocol(linkableID);


        // Se uno dei vicini e' morto, resuscito l'esercito
        if (!checkNeighboor(link)){
            this.army.revive(this);
            this.resetCount();
        }

        // Con probabilita' p = 0.50 faccio una computazione BEACON fra due nodi
        if (Math.random() > 0.50){

            CountBeaconModule next = null;
            int count = link.degree();

            // Itero fin quando non trovo un nodo up
            while (next == null && count != 0){
                int index = (int) (Math.random()* link.degree());
                Node d = link.getNeighbor(index);
                if (d.isUp()) next = (CountBeaconModule)d.getProtocol(protocolID);
                count--;
            }

            if (count == 0 && next == null) return;

            // Eseguo la schermaglia oppure aggiorno il percorso
            if (next.army.isSameArmy(this.army)){
                Army.updateShortest(this, next);
            } else {
                Army.skirmish(this, next);
            }
        }
    }

    /**
     * Funzione per controllare se tutti i nodi vicini sono sempre up
     *
     * @param link Protocollo che rappresenta la rete
     * @return true se sono sutti up, false altrimenti
     */
    private boolean checkNeighboor(Linkable link) {
```

```
        for (int i = 0; i < link.degree(); i++){
            Node e = link.getNeighbor(i);
            if (!e.isUp() && !disconnected.contains(e)){
                disconnected.add(e);
                return false;
            }
        }
        return true;
    }


    /* (non-Javadoc)
     * @see it.ncorti.p2p.CountModule#
     *  updateMessage(it.ncorti.p2p.CountModule)
     */
    @Override
    protected boolean updateMessage(CountModule sender) {

        // Scarta i messaggi che non sono
        if (sender instanceof CountBeaconModule){
            CountBeaconModule send = (CountBeaconModule) sender;
            if (send.army.isSameArmy(this.army)){
                return super.updateMessage(sender);
            }
        }
        return false;
    }


    /* (non-Javadoc)
     * @see it.ncorti.p2p.CountModule#getNeighboor(peersim.core.Node, int)
     */
    @Override
    protected CountModule getNeighboor(Node node, int protocolID) {
        // Ritorna sempre il vicino piu' prossimo al beacon, tranne se lui stesso
            e' il beacon

        if (waiting.IC() && !this.isBeacon())
            return this.army.nexthop;
        return super.getNeighboor(node, protocolID);
    }


    /**
     * Funzione che ritorna true se il nodo in questione e' il beacon
         dell'esercito.
     *
     * @return true se il nodo e' il beacon, false altrimenti
     */
    private boolean isBeacon() {
        return (army.beacon.equals(this));
    }


    /* (non-Javadoc)
     * @see it.ncorti.p2p.CountModule#clone()
     */
    @Override
    public Object clone(){
        CountBeaconModule node = null;
        node=(CountBeaconModule)super.clone();
        node.army = new Army(node);
        node.disconnected = new ArrayList<>();
        return node;
    }

    /* (non-Javadoc)
     * @see it.ncorti.p2p.CountModule#toString()
     */
    @Override
```

```
    public String toString(){
        return super.toString() + "\t | Beacon : " + army;
    }

}
```

A.2.2 Classe **Army**

```
package it.ncorti.p2p;

import peersim.core.Network;

/**
 * Classe che rappresenta un esercito, utilizzata dal modulo BEACON per il calcolo
     del combattimento fra gli esericiti.
 * Ogni nodo che implementa il protocollo COUNT-BEACON disporra' di un'istanza di
     questa classe.
 *
 * @author Nicola Corti
 */
public class Army {

    /** Valore che rappresenta la forza dell'esercito */
    public int strenght;
    /** Riferimento al nodo che adesso svolge il ruolo di beacon */
    public CountBeaconModule beacon;
    /** Riferimento al prossimo nodo verso il beacon */
    public CountBeaconModule nexthop;
    /** Distanza verso il beacon */
    public int distance;
    /** Immunita' verso un esercito */
    public CountBeaconModule immunity;

    /**
     * Costruttore che crea un nuovo esercito a partire dal nodo che lo deve
         possedere.
     * Lo inizializza con una forza random e imposta il nodo come beacon
     *
     * @param node
     */
    public Army(CountBeaconModule node){

        strenght = (int) (Math.random() * Network.size());
        beacon = node;
        nexthop = node;
        distance = 0;
        immunity = null;

    }

    /**
     * Confronta due eserciti e indica se sono uguali
     *
     * @param army Esercito con cui confrontare
     * @return true se i due eserciti sono lo stesso (stesso beacon) oppure false
     */
    public boolean isSameArmy(Army army) {
        return (this.beacon.equals(army.beacon));
    }

    /**
     * Aggiorna la distanza fra due nodi appartenenti allo stesso beacon fra la
         piu' breve
     * dei due nodi
     *
     * @param i Primo nodo del confronto
     * @param j Secondo nodo del confronto
```

```
     */
    public static void updateShortest(CountBeaconModule i, CountBeaconModule j) {

        if (i.army.distance < j.army.distance + 1){
            j.army.nexthop = i;
            j.army.distance = i.army.distance + 1;
        } else if (j.army.distance < i.army.distance + 1){
            i.army.nexthop = j;
            i.army.distance = j.army.distance + 1;
        }

    }

    /**
     * Svolge una battaglia fra due nodi, andando a controllare se i due nodi
         appartengono allo stesso esercito,
     * se ci sono delle immunita', oppure quale dei due nodi risulta vincitore
     *
     * @param i Primo nodo coinvolto
     * @param j Secondo nodo coinvolto
     */
    public static void skirmish(CountBeaconModule i, CountBeaconModule j) {
        if (i.army.beacon.equals(j.army.immunity)){
            computeWinner(j, i);
            return;
        }
        if (j.army.beacon.equals(i.army.immunity)){
            computeWinner(i, j);
            return;
        }
        if (j.army.beacon.equals(i.army.beacon)){
            updateShortest(i, j);
            return;
        }
        if (i.army.strenght > j.army.strenght){
            computeWinner(i, j);
        } else {
            computeWinner(j, i);
        }
    }

    /**
     * Calcola il vincitore fra due nodi che si sono contesi e ne aggiorna i
         parametri
     *
     * @param i Nodo vincitore
     * @param j Nodo perdente
     */
    private static void computeWinner(CountBeaconModule i, CountBeaconModule j){

        j.army.beacon = i.army.beacon;
        j.army.strenght = i.army.strenght;
        j.army.immunity = i.army.immunity;
        j.army.distance = i.army.distance + 1;
        j.army.nexthop = i;
        j.resetCount();
    }

    /**
     * Resuscita un esercito, utile quando un nodo si rende conto che uno dei suoi
         vicini si e' disconnesso.
     *
     * @param i Nodo verso cui impostare l'immunita'
     */
    public void revive(CountBeaconModule i){

        immunity = beacon;
        strenght = (int) (Math.random() * Network.size());
        beacon = i;
        nexthop = i;
```

```
            distance = 0;
        }

        /* (non-Javadoc)
         * @see java.lang.Object#toString()
         */
        @Override
        public String toString(){
            if (immunity != null)
                return ("{ B: " + beacon.getIndex() + " S:" + strenght + " I: " +
                    immunity.getIndex() + " D: " + distance + " Next: " +
                    nexthop.getIndex() + " }");
            else
                return ("{ B: " + beacon.getIndex() + " S:" + strenght + " I: null D:
                    " + distance + " Next: " + nexthop.getIndex() + " }");
        }

    }
```

A.3 Classi di Inizializzatori

A.3.1 Classe **CountBeaconInitializer**

```
package it.ncorti.p2p;

import java.util.ArrayList;
import java.util.List;

import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;


/**
 * Classe che rappresenta un inizializzatore per il protocollo COUNT-BEACON.
 * Imposta un ID a tutti i nodi in modo da poterlo utilizzare per le stampe di
     debug
 * e comprendere come si evolve la simulazione
 *
 * Questa classe inoltre inizializza il valore di forza di ogni esercito in modo
     che sia random perfetto
 *
 * @author Nicola Corti
 */
public class CountBeaconInitializer implements Control {


    /** Stringa per recuperare il nome del protocollo dal file di conf */
    private static final String PAR_PROT = "protocol";
    /** Pid del protocollo CountBeacon */
    private final int pid;

    /** Lista di interi distinti per assegnare i valori di forza */
    private List<Integer> ints;


    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public CountBeaconInitializer(String prefix) {
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
        ints = new ArrayList<>();
        for (int j = 0; j < Network.size(); j++){
            ints.add(j);
        }
```

```
    }

    /* (non-Javadoc)
     * @see peersim.core.Control#execute()
     */
    @Override
    public boolean execute() {

        for (int i = 0; i < Network.size(); i++) {
            CountModule prot = (CountModule) Network.get(i).getProtocol(pid);
            prot.ID = i;
            if (prot instanceof CountBeaconModule){
                CountBeaconModule cbm = (CountBeaconModule) prot;
                int j = (int) (Math.random() * ints.size());
                if (j >= 0 && j < ints.size()){
                    cbm.army.strenght = ints.get(j);
                    ints.remove(j);
                } else {
                    cbm.army.strenght = ints.get(0);
                    ints.remove(0);
                }
            }
        }

        return false;
    }
}
```

A.3.2 Classe **RandomInitializer**

```
package it.ncorti.p2p;

import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;

/**
 * Inizializzatore per il modulo COUNT che assegna il valore a tutti i nodi
 * in modo casuale, potendo indicare il valore massimo e minimo in cui generare i
     numeri casuali
 *
 * @author Nicola Corti
 */
public class RandomInitializer implements Control {

    /** Stringa per recuperare il nome del protocollo dal file di conf */
    private static final String PAR_PROT = "protocol";
    /** Stringa per recuperare il valore massimo dal file di conf */
    private static final String PAR_MAX = "max";
    /** Stringa per recuperare il valore minimo dal file di conf */
    private static final String PAR_MIN = "min";

    /** Pid del protocollo */
    private final int pid;
    /** Valore massimo del random */
    private final int max;
    /** Valore minimo del random */
    private final int min;

    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public RandomInitializer(String prefix) {
        max = Configuration.getInt(prefix + "." + PAR_MAX);
        min = Configuration.getInt(prefix + "." + PAR_MIN);
```

```
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
    }

    /* (non-Javadoc)
     * @see peersim.core.Control#execute()
     */
    @Override
    public boolean execute() {
        for (int i = 0; i < Network.size(); i++) {
            CountModule prot = (CountModule) Network.get(i).getProtocol(pid);
            int val = (int) (Math.random() * (max - min));
            val += min;
            prot.initValue(val);
        }
        return false;
    }
}
```

A.3.3 Classe **PeakInitializer**

```
package it.ncorti.p2p;

import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;


/**
 * Inizializzatore per il modulo COUNT che assegna il valore 0 a tutti i nodi
 * e assegna un valore di picco (parametro 'val' nel file di conf) al primo nodo
     della
 * rete
 *
 * @author Nicola Corti
 */
public class PeakInitializer implements Control {

    /** Stringa per recuperare il nome del protocollo dal file di conf */
    private static final String PAR_PROT = "protocol";
    /** Pid del protocollo */
    private final int pid;

    /** Stringa per recuperare il valore del picco dal file di conf */
    private static final String PAR_VALUE = "value";
    /** Valore del picco */
    private final int value;

    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public PeakInitializer(String prefix) {
        value = Configuration.getInt(prefix + "." + PAR_VALUE);
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
    }

    /* (non-Javadoc)
     * @see peersim.core.Control#execute()
     */
    @Override
    public boolean execute() {
        for (int i = 0; i < Network.size(); i++) {
            CountModule prot = (CountModule) Network.get(i).getProtocol(pid);
            prot.initValue(0);
        }
        CountModule prot = (CountModule) Network.get(0).getProtocol(pid);
```

```
            prot.initValue(value);
            return false;
        }
    }
```

A.3.4 Classe **LinearInitializer**

```
package it.ncorti.p2p;

import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;


/**
 * Inizializzatore per il modulo COUNT che assegna un valore in modo lineare,
 * partendo da 1 fino alla dimensione della rete.
 *
 * @author Nicola Corti
 */
public class LinearInitializer implements Control {

    /** Stringa per recuperare il nome del protocollo dal file di conf */
    private static final String PAR_PROT = "protocol";
    /** Pid del protocollo */
    private final int pid;


    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public LinearInitializer(String prefix) {
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
    }

    /* (non-Javadoc)
     * @see peersim.core.Control#execute()
     */
    @Override
    public boolean execute() {
        for (int i = 0; i < Network.size(); i++) {
            CountModule prot = (CountModule) Network.get(i).getProtocol(pid);
            prot.initValue(i);
        }
        return false;
    }
}
```

A.4 Sottoclassi di Message

A.4.1 Classe MaxMessage

```java
package it.ncorti.p2p;

/**
 * Classe che rappresenta un messaggio di COUNT ed utilizza la funzione di
     aggregazione
 * MAX in modo da trovare il massimo dei valori
 *
 * @author nicola
 */
public class MaxMessage extends Message {

    /**
     * Costruttore per creare un nuovo messaggio di massimo, partendo da un valore
     *
     * @param val Valore iniziale
     */
    public MaxMessage(int val){
        super(val);
    }

    @Override
    public void merge(Message m) {
        this.freshness += m.freshness;
        this.type = 1;
        this.value = Math.max(this.value, m.value);
    }

}
```

A.4.2 Classe MinMessage

```java
package it.ncorti.p2p;

/**
 * Classe che rappresenta un messaggio di COUNT ed utilizza la funzione di
     aggregazione
 * MIN in modo da trovare il minimo dei valori
 *
 * @author nicola
 */
public class MinMessage extends Message {

    /**
     * Costruttore per creare un nuovo messaggio di minimo, partendo da un valore
     *
     * @param val Valore iniziale
     */
    public MinMessage(int val){
        super(val);
    }

    /* (non-Javadoc)
     * @see it.ncorti.p2p.Message#merge(it.ncorti.p2p.Message)
     */
    @Override
    public void merge(Message m) {
        this.freshness += m.freshness;
        this.type = 1;
        this.value = Math.min(this.value, m.value);
    }

}
```

A.5 Classi di Osservatori

A.5.1 Classe **Debugger**

```java
package it.ncorti.p2p;

import peersim.config.Configuration;
import peersim.core.Control;
import peersim.core.Network;

/**
 * Classe di Debug che stampa su standard output tutte le informazioni di tutti i
     nodi della rete
 * ad ogni ciclo di esecuzione
 *
 * @author Nicola Corti
 */
public class Debugger implements Control {

    /** Stringa per recuperare il nome del protocollo dal file di conf */
    private static final String PAR_PROT = "protocol";
    /** Pid del protocollo CountBeacon */
    private final int pid;


    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public Debugger(String prefix){
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
    }


    /* (non-Javadoc)
     * @see peersim.core.Control#execute()
     */
    @Override
    public boolean execute() {
        for (int i = 0; i < Network.size(); ++i){
            CountModule node = ((CountModule) Network.get(i).getProtocol(pid));
            if (node != null) System.out.println(node);
        }
        System.out.println("-------------");
        return false;
    }

}
```

A.5.2 Classe **Statistics**

```java
package it.ncorti.p2p;

import peersim.config.Configuration;
import peersim.core.CommonState;
import peersim.core.Control;
import peersim.core.Network;

/**
 * Classe per la raccolta delle statistiche dell'esecuzione.
 * Vengono raccolte informazioni sul numero di messaggi IC e IS presenti ad ogni
     ciclo.
 * Viene inoltre visualizzato quando vengono completate le seguenti fasi (nel caso
     di COUNT-BEACON)
 *
 * 1) Raggiunto un singolo BEACON
```

```
 * 2) Tutti i messaggi IC sono stati raccolti in un singolo nodo
 * 3) L'informazione e' stata condivisa a tutti i nodi presenti nella rete
 *
 * Impostato la configurazione su ('silent true') verranno solo visualizzate le 3
     fasi, senza indicare
 * il conteggio dei messaggi ad ogni ciclo
 *
 * @author Nicola Corti
 */
public class Statistics implements Control {

    /** Stringa per recuperare il nome del protocollo dal file di conf */
    private static final String PAR_PROT = "protocol";
    /** Pid del protocollo */
    private final int pid;

    /** Stringa per recuperare l'impostazione del silent dal file di conf */
    private static final String PAR_SILENT = "silent";
    /** Valore del picco */
    private boolean silent = true;

    /** Variabile di stato per indicare la fase in cui ci si trova */
    private int phase = 1;


    /**
     * Costruttore Base invocato dal simulatore
     * @param prefix prefisso indicato nel file di configurazione
     */
    public Statistics(String prefix){
        this.pid = Configuration.getPid(prefix + "." + PAR_PROT);
        this.silent = Configuration.getBoolean(prefix + "." + PAR_SILENT, false);
    }



    /* (non-Javadoc)
     * @see peersim.core.Control#execute()
     */
    @Override
    public boolean execute() {

        // Conteggio messaggi IS
        int IScount = 0;
        // Conteggio messaggi IC
        int ICcount = 0;
        // Conteggio beacon
        int BeaconCount = 0;
        // Valore atteso dalla rete
        int ExpectedVal = -1;
        // Flag che indica se tutti i nodi hanno lo stesso valore
        boolean SameVal = true;

        for (int i = 0; i < Network.size(); ++i){

            CountModule node = ((CountModule)
                Network.get(i).getProtocol(this.pid));

            if (node instanceof CountBeaconModule){
                CountBeaconModule cnode = (CountBeaconModule) node;
                if (cnode.army.distance == 0) BeaconCount++;
            }

            if (node.waiting.IC()) ICcount++;
            if (node.waiting.IS()) {
                IScount++;
                if (ExpectedVal == -1) ExpectedVal = node.waiting.value;
            }

            SameVal &= (node.waiting.value == ExpectedVal);
```

```
        }

        if (Network.get(0).getProtocol(this.pid) instanceof CountBeaconModule){

            // Aggiorna fase nel caso COUNT-BEACON
            if (BeaconCount == 1 && ICcount != 1 && !SameVal && phase == 1){
                System.out.println(" ---------------- PHASE 1 COMPLETE @" +
                    CommonState.getIntTime() + " ------------- ");
                phase++;
            }
            if (BeaconCount == 1 && ICcount == 1 && !SameVal && phase == 2){
                System.out.println(" ---------------- PHASE 2 COMPLETE @" +
                    CommonState.getIntTime() + " ------------- ");
                phase++;
            }
            if (BeaconCount == 1 && ICcount == 1 && SameVal && phase == 3){
                System.out.println(" ---------------- PHASE 3 COMPLETE @" +
                    CommonState.getIntTime() + " ------------- ");
                System.out.println(" ################     CONVERGENCE  @" +
                    CommonState.getIntTime() + "  ############## ");
                System.out.println(" ################     IC  @" + ExpectedVal + "
                    ############## ");

                phase++;
                System.exit(0);
            }

            if (!silent) System.out.println("CYCLE: " + CommonState.getIntTime() +
                " - IC: " + ICcount + " - IS: " + IScount + " - Beacon: " +
                BeaconCount);
        } else {

            // Aggiorna fase nel caso COUNT
            if (ICcount == 1 && !SameVal && phase == 1){
                System.out.println(" ---------------- PHASE 1 COMPLETE @" +
                    CommonState.getIntTime() + " ------------- ");
                phase++;
            }
            if (ICcount == 1 && SameVal && phase == 2){
                System.out.println(" ---------------- PHASE 2 COMPLETE @" +
                    CommonState.getIntTime() + " ------------- ");
                System.out.println(" ################     CONVERGENCE  @" +
                    CommonState.getIntTime() + "   ############## ");
                System.out.println(" ################     IC  @" + ExpectedVal + "
                    ############## ");

                phase++;
                System.exit(0);
            }

            if (!silent) System.out.println("CYCLE: " + CommonState.getIntTime() +
                " - IC: " + ICcount + " - IS: " + IScount);
        }

        SameVal = false;
        return false;
    }
}
```

Riferimenti bibliografici

[1] van de Bovenkamp, Ruud, Fernando Kuipers, and Piet Van Mieghem. Gossip-based counting in dynamic networks. NETWORKING 2012. Springer Berlin Heidelberg, 2012. 404-417.

[2] Giakkoupis, George. Tight bounds for rumor spreading in graphs of a given conductance. Leibniz International Proceedings in Informatics (LIPIcs) series 9 (2011): 57-68.

[3] Peersim Simulator, `http://peersim.sourceforge.net/`