ALMA MATER STUDIORUM -
UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN
SCIENZE E TECNOLOGIE
INFORMATICHE

IMPLEMENTAZIONE DI ALCUNI MIGLIORAMENTI AL SIMULATORE LUNES

Relazione finale in
Reti di Calcolatori

Relatore:
Gabriele D'Angelo

Presentata da:
Gabriele Sani

III Sessione
Anno Accademico 2016/2017

Introduzione

Ogni giorno comunichiamo e viviamo all'interno di reti interconnesse tra loro, utilizzando device che trasmettono tra loro messaggi in modo costante. Diventa di grande importanza quindi capire quali siano i metodi migliori per rendere questa comunicazione veloce e non ridondante.

Vengono quindi costantemente sviluppati nuovi algoritmi per raggiungere questi obiettivi. Uno dei problemi principali che si incontra una volta scritti è la fase di testing, in quanto disporre di un numero di dispositivi abbastanza grande da poter ritenere una simulazione accurata è raramente possibile.

Entra quindi in gioco il campo delle simulazioni di reti, che sopperisce a questo limite. LUNES (Large Unstructured NEtwork Simulator) [1] si occupa della creazione di un modello sul quale poter eseguire i nostri algoritmi, di simulare interazioni tra dispositivi all'interno di esso e di fare analisi statistiche alla fine di queste.

Un'applicazione di questo tipo ha nell'affinare le performance il suo più evidente margine di miglioramento. Lo scopo di questa tesi è infatti quello di mostrare alcune modifiche possibili che possano diminuire i tempi di esecuzione di LUNES, lasciandone inalterati i risultati.

Nel primo capitolo verrà fatta un'analisi del software, in modo da mostrare il funzionamento del simulatore ed elencare le parti in cui viene suddivisa una simulazione, oltre a mostrare alcun degli esempi presenti per dare un'idea più chiara di come funziona.

Nel secondo vengono analizzate le problematiche riscontrate e vengono proposte alcune modifiche che possono migliorare le performance del simulatore, oltre a spiegare le motivazioni alla base di tali scelte.

Nel terzo vengono elencati gli algoritmi di disseminazione sui quali si andrà ad eseguire i test, oltre a rappresentare con alcuni grafici il loro funzionamento all'interno di una rete simulata.

Nel quarto vengono mostrati i risultati ottenuti in seguito all'applicazione di queste modifiche al software, evidenziando in quali parti

della simulazione risulta più evidente la riduzione del tempo d'esecuzione.

Infine, nel quinto capitolo, vengono proposti alcuni spunti per ulteriori miglioramenti che si potrebbero apportare a LUNES.

Indice

Elenco delle figure

Elenco delle tabelle

1 Analisi del Software

LUNES è un simulatore di protocolli di disseminazione complessi su reti di grandi dimensioni e non strutturate; simula quindi reti peer-to-peer. Per alcune sue funzionalità si appoggia ad altri due software.

1.1 ARTÌS

ARTÌS (Advanced RTI System) è un middle middleware per PADS (Parallel And Distributed Simulation) che supporta modelli molto popolati. Un PADS è composto da molti LP (Logical Processes) che comunicano tra loro, quindi la velocità di esecuzione è fortemente affetta dalla performance di comunicazione tra i vari LP. ARTÌS utilizza un approccio adattivo e sfrutta la posizione fisica degli LP; supporta inoltre il multi-threading [3].

È composto da un insieme di moduli logici organizzati in una pila. Il livello di comunicazione sta alla base di essa, ed è composto da un insieme di diversi moduli di comunicazione. ARTÌS è in grado di selezionare adattivamente il miglior modulo di interazione rispetto all'allocazione dinamica degli LP nell'ambiente di esecuzione. Al momento dato un insieme di LP sullo stesso host, tali processi comunicano e si sincronizzano tramite memoria condivisa [4].

1.2 GAIA

GAIA (Generic Adaptive Interation Architecture) è un framework di migrazione costruito su ARTÌS. Fornisce al modello simulato i servizi di comunicazione e il supporto per la migrazione delle entità. In questo modo il livello di astrazione fornito allo sviluppatore è relativamente alto [5].

Il suo compito principale è quello quindi di controllare i pattern di comunicazione di ogni entità simulata per tutta la durata della simulazione stessa.

Questo framework è stato utilizzato anche per creare alcuni esempi per introdurre alle simulazioni di questo genere, che andiamo brevemente ad analizzare.

1.2.1 WIRELESS

Questo esempio simula una rete wireless, utilizzando un numero di LP e nodi da noi stabilito. Possono inoltre essere impostate le coordinate X e Y del toroide all'interno del quale avviene la simulazione.

Lanciando la simulazione dopo aver opportunamente modificato i parametri, otteniamo un output (un file per ogni LP) di questo tipo:

```
#LP [0] HOSTNAME [hostname]
<PARAMETERS: TRANSMISSION RANGE=250 AREA_X=100 AREA_Y=100>
#[        0.00]
#[        1.00]
#[        2.00]
[...]
#[       98.00]
#[       99.00]

-- Termination condition reached
-- Clock: 100.00
-- Elapsed Time: 0.02s
-- Processed events: 6933
++ Events per second: 339038.59
```

Figura 1: Esempio di output del modello WIRELESS

Nella prima riga abbiamo l'identificativo del processore logico (#LP [0]) e l'hostname della nostra macchina. Nella seconda riga ci sono le impostazioni lette da uno dei file di configurazione da noi modificabili, mentre di seguito abbiamo una riga per ogni ciclo della simulazione: ne vengono eseguiti 100 di default. Infine abbiamo le statistiche temporali

dell'esecuzione. Per avere un output più consistente dobbiamo attivare la modalità di debug.

A questo punto un esempio di output si mostra invece nel seguente formato:

```
#LP [0] HOSTNAME [hostname]
<PARAMETERS: TRANSMISSION RANGE=250 AREA_X=100 AREA_Y=100>
<BEGIN 0.0000>
<END>
#[        0.00]
<BEGIN 1.0000>
0 0       9.29 99.68
1 0      18.64 95.87
2 0      85.81 96.56
3 0      86.53 19.71
4 0      74.99 77.55
5 1      59.52 36.67
6 1      84.50 21.56
7 1      56.17 92.16
8 1      83.75 22.42
9 1      49.51 54.77
<END>
#[        1.00]
<BEGIN 2.0000>
1 0      22.38       5.15
2 0      77.06       1.41
6 1      75.30      17.63
7 1      55.03       2.09
8 1      92.24      17.14
9 1      58.77      51.00
<END>
```

Figura 2: Esempio di output del modello WIRELESS con debug attivo

e così via per ogni ciclo di simulazione. Le 2 nuove colonne indicano le coordinate nello spazio bidimensionale del nodo preso in esame.

1.2.2 AIRPORT

Questo esempio invece simula 3 aeroporti e gli aerei in volo tra di loro. Sono implementati diversi algoritmi di sincronizzazione per evitare possibili "scontri" tra di essi, alcuni dei quali sono:

- **cmb** (Chandy-Misra-Bryant): concetto alla base del protocollo è quello di messaggio safe, un messaggio si può definire safe solamente quando la sua elaborazione sicuramente non provocherà violazioni causali (es. ricezione da parte dell'LP di messaggi nel passato). Per verificare se un messaggio è safe o meno ogni LP deve essere informato sullo stato di avanzamento degli altri LP connessi;

- **ts** (TimeStepped): In questo caso il tempo viene suddiviso in step sequenziali di una certa durata, gli eventi si riferiscono agli step. Anche questa gestione del tempo è sostanzialmente pessimistica: l'avanzare degli step di simulazione non permette che il vincolo di causalità sia mai violato. In particolare, l'algoritmo TS può essere considerato come una variante del precedente, dove il lookahead viene staticamente determinato in fase di inizializzazione, ed è identico per ogni canale di comunicazione esistente tra i diversi LP;

In questo caso l'output si presenta in questo modo:

```
BOLOGNA <- [MILAN] -> ROME
[MILAN]: 1.0 Departed flight MILAN-0 with destination BOLOGNA
[MILAN]: 2.0 Departed flight MILAN-1 with destination BOLOGNA

[...]

[MILAN]:     1999.0  Landed flight ROME-0 from ROME
[MILAN]: -- ending condition satisfied, clock=2000.00
Total flights departed from MILAN: 553


ROME <- [BOLOGNA] -> MILAN
[BOLOGNA]: 1.0 Departed flight BOLOGNA-0 with destination ROME
[BOLOGNA]: 2.0 Departed flight BOLOGNA-1 with destination MILAN

[...]

[BOLOGNA]:     2000.0  Landed flight MILAN-0 from MILAN
[BOLOGNA]: -- ending condition satisfied, clock=2000.00
Total flights departed from BOLOGNA: 551


MILAN <- [ROME] -> BOLOGNA
[ROME]: 1.0 Departed flight ROME-0 with destination MILAN
[ROME]: 2.0 Departed flight ROME-1 with destination BOLOGNA

[...]

[ROME]: 2000.0 Departed flight BOLOGNA-1 with destination MILAN
[ROME]: -- ending condition satisfied, clock=2000.00
Total flights departed from ROME: 567
```

Figura 3: Esempio di output del modello AIRPORT

Abbiamo un file di output per ogni aeroporto (LP), il timestep in cui è stata effettuata l'operazione e l'operazione (landing, waiting o departure), oltre al numero di voli partiti dall'aeroporto in questione.

Come possiamo vedere da questi semplici esempi, il framework è utilizzabile sia per modellazioni teoriche che per esemplificazioni pratiche (semafori, stazioni ecc.).

1.2.3 Modelli implementati con GAIA

Oltre a questi esempi troviamo anche modelli implementati tramite le Application Program Interface (API) di GAIA [6]. Analizziamo di seguito il modello **MIGRATION-AGENTS**.

Anche in questo caso possiamo modificare alcuni parametri, come il numero di LP, il numero di essi da eseguire sull'host corrente e il numero di entità da simulare. Dopo aver eseguito la simulazione otteniamo un output di questo tipo:

```
#LP [0] HOSTNAME [hostname]
 #LP[0] STARTED
 #
 #Generating Simulated Entities from 0 to 49 ...  OK
 #
 # Data format:
 # column 1: elapsed time (seconds)
 # column 2: timestep
 # column 3: number of entities in this LP
 # column 4: number of migrating entities (from this LP)
 # column 5: local communication ratio (percentage)
 # column 6: remote communication ratio (percentage)
 # column 7: total number of migrations in this timestep
 #
 # [       0.00] [0.00000] 0 0  -nan   -nan  0
 # [       0.00] [1.00000] 50 0  -nan   -nan  0

 [...]

 # [       0.15] [997.00000] 49 1  55.00  45.00  8
 # [       0.15] [998.00000] 46 4  56.00  44.00  14
 # [       0.15] [999.00000] 45 5  54.08  45.92  2


 #-- Termination condition reached (7810)
 #-- Clock                1000.00
 #-- Elapsed Time           0.15s
 #-- Total sent pings:      49900; Total received pings:      49934
```

Figura 4: Output migration agents

Come si può vedere questo modello gestisce il passaggio di host wireless in movimento dal controllo di un LP a un altro, contenendo anche statistiche sulla percentuale di comunicazione locale e remota.

Inizialmente viene istanziata una hash table; ad ogni ciclo della simulazione, dopo aver stabilito la dimensione massima di ogni messaggio, ogni nodo controlla se ne sono in arrivo di nuovi: in caso affermativo in base al tipo di esso vengono eseguite diverse azioni:

- **notify_migration_event_handler**: sposta un'entità simulata (SE) da un LP a un altro; ciò in realtà non può essere fatto immediatamente, poichè per evitare problemi di sincronizzazione dobbiamo attendere la fine del ciclo attuale. Nel frattempo, tramite una sorta di meccanismo di caching, viene creata una lista degli SE che devono essere spostati e alla fine del timestamp corrente essa viene svuotata (viene effettuata la migrazione effettiva).

- **notify_ext_migration_event_handler**: significa che la migrazione non coinvolge direttamente l'LP locale, ma vengono comunque salvate le informazioni relative alla migrazione, poichè sono necessarie in alcuni casi particolari;

- **register_event_handler**: viene creato un nuovo SE (viene registrato nella hash table con un nuovo identificativo);

- **migration_event_handler**: la migrazione vera e propria viene eseguita; viene quindi spostato il nodo all'interno della hash table.

Un altro dei modelli che troviamo implementati è proprio LUNES.

1.3 LUNES

In LUNES una simulazione può essere genericamente suddivisa in tre parti.

1.3.1 Generazione del corpus

Per la simulazione è richiesto un *corpus*, ovvero un insieme di grafi che simulino i collegamenti di un network: tramite il tool igraph [7] lo si può generare. Questo passaggio è opzionale, in quanto si può utilizzare

un grafo creato in precedenza alla simulazione: ciò è particolarmente utile se si vogliono testare le performance dopo eventuali modifiche al codice, in quanto usare lo stesso grafo permette di poter confrontare i tempi d'esecuzione della simulazione. Alcuni grafi sono già disponibili in una directory all'interno del software.

1.3.2 Esecuzione della simulazione

Una volta stabilita la rete sulla quale eseguire i propri test, la simulazione viene avviata, permettendoci di testare l'algoritmo di disseminazione scelto. Durante questo passaggio viene generato un enorme quantitativo di dati raccolti in file (trace file) all'interno dei quali sono riportate, una per riga, tutte le operazioni riguardanti ogni messaggio che viene scambiato o generato durante la simulazione. La struttura di ogni riga del trace file è molto semplice:

- in caso di generazione di un messaggio avremo stampato STAT GEN seguito da un numero intero che lo identifica;
- in caso di ricezione del messaggio avremo stampato STAT RCV seguito dall'identificativo del ricevente, l'identificativo del messaggio ricevuto, il numero di salti che sono stati necessari al messaggio per giungere a destinazione e, se il messaggio non è stato generato localmente, il Time To Leave (TTL) residuo del messaggio;
- infine, preceduto da STAT MSG, alla fine di ogni ciclo della simulazione troveremo il numero totale di messaggi spediti durante il ciclo stesso.

1.3.3 Generazione delle statistiche

Terminata la simulazione vengono computate alcune statistiche che vengono riportate in un singolo file con il seguente contenuto su un'unica riga:

- il numero di nodi del corpus utilizzato per la simulazione;

- la copertura media del grafo, ossia la percentuale media calcolata su ogni messaggio dei nodi del corpus raggiunti durante la simulazione;
- il ritardo medio (delay), ossia la media del numero di salti fatti da ogni messaggio durante la simulazione;
- il numero medio di messaggi spediti durante ogni disseminazione;
- il rapporto di overhead, ovvero il rapporto tra il numero totale di messaggi inviati durante la simulazione e il numero minimo di messaggi necessari per ottenere una copertura completa del grafo [8].

2 Miglioramenti Apportati

Analizzando il software sono stati individuati due punti in cui si poteva tentare di apportare modifiche che incrementassero le performance, entrambi nella parte di generazione delle statistiche.

2.1 Scelta della struttura dati

Dopo che la simulazione è stata effettuata, il software come abbiamo detto calcola alcune statistiche, come la copertura media del grafo, il ritardo medio ecc.. Una parte di queste analisi viene eseguita usando come struttura dati un array di caratteri, che permette un semplice accesso ai dati ricercati. Non essendo però una struttura performante, ho deciso di sostituirla con un'altra, una hash table.

Una hash table è una struttura dati per l'implementazione di dizionari (coppie chiave/valore) che generalizza la nozione più semplice di array ordinato. Teoricamente, la ricerca di un valore al suo interno può essere dispendiosa in termini di tempo come la ricerca in una lista doppiamente linkata (Θ(n) nel caso peggiore), ma nella pratica ha delle ottime performance, e fatte le ragionevoli premesse, il tempo di ricerca medio di un elemento si attesta a $\mathcal{O}(1)$. Nel caso infatti in cui l'insieme delle possibili chiavi sia relativamente limitato, U = 0,1,...,m-1 con m non molto grande, possiamo assumere che nessuna coppia di elementi abbia la stessa chiave abbinata. A questo punto possiamo rappresentare l'insieme di valori come un array T[0,...,m-1] in cui ogni posizione, o slot, corrisponde ad una chiave dell'insieme U [2].

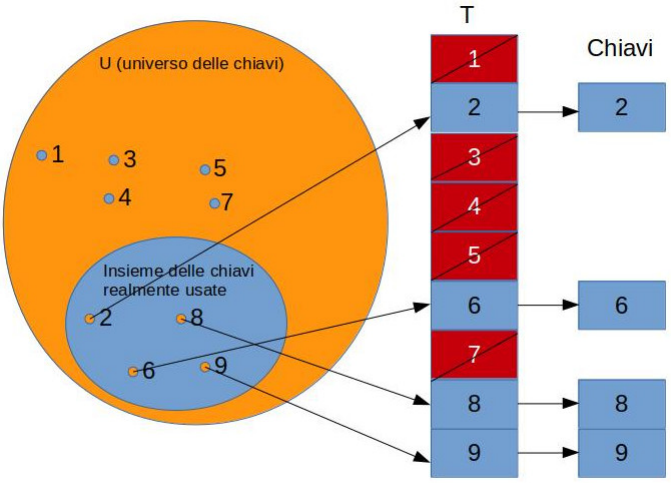

Figura 5: Hash table ad accesso diretto [2]

Questa modalità di implementazione, chiamata tabella ad accesso diretto, ha svariati vantaggi, in quanto le seguenti operazioni hanno tutte tempo di esecuzione pari a $\mathcal{O}(1)$:

- ricerca: data in input una chiave, ritornare il valore ad essa associata;
- inserimento: associare alla chiave k il valore x;
- rimozione: associare alla chiave k un valore nullo.

Per l'implementazione in LUNES sono state utilizzate le Gnome Library (glib) [9], che mettono a disposizione la struttura dati e tutti i metodi ad essa relativi: creazione di hash table, inserimento, ricerca ed eliminazione di valori all'interno di essa.

2.2 Generazione Statistiche

L'analisi dei file di trace è probabilmente l'operazione più onerosa in termini di tempo da parte di LUNES, in quanto tendenzialmente vengono generati diversi gigabyte di dati, che vengono poi analizzati riga

per riga. La lettura dei file trace e il calcolo delle medie che vengono poi scritte nei file di output vengono fatte tramite un breve script Awk.

Awk (dal cognome dei suoi creatori: Alfred Aho, Peter Weinberger e Brian Kernighan) è un linguaggio di programmazione interpretato (non compilato) il cui utilizzo è perlopiù orientato alla manipolazione di testo e stringhe in generale. I suoi vantaggi principali, che condivide con gli altri linguaggi interpretati, sono la semplicità di scrittura, quella di mantenimento del codice e la velocità con cui si possono testare le modifiche del codice, dato che non deve essere ricompilato ogni volta

D'altro canto un linguaggio compilato, come il C, ha in media performance migliori, anche se le operazioni di manipolazione di stringhe sono più complesse da implementare

Il codice sorgente dei linguaggi compilati, infatti, viene trasformato dal compilatore in codice macchina di basso livello, facilmente eseguibile dal computer, e quindi più veloce. Un linguaggio interpretato, invece, viene letto da un interprete che ne esegue i comandi, in modo più lento di quanto non succeda per un linguaggio compilato.

2.2.1 Conversione di Codice AWK in C

LUNES esegue le operazioni di calcolo della media tramite chiamate al file *mean.awk*, al quale vengono passati in input dei file di log:

```
{
   if (NR == 1) {
       sum=min=max=$1;
   } else {
       sum += $1;
   }
}
END {
   printf "%f\n", sum/NR;
}
```

Figura 6: File mean.awk

La prima parentesi graffa indica l'inizio dell'iterazione del file, riga per riga; se il file ha una riga sola (NR è una variabile built-in del linguaggio che indica il numero di righe del file), allora nella variabile **sum** viene salvata la riga stessa, altrimenti ad ogni riga iterata si incrementa il valore della variabile: infine si stampa a schermo la somma divisa per il numero di righe, cioè la media del valore.

Il cambiamento è semplicemente consistito nel sostituire il file **mean.awk** con un sorgente C, che eseguisse le stesse operazioni e mantenesse inalterato il log su console.

3 Algoritmi utilizzati

Per testare le performance in seguito alle modifiche apportate, sono stati utilizzati alcuni degli algoritmi già presenti all'interno del software.

3.1 Trasmissione Probabilistica (Probabilistic Broadcast)

Utilizzando questo protocollo di disseminazione ad ogni chiamata della procedura di gossip, se il nodo in questione ha generato il messaggio ed esso non è ancora stato propagato, quest'ultimo viene mandato a tutti i nodi vicini. Se invece il messaggio è stato ricevuto, allora viene deciso con una certa probabilità se propagarlo o meno. Prima di iniziare la simulazione si può modificare il parametro DISPROB assegnandoli un array di valori interi: per ognuno di essi verrà eseguita una simulazione considerando come probabilità di propagazione ogni elemento.

Algoritmo 1 Trasmissione Probabilistica [3]

```
1: function GOSSIP(msg)
2: if randr ≤ DISPROB||notDisseminated() then
3:     for all neighbours do
4:         send(message, neighbour)
```

Figura 7: Algoritmo di Trasmissione Probabilistica

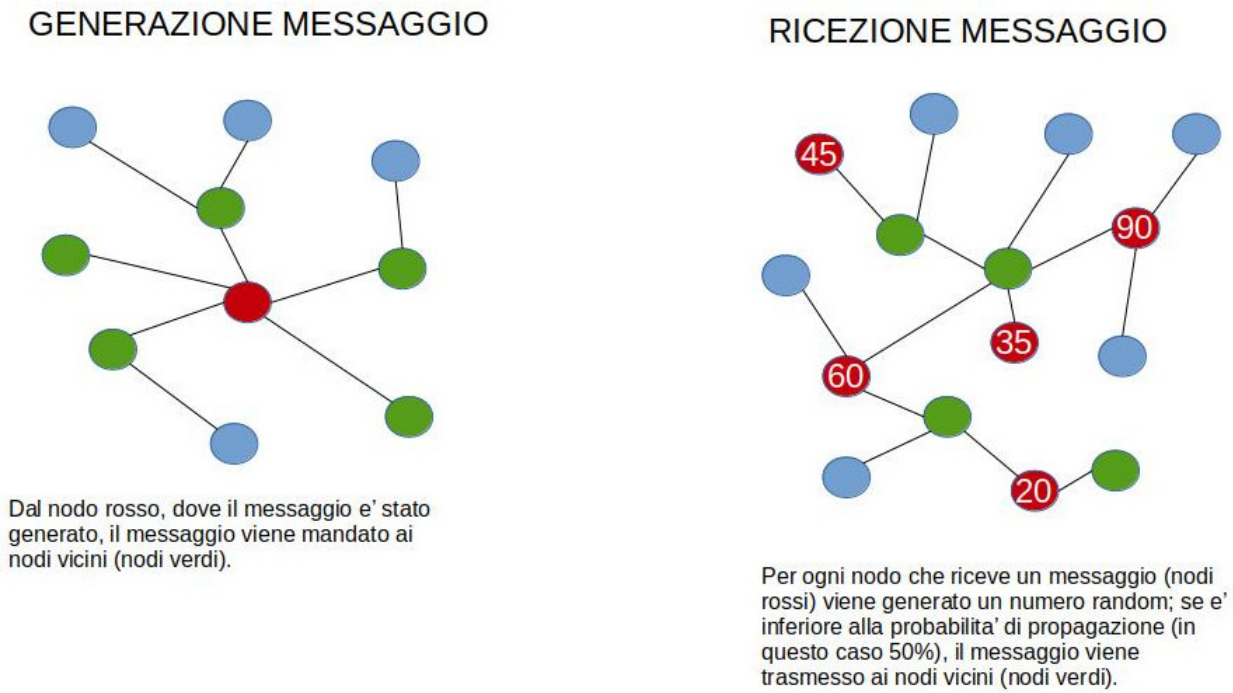

Figura 8: Esempio di Trasmissione Probabilistica

3.2 Probabilità Fissa di Disseminazione (Fixed Probability of Dissemination)

In questo protocollo il comportamento è leggermente diverso: ogni volta che viene chiamata la procedura di gossip, il confronto con la probabilità di propagazione viene fatto per ogni nodo vicino. Questo significa che il messaggio non viene automaticamente spedito a tutti i nodi vicini (la probabilità di propagazione è sempre modificabile tramite il parametro DISPROB.

Algoritmo 2 Probabilità Fissa di Disseminazione [3]

```
1: function GOSSIP(msg)
2: for all neighbours do
3:     if random() < DISPROB then
4:         send(message, neighbour)
```

Figura 9: Algoritmo di Probabilità Fissa di Disseminazione

3.3 Funzione Dipendente dal Grado (Degree Dependent Function)

Quest'ultimo protocollo si basa invece su un concetto un po' più complesso: se il nodo ha 1 solo vicino oppure 2 è obbligato a propagare il messaggio ad essi. Questo per un ragionamento secondo il quale non farlo potrebbe impedire al messaggio di propagarsi in un'intera sezione del grafo. Per sapere però quanti vicini ha ogni nodo (di che grado è) dobbiamo supporre che essi si scambino informazioni circa il loro grado, passandosi questo dato all'interno dei messaggi che si trasmettono, in modo da non doverne generare altri. Siccome all'inizio della simulazione nessun nodo sa di che grado sono i suoi vicini, sarà costretto a propagare i messaggi a tutti.

Algoritmo 3 Funzione Dipendente dal Grado [3]

```
1: function GOSSIP(msg)
2: if BeginningSimulation then
3:     for all neighbours do
4:         send(message, neighbour, myDegree)
5: else
6:     if myDegree == 1 V myDegree == 2 then
7:         send(message, neighbour)
8:     else
9:         for all neighbours do
10:            if random() < 1/i^α then
11:                send(message, neighbour)
```

Figura 10: Algoritmo di Funzione Dipendente dal Grado

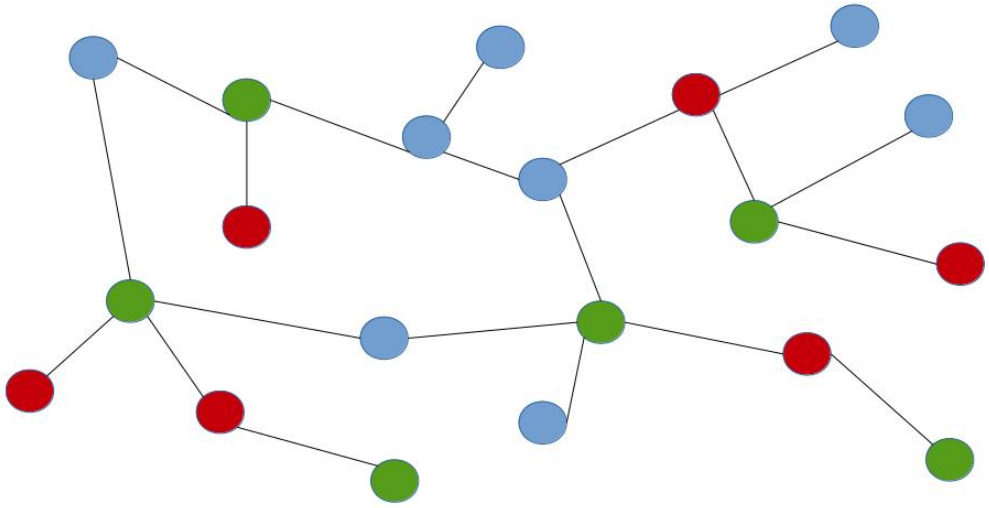

Figura 11: Esempio di Funzione Dipendente dal Grado

4 Test delle Performance

Tutte le modifiche apportate sono ovviamente state affiancate da svariati test effettuati per verificare se il lavoro svolto avesse effettivamente migliorato i tempi di esecuzione e, soprattutto, mantenuti inalterati i risultati. I test sono stati eseguiti su un Virtual Private Server (VPS) con 4 GB di RAM e 3 CPU da 2.4 Ghz, con sistema operativo Ubuntu 16.10 (Yakkety Yak). Ho scelto alcuni degli algoritmi già implementati in LUNES e ho eseguito alcuni test case su uno dei corpus già presenti tra quelli nella cartella **example-corpuses**, poichè avevo bisogno di controllare l'integrità dei risultati sullo stesso grafo. Per ogni test saranno riportati il nome del corpus utilizzato e i tempi d'esecuzione sia per il programma originale che per quello con le modifiche. Per verificare le tempistiche della simulazione mi sono avvalso di un comando presente su linux, time, con il quale si possono cronometrare parti di codice, grazie al quale vengono restituiti 3 valori:

- **Real**, che indica la quantità totale di tempo passata, comprese le porzioni di tempo usate da altri processi concorrenti;
- **User**, che indica il tempo che la CPU ha utilizzato per il nostro processo;
- **Sys**, che indica il tempo che il nostro processo ha impiegato in chiamate al kernel.

Sommando le tempistiche User e Sys abbiamo il tempo effettivamente impiegato dal nostro processo [10].
Secondo il modello implementato, quando viene chiamata la procedura di generazione ogni nodo può generare con una certa probabilità un solo messaggio; viene poi chiamata la procedura di gossip che si occupa di disseminarlo nel grafo tramite il protocollo da noi implementato; il messaggio viene inoltre inserito in una cache di dimensione da noi decisa. Quando invece un nodo riceve un messaggio, lo propaga nuovamente nel grafo, a meno che esso non sia già in cache. Se al momento di inserire un messaggio in cache, essa è già piena, viene

rimosso il messaggio più vecchio. Inoltre, ogni messaggio ha un TTL ad esso assegnato, che decresce progressivamente sino a raggiungere lo 0: a questo punto esso smette di essere propagato.

4.1 Test Trasmissione Probabilistica

Per ogni corpus utilizzato sono stati prese in considerazione tre casistiche, differenziate tra loro dal valore del parametro DISPROB. I tempi d'esecuzione riportati per ogni casistica sono in realtà il risultato di una media di 5 test.

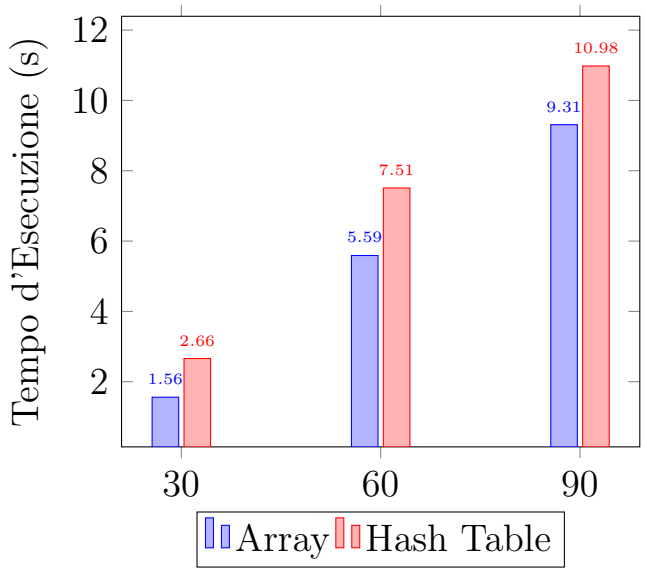

Figura 12: Performance Strutture Dati Trasmissione Probabilistica 1

kregular_corpus-500_vertex-1500_edges-diameter_6-10_graphs

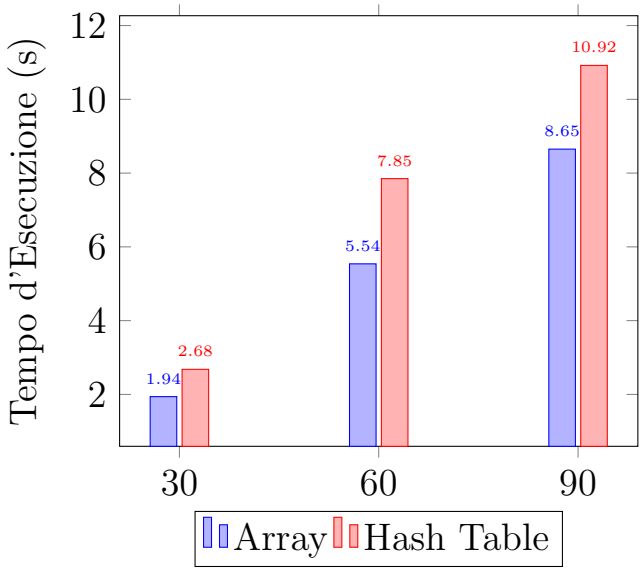

Figura 13: Performance Strutture Dati Trasmissione Probabilistica 2

smallworld_corpus-500_vertex-1500_edges-p_0_1-diameter_9-10_graphs

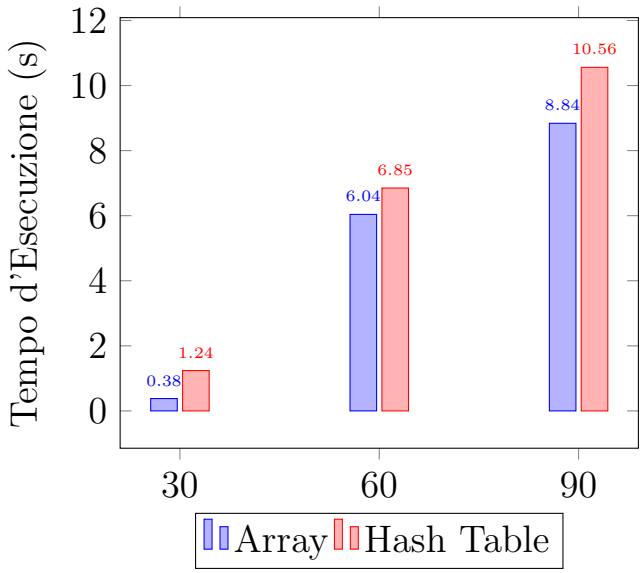

Figura 14: Performance Strutture Dati Trasmissione Probabilistica 3

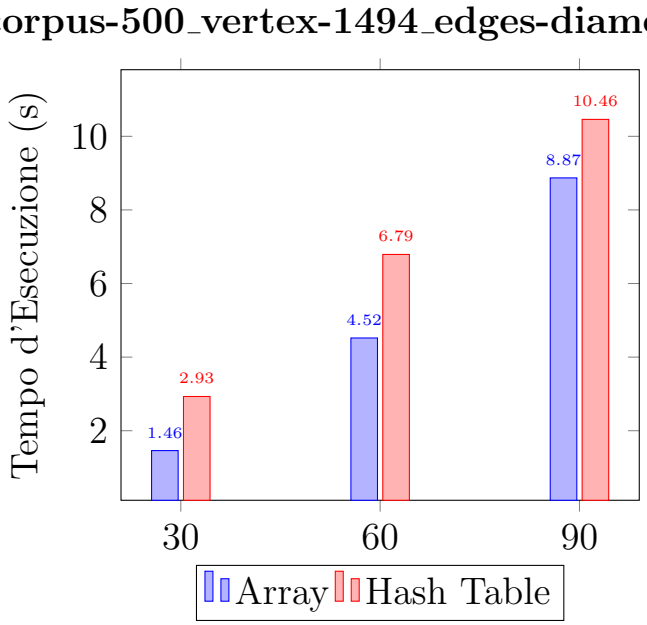

Figura 15: Performance Strutture Dati Trasmissione Probabilistica 4

random_corpus-500_vertex-1500_edges-diameter_7-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0003 s	<0.0001 s
60	0.0005 s	0.0001 s
90	0.0006 s	0.0001 s

kregular_corpus-500_vertex-1500_edges-diameter_6-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0002 s	<0.0001 s
60	0.0004 s	<0.0001 s
90	0.0005 s	0.0001 s

smallworld_corpus-500_vertex-1500_edges-p_0_1-diameter_9-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0004 s	<0.0001 s
60	0.0005 s	0.0001 s
90	0.0008 s	0.0001 s

scalefree_corpus-500_vertex-1494_edges-diameter_5-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0003 s	<0.0001 s
60	0.0003 s	0.0001 s
90	0.0006 s	0.0001 s

Tabella 1: Calcolo Media Trasmissione Probabilistica

4.2 Test Probabilità Fissa di Disseminazione

Per ogni corpus utilizzato sono stati prese in considerazione tre casistiche, differenziate tra loro dal valore del parametro DISPROB. I tempi d'esecuzione riportati per ogni casistica sono in realtà il risultato di una media di 5 test.

random_corpus-500_vertex-1500_edges-diameter_7-10_graphs

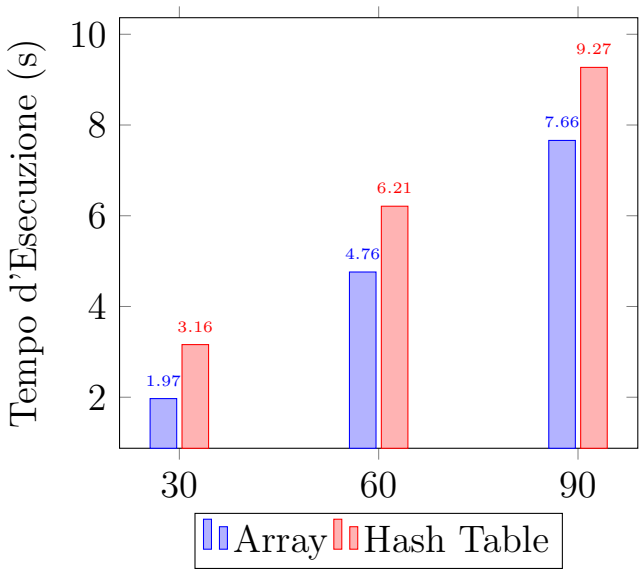

Figura 16: Performance Strutture Dati Probabilità Fissa 1

kregular_corpus-500_vertex-1500_edges-diameter_6-10_graphs

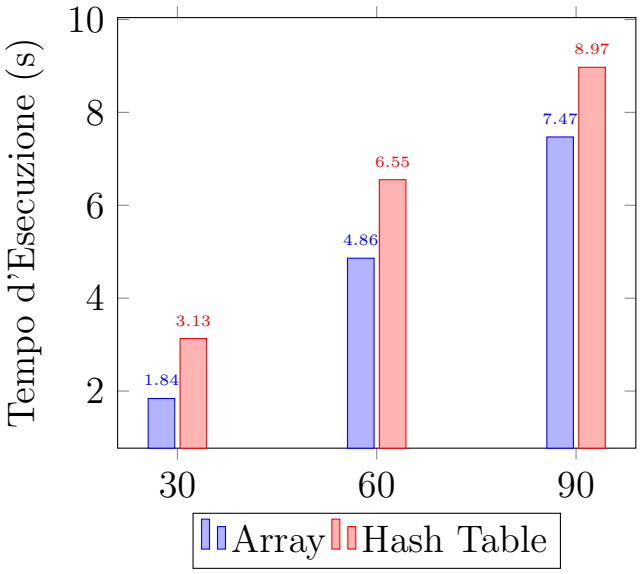

Figura 17: Performance Strutture Dati Probabilità Fissa 2

smallworld_corpus-500_vertex-1500_edges-p_0_1-diameter_9-10_graphs

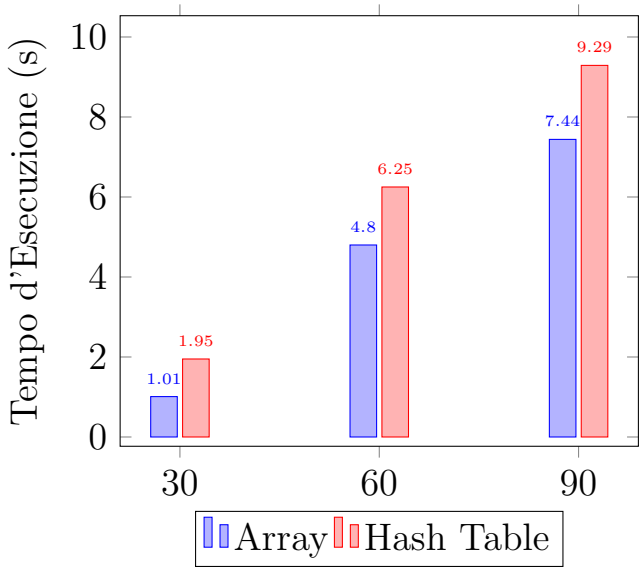

Figura 18: Performance Strutture Dati Probabilità Fissa 3

scalefree_corpus-500_vertex-1494_edges-diameter_5-10_graphs

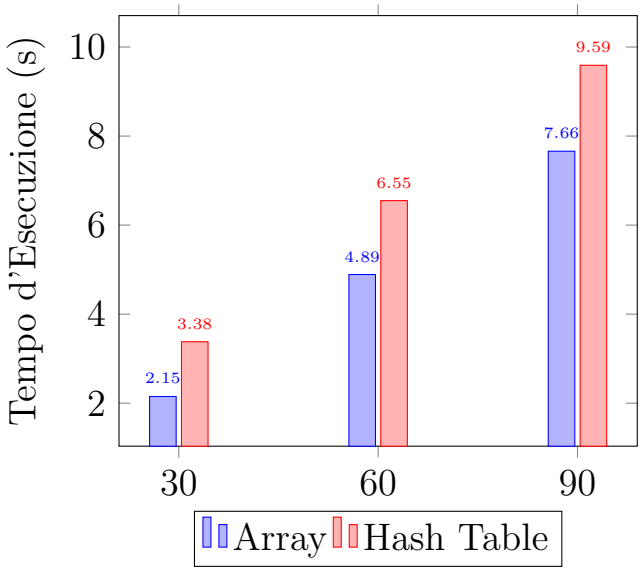

Figura 19: Performance Strutture Dati Probabilità Fissa 4

random_corpus-500_vertex-1500_edges-diameter_7-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0002 s	<0.0001 s
60	0.0003 s	<0.0001 s
90	0.0005 s	0.0001 s

kregular_corpus-500_vertex-1500_edges-diameter_6-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0001 s	<0.0001 s
60	0.0003 s	0.0001 s
90	0.0004 s	0.0001 s

smallworld_corpus-500_vertex-1500_edges-p_0_1-diameter_9-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0002 s	<0.0001 s
60	0.0005 s	0.0001 s
90	0.0006 s	0.0001 s

scalefree_corpus-500_vertex-1494_edges-diameter_5-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0003 s	<0.0001 s
60	0.0003 s	0.0001 s
90	0.0006 s	0.0001 s

Tabella 2: Calcolo Media Probabilità Fissa

4.3 Test Funzione Dipendente da Grado

Per ogni corpus utilizzato sono stati prese in considerazione tre casistiche, differenziate tra loro dal valore dal coefficiente α. I tempi d'esecuzione riportati per ogni casistica sono in realtà il risultato di una media di 5 test.

random_corpus-500_vertex-1500_edges-diameter_7-10_graphs

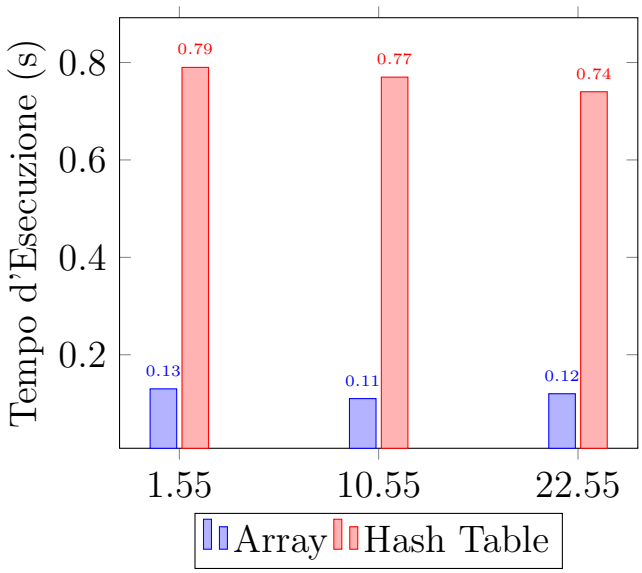

Figura 20: Performance Funzione Dipendente da Grado 1

kregular_corpus-500_vertex-1500_edges-diameter_6-10_graphs

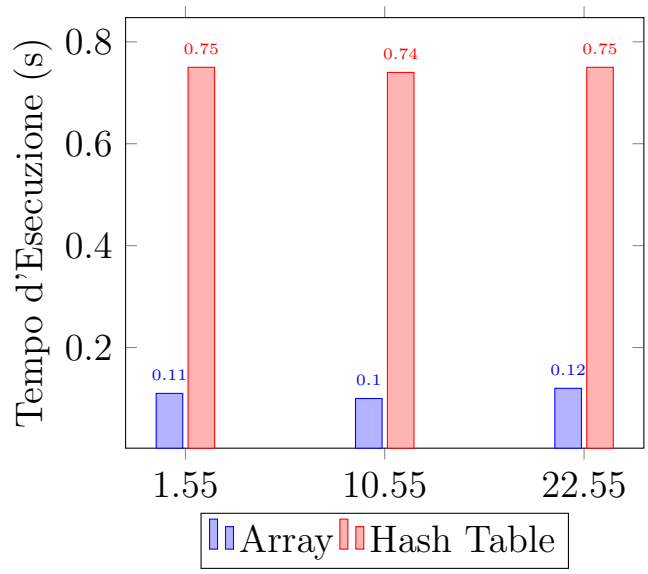

Figura 21: Performance Funzione Dipendente da Grado 2

smallworld_corpus-500_vertex-1500_edges-p_0_1-diameter_9-10_graphs

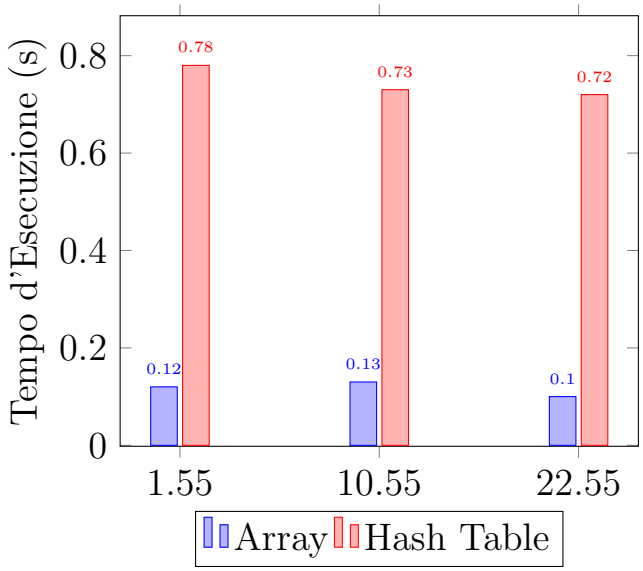

Figura 22: Performance Funzione Dipendente da Grado 3

scalefree_corpus-500_vertex-1494_edges-diameter_5-10_graphs

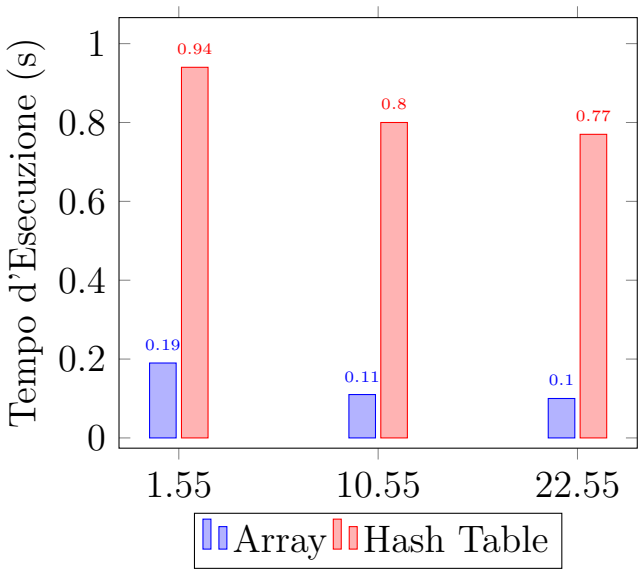

Figura 23: Performance Funzione Dipendente da Grado 4

random_corpus-500_vertex-1500_edges-diameter_7-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0001 s	<0.0001 s
60	0.0002 s	<0.0001 s
90	0.0001 s	<0.0001 s

kregular_corpus-500_vertex-1500_edges-diameter_6-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0002 s	<0.0001 s
60	0.0001 s	<0.0001 s
90	0.0002 s	<0.0001 s

smallworld_corpus-500_vertex-1500_edges-p_0_1-diameter_9-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0002 s	<0.0001 s
60	0.0001 s	<0.0001 s
90	0.0001 s	<0.0001 s

scalefree_corpus-500_vertex-1494_edges-diameter_5-10_graphs

DISPROB	Calcolo Media AWK	Calcolo Media C
30	0.0001 s	<0.0001 s
60	0.0002 s	<0.0001 s
90	0.0002 s	<0.0001 s

Tabella 3: Calcolo Media Probabilità Fissa

5 Conclusione

Dai risultati ottenuti possiamo evincere che la modifica effettuata per trasformare il file awk in sorgente C ha migliorato di alcuni millesimi di secondo il calcolo della media, dimostrando quindi come un linguaggio compilato sia più veloce in termini di esecuzione di uno interpretato.

La modifica alla struttura dati non ha invece apportato nessun miglioramento, ha anzi rallentato il tempo di esecuzione: questo probabilmente perchè, nonostante le premesse fatte in precedenza, all'interno dell'analisi delle statistiche l'array preesistente era utilizzato per ope-

razioni molto semplici (inserimento e confronto di valori). Quando si ha a che fare con operazioni così elementari e si conosce a priori l'indice dell'elemento a cui vogliamo accedere, l'hash table risulta più lenta, e questo è visible nei test riportati.

Altre modifiche future che potrebbero essere applicate a LUNES sono sicuramente un aumento dei commenti all'interno del codice, in quanto alcune parti sono bene documentate, altre per nulla.

Si potrebbe inoltre variare il modo in cui vengono letti e analizzati i file trace, in quanto in simulazioni di grandi dimensioni si otterrebbero ottimi miglioramenti.

Infine l'hash table è stata implementata utilizzando le librerie di Gnome, in quanto erano già utilizzate per altri scopi all'interno del software: si potrebbe provare a utilizzare altre librerie o ad implementarle ex novo, per verificare se è possibile utilizzare questa o altre strutture dati per sostituire il semplice array attualmente utilizzato.

Riferimenti bibliografici

[1] Gabriele D'Angelo and Stefano Ferretti. LUNES: Agent-based simulation of P2P systems. In *Proceedings of the International Workshop on Modeling and Simulation of Peer-to-Peer Architectures and Systems (MOSPAS 2011)*. IEEE, 2011.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[3] Luciano Bononi, Michele Bracuto, Gabriele D'Angelo, and Lorenzo Donatiello. Scalable and efficient parallel and distributed simulation of complex, dynamic and mobile systems. In *Proceedings of the 2005 Workshop on Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems*, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Luciano Bononi, Michele Bracuto, Gabriele D'Angelo, and Lorenzo Donatiello. Artìs: a parallel and distributed simulation middleware for performance evaluator. In *Proceedings of the 2004 Workshop of the 19-th International Symposium on Computer and Information Sciences*, pages 627–637. Springer, 2004.

[5] Gabriele D'Angelo and Michele Bracuto. Distributed simulation of large-scale and detailed models. *International Journal of Simulation and Process Modelling (IJSPM)*, 5(2):120–131, 2009.

[6] Gabriele D'Angelo. Gaia apis, 2011. Consultabile presso `http://pads.cs.unibo.it/doku.php?id=pads:gaia-apis`.

[7] The igraph core team, 2003. Consultabile presso `http://igraph.org/redirect.html`.

[8] Gabriele D'Angelo and Stefano Ferretti. Adaptive event dissemination for peer-to-peer multiplayer online games. In *2nd Workshop on DIstributed SImulation and Online gaming (ICST/CREATE-NET DISIO 2011)*. IEEE, 2011.

[9] The GNOME Project. Glib hash table documentation, 2014. Consultabile presso https://developer.gnome.org/glib/stable/glib-Hash-Tables.html.

[10] time (1) linux user's manual, 1993. Consultabile presso https://linux.die.net/man/1/time.