ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

SVILUPPO DI SOLUZIONI PER IL BILANCIAMENTO DEL TRAFFICO DI RETE SU SISTEMI P2P CON INDICIZZAZIONE DISTRIBUITA DEI DATI

Elaborata nel corso di: Sistemi Informativi Distribuiti LS

Tesi di Laurea di:
JIMMY CICCOLINI

Relatore:
Prof. GIANLUCA MORO

Correlatore:
Ing. TOMMASO PIRINI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

P2P

Bilanciamento

Traffico

Reti

GGrid

...alla mia fidanzata Alessia,
alla mia famiglia
e a tutti coloro che mi hanno sostenuto nel
raggiungimento di questo importante obiettivo.

Indice

Introduzione

L'uomo passa gran parte della sua esistenza a ricercare.
Gran parte delle sue ricerche sono finalizzate al rinvenimento di informazioni, a lui utili. Non possiamo non citare Internet, la cui connettività ha raggiunto la quasi totalità delle persone e i cui accessi stanno aumentando in maniera esponenziale. Come esponenziale è diventata la richiesta di informazioni, che ha portato gli individui a socializzare dati inizialmente posseduti da un singolo, così da creare una sorta di grande rete di comunicazione e condivisione.
Il meccanismo appena descritto è, sostanzialmente, quello delle reti *peer-to-peer* (in breve *p2p*). Quest'ultime sono un insieme di nodi, tutti allo stesso livello gerarchico, diversamente alle classiche strutture *client-server*. Nelle reti *p2p* ogni utente che si collega mette a disposizione i propri dati e può sfruttare la rete stessa per ricercare dati condivisi da altri utenti. L'utilizzo sempre più smodato di queste reti ha attirato l'attenzione degli studiosi delle reti informatiche che hanno iniziato ad osservare la loro evoluzione e ad interferire con essa apportando dei miglioramenti. Non essendoci un ruolo centrale all'interno della rete, i collegamenti tra computer sono semplici collegamenti punto-punto. Con l'aumento degli utenti, e quindi con l'aumento delle informazioni presenti nella rete, si è osservato che il problema principale era quello di ottenere l'informazione, se presente, in un numero finito e limitato di operazioni. In effetti, se un'informazione ricercata da un utente si trova su una macchina lontana dal richiedente, il messaggio che ricerca l'informazione potrebbe navigare a lungo all'interno della rete, creando un traffico eccessivo che limiterebbe la banda ad altri utenti. Per questo motivo, si è cercato di organizzare le reti *p2p* in modo da porre un limite al numero di passi medi necessari al raggiungimento delle informazioni di tutta la rete.

Lo schema organizzativo della rete, detto *overlay*, permette di avere un numero di *hop* (passi medi per raggiungere le informazioni) molto limitato e che, spesso, si avvicina al valore logaritmico del numero degli utenti connessi ad una rete. Ovviamente per ottenere una struttura organizzata è necessario che i *peer*, ossia gli utenti connessi, si scambino informazioni riguardo il mantenimento della struttura ed i vincoli correlati. Queste informazioni viaggiano su messaggi, e quindi creano traffico sulla rete stessa. Inoltre, ogni volta che un *peer* si connette o disconnette, probabilmente violerà i vincoli di struttura, obbligando una ristrutturazione dell'*overlay*. Ovviamente è importante limitare il crescere di questi messaggi con il dinamismo degli utenti che si collegano e si disconnettono dalla rete. Una volta limitato il numero di *hop*, si è cominciato ad osservare un secondo, ma non meno importante, parametro valutativo di queste reti: il bilanciamento del traffico. All'interno della rete, infatti, potrebbero esserci *peer* con più informazioni di altri, o semplicemente con informazioni più importanti, incentrando la maggior parte del traffico sul computer della rete che mantiene questi dati. Questa osservazione porta ad avere un collo di bottiglia della rete, in quanto il *peer* che dovrà soddisfare molte richieste avrà le proprie risorse impegnate per molto tempo e rallenterà l'evasione delle richieste stesse. Se da una parte sono state proposte diverse tipologie di *overlay* incentrate sul miglioramento delle caratteristiche in termini di *hop*, dall'altra vi sono ancora poche soluzioni attive per bilanciare il traffico di lavoro dei *peer* della rete.
In questo elaborato verranno presentate e realizzate alcune strutture di *overlay*, che verranno poi sperimentalmente, analizzate e confrontate per osservare il bilanciamento del carico di lavoro, oltre che, ovviamente, le caratteristiche di base quali numero di *hop* e costo di manteniemnto della struttura stessa.
Nel primo capitolo saranno menzionate alcune strutture di *overlay* attuali.
Nel secondo capitolo sarà presentato in dettaglio la struttura chiamata SkipCluster.
Nel terzo capitolo vi saranno quattro implementazioni di una struttura di *overlay* chiamata GGrid ed ideata dal professor Gianluca Moro dell'Università di Bologna.
Nel quarto capitolo sarà presentata la piattaforma di esecuzione degli

esperimenti, gli scenari sperimentati, le variabili misurate e i risultati delle simulazioni.
Il quinto capitolo concluderà il lavoro con alcune considerazioni e includerà spunti per lavori futuri.
Vi sarà poi un'appendice che tratterà lo studio dei collegamenti casuali ed il piccolo mondo, ripercorrendo la storia di queste teorie dai primi esperimenti agli studi attuali.

Capitolo 1

Stato dell'arte

La crescente espansione in merito all'utilizzo delle reti *p2p* ha focalizzato l'attenzione sullo studio delle loro architetture al fine di poter costruire reti più efficienti, migliorandone alcune caratteristiche tecniche.
In questo primo capitolo faremo un excursus di alcune delle strutture più note in termini di reti *p2p*, mostrandone l'architettura e le caratteristiche che le contraddistinguono. Arriveremo, inoltre, ad evidenziare un male comune: lo sbilanciamento del traffico di lavoro. Concluderemo il capitolo con il selezionare due delle strutture precedentemente analizzate, sulle quali condurremo un'attenta analisi in merito alla suddivisione del traffico di rete che si genera al proprio interno, nonchè altri importanti fattori tra cui il costo di mantenimento e la lunghezza del cammino medio per il raggiungimento degli obiettivi.

1.1 Panoramica sulle strutture attuali

Le soluzioni di *overlay* più avanzate [21,22,23,24,25,26] hanno caratteristiche molto diverse tra loro, alcune sono molto organizzate, altre molto meno, portando, ovviamente, a parametri valutativi differenti. Una caratteristica che li accomuna è l'utilizzo di un algoritmo Greedy per raggiungere la soluzione. Il termine Greedy fa riferimento all'algoritmo di ricerca dell'ottimo globale. Ad ogni iterazione, l'algoritmo avanza nella direzione migliore raggiungendo un ottimo locale in tempi polinomiali.

Come primo *overlay* vediamo GGrid[1]. Questa architettura è stata realizzata dall'Università di Bologna, in collaborazione con l'Università di Chicago, e tra i responsabili del progetto troviamo il professor Gianluca Moro. GGrid, che vedremo descritto approfonditamente nei successivi capitoli, presenta una struttura con due tipologie di attori: gli S-Peer, peer che mantengono dei dati al loro interno e che fanno parte di un indice distribuito, e i C-Peer, attori che hanno solo il compito di effettuare query distribuite nella rete. I dati vengono quindi memorizzati su di un S-Peer centrale che, al raggiungimento di una soglia di memorizzazione, sposta i propri dati in eccesso ad un C-Peer. Quest'ultimo, essendo a questo punto manutentore di dati, diventa un S-Peer ed entra quindi a far parte della struttura di indici distribuiti. Ad una prima analisi, senza entrare nei dettagli, notiamo che il raggiungimento del dato trovato, in termini di *hop* medi, si aggira sul valore logaritmico del numero di S-Peer presenti nella rete. Il costo di mantenimento è esiguo in quanto l'*overlay* costituisce una struttura ad albero binario i cui collegamenti si limitano ad essere tra padre e figli.
Il secondo *overlay* che presento è SkipCluster[3]. Anche questo *overlay* verrà descritto dettagliatamente nel capitolo successivo. In questa sezione ci limiteremo a dire che la rete viene divisa in *cluster*, ossia un insieme di nodi con la parte più significativa dell'identificatore in comune, ciascuno dei quali ha a capo un attore chiamato super-peer. Il super-peer è responsabile dei collegamenti tra il suo *cluster* e gli altri super-peer e del corretto mantenimento delle tabelle di *routing* dei *peer* interni al proprio *cluster*. Il super-peer è, quindi, l'unica via d'accesso per accedere ad un *peer* situato in un *cluster* diverso dal *cluster* di appartenenza. Come vedremo successivamente anche in questo caso il numero medio di *hop* cresce in maniera logaritmica rispetto al numero di entità presenti sulla rete.
Un *overlay* che si discosta dai precedenti in termini di organizzazione della struttura è Fuzzynet[4,20]. Questo algoritmo sfrutta le teorie alla base delle reti piccolo mondo, che vedremo nell'appendice A. L'algoritmo utilizzato per creare la rete piccolo mondo è l'algoritmo di Oscar[5]. Le novità apportate da Fuzzynet sono visibili in particolare nell'inserimento e nelle ricerche delle informazioni nella rete. Oscar è un algoritmo che si applica principalmente in caso di reti di dati con

distribuzione delle chiavi non uniforme. Ossia, ogni peer della rete organizzata mediante l'algoritmo Oscar, crea dei collegamenti con alcuni *peer* vicini ed un numero predefinito, pari al valore logaritmico dei nodi della rete, di *link* con *peer* lontani. Per creare i *link* di lungo raggio, in modo che ogni *peer* abbia lo stesso numero di *link* in ingresso, viene utilizzata la tecnica dei *randomwalkers*. Questa tecnica vede un *peer* inviare dei messaggi-sonda, i quali navigano casualmente sulla rete e rilevano la distribuzione delle chiavi. In questo modo, il *peer* che ha inviato le sonde ha il quadro approssimato della distribuzione delle chiavi di tutta la rete. A questo punto, suddivide lo spazio in zone di dimensioni logaritmiche ed invia un messaggio per ogni zona. Questo messaggio naviga in maniera casuale nella zona a cui è stato assegnato e, finito un numero di passaggi, invia, al *peer* che lo ha generato, il collegamento con un nodo della zona su cui era stato inviato. L'operazione di creazione dei *link* lontani viene eseguita ogni volta che un *peer* esegue un *join* nella rete. La novità portata da Fuzzynet, come abbiamo detto prima, è però il modo di ricerca e scrittura dei dati. La ricerca viene effettuata attraverso un algoritmo Greedy che restituirà con buona probabilità il dato ricercato. L'algoritmo Greedy potrebbe entrare in un piccolo mondo vicino ma non collegato direttamente con la soluzione. Per definizione, infatti, un algoritmo Greedy non torna sui suoi passi e potrebbe sospendere le ricerche con esito negativo (informazione non trovata) se non trova passi migliorativi. La percentuale indicata dagli sviluppatori è piuttosto elevata, di circa il 96%. La scrittura di un dato avviene in due fasi. Viene, innanzitutto, eseguita una lettura per stabilire qual è il *peer* che dovrà mantenere il dato. Il *peer* candidato è un nodo che mantiene una chiave simile a quella del dato da inserire. Una volta stabilito il *peer* destinatario del dato, questo viene scritto su di esso. A questo punto, il *peer* invia delle repliche ai *peer* vicini che, attraverso un fattore prestabilito, continuano la propagazione del dato nel vicinato. In questo modo, le letture successive non hanno bisogno di raggiungere un *peer* specifico ma possono restituire la prima replica utile del dato. Dal momento che le repliche di un dato non sono collegate tra loro vi è il rischio di non avere l'uniformità del dato a seguito di un'operazione di *update*. Durante un *update*, infatti, non tutte le repliche potrebbero essere sovrascritte. Una proposta degli autori è quella di inserire un *time-*

to-live, a seguito del quale la replica viene considerata obsoleta e, di conseguenza, eliminata. Un altro problema è l'aggiornamento dei *link* dei *peer*. Questo *overlay* porta ad avere costi di ingresso bassi, dovuti dalla creazione dei *link* lontani, e di uscita nulli. Gli autori non indicano una modalità specifica di aggiornamento dei *link*. Si intuisce però che l'uniformità della distribuzione dei *link* lontani verrà mantenuta in scenari dinamici, ossia di ingressi e uscite di nodi elevati, solo con un aggiornamento periodico dei *link* lontani.
Infine menzioniamo Mercury[6] come *overlay* di reti capaci di supportare *query* multi-attributo. Mercury organizza i propri dati creando un anello di *overlay* per ogni attributo. Ovviamente un dato con più attributi dovrà essere inserito negli anelli (detti anche *hub*) di cui è composto, portando ad una replicazione del dato sparsa in *peer* non contigui della rete. Anche le ricerche verranno effettuate seguendo l'attributo, e quindi l'*hub*, più vincolante all'interno della *query* generata. Ogni nodo del singolo *hub* avrà collegamenti con successore e predecessore nell'anello ed avrà una serie di *link* di tipo *long-range*, ossia a lungo raggio. Per costruire un *link* lontano un nodo A genera un valore x tra 0 ed il numero di nodi dell'anello, utilizzando la distrubuzione armonica. Verrà, quindi creato un collegamento con il nodo di distanza x da A. Inoltre, un nodo avrà collegamenti con dei rappresentanti di tutti gli altri *hub*. I rappresentanti vengono scelti con dei *random walkers* circoscritti all'interno dei singoli *hub*. Ogni volta che un nodo si connette dovrà scegliere un hub di appartenenza in maniera casuale e si inserirà nella struttura ad anello relativa. Dovrà in seguito creare i propri *link* lontani ed i *link* ai rappresentanti degli altri *hub*. Al momento della disconnessione dovrà ripristinare l'anello dell'*hub* di appartenenza prima di lasciare la sua posizione.

1.2 Il problema del bilanciamento

Quello che avevamo annunciato essere un problema comune a tutte queste soluzioni è lo sbilanciamento del carico di lavoro [19].
In GGrid, per inviare un messaggio ad un *peer* di pari altezza, si è costretti a percorrere in alto e in basso l'albero. Questo porta ad avere un traffico tanto maggiore quanto più è vicina alla radice la posizione

di un *peer*. Inversamente, un *peer* che risiede nelle foglie dell'albero, verrà contattato solo da messaggi di cui è lui stesso il destinatario.
Anche SkipCluster presenta il medesimo problema. Infatti, un super-peer, essendo l'unico abilitato ad inoltrare i messaggi da un *cluster* ad un altro, sarà costretto a smistare un numero di messaggio più elevato rispetto ai *peer* interni ad un *cluster*, contattati solo per comunicazioni interne al *cluster* stesso. Ovviamente, come si evince dai risultati riportati nell'articolo [3], la distribuzione del traffico dipenderà fortemente dalla dimensione del *cluster*.
Mercury propone una soluzione per migliorare la distribuzione del carico costruendo i propri *link* lontani non più in maniera casuale ma inviando più *link* ai *peer* meno carichi. Per fare questo ha bisogno di conoscere approssimativamente la distribuzione del traffico nella rete. Un metodo proposto è quello di inviare *random walkers* che raccolgano informazioni di traffico e le riportano al generatore dei camminatori. Ovviamente maggiori sono i camminatori generati maggiore è l'approssimazione del grafico di distribuzione del traffico dell'intera rete. Questo porta, però, ad un aumento del numero di messaggi di costruzione, e, quindi, di sistema. Ovviamente i grafici vanno mantenuti periodicamente per fornire la distribuzione attuale corretta del traffico. La periodicità incide, insieme al numero di *walkers*, sul costo di mantenimento della rete.
Dopo aver studiato le possibili strutture che mantengono ordinata la ricerca sulle reti *peer-to-peer*, gli studiosi sono passati ad analizzare la distribuzione del carico dei singoli *peer*, con l'obiettivo di rendere il più possibile omogeneo l'utilizzo delle risorse dei *peer*, come riportato dagli articoli [12] e [13]. L'esigenza nasce dall'osservazione di colli di bottiglia dovuti alla non uniformità di elementi mantenuti per ogni attore. Per esempio, basta pensare ad una struttura in cui i *peer* si suddividono aree di interesse in maniera piuttosto simmetrica, ma la maggior parte di elementi si concentra, in base alla loro chiave, in una singola area. Questo ovviamente porterà maggior carico, e quindi, maggior traffico, sulla macchina che mantiene tali oggetti. Le reti *peer-to-peer*, inoltre, hanno attori che potrebbero essere anche molto diversi tra loro, basti pensare che nelle reti attuali vi sono computer di ultima generazione affiancati a computer di qualche anno fa. Il collo di bottiglia precedentemente citato si accentua se si ipotizza che l'attore

con la maggior parte degli elementi è un vecchio computer con capacità di elaborazione limitate. Per porre rimedio, gli studi condotti attualmente si stanno focalizzando su particolari variabili, quali la capacità di memorizzazione dei *peer*, la banda e la capacità computazionale. Ovviamente non è possibile rendere omogenei tutti i fattori contemporaneamente, quindi si è primariamente cercato di rendere omogenea la distribuzione del carico. Ne deriva, che una distribuzione uniforme del carico porterebbe al quasi raggiungimento dell'uniformità di distribuzione del traffico di rete. Riportandoci all'esempio precedente, se gli elementi di una rete fossero posizionati uniformemente sulle aree di competenza dei singoli *peer*, questi si dividerebbero anche il traffico.
Per suddividere uniformemente il carico, studi recenti propongono di mantenere un grafico all'interno di ogni elemento che mostra al *peer* l'attuale distribuzione degli elementi. In questo modo, se un *peer* non mantiene elementi, o ne mantiene pochi, potrebbe passare i propri elementi ad un *peer* vicino e spostarsi a fianco ad un *peer* sovraccarico di oggetti, dividendo con esso la zona di competenza. Ovviamente per mantenere questo grafico sono necessari messaggi di sistema che costruiscono ed aggiornano il grafico. Per limitare il crescere di questi messaggi si può pensare di ottendere delle stime inviando dei messaggi ad un numero limitato di *peer* con lo scopo di reperire informazioni. Quanti più *peer* sono contattati, tanto più la stima si avvicina ad un valore reale di carico, con il difetto che il numero di messaggi di rete per mantenere la struttura aumenterebbe vertiginosamente.

1.3 HiGLoB

Una valida proposta al problema della distribuzione del carico arriva nel 2009 con la presentazione di un *framework* chiamato HiGLoB[7], acronimo di Histogram-based Global LOad Balance. Dal nome, questo *framework* fornisce direttive su come costruire un grafico della distribuzione degli elementi per permettere una ristrutturazione che suddivida meglio gli oggetti presenti nella rete.
HiGLoB è costituito da due elementi principali: il gestore dell'istogramma ed il gestore del bilanciamento del carico. Questo *framework*

permette di avere un grafico completo ed aggiornato, quindi senza approssimazioni.
Ogni *peer* suddivide la rete in tante zone quanti sono i suoi vicini. La particolarità è quella di trovare delle zone che non si sovrappongono. A questo punto un *peer* crea il proprio istogramma assemblando le informazioni di zona raccolte dai propri vicini, quali carico di dati, distribuzione del traffico, o altri parametri che si vogliono distribuire equamente. Successivamente invia informazioni ai vicini riguardo il proprio carico. L'aggiornamento a cascata che si limita alle zone di competenza divise idealmente da ogni *peer*, permette il mantenimento del grafico. Ad ogni variazione di carico un *peer* invia un messaggio di aggiornamento ai vicini che informeranno i propri vicini di zona e così via. Quando, dal grafico, un *peer* si accorge di essere sottocaricato, potrebbe inviare i propri dati ad un vicino ed effettuare un *re-join*, o ingresso sulla rete, su di un'altra zona. In questo modo il *peer* acquisisce parte degli elementi di un *peer* sovraccarico portando il sistema in una omogeneità di distribuzione degli oggetti presenti. L'utilizzo di questo *framework* è quasi come l'ingresso di un nuovo *overlay* e porta ad un aumento dei messaggi di rete scambiati per la costruzione e lo spostamento degli elementi. Gli autori del *framework* propongono una versione alternativa, in cui i nodi della rete inoltrano l'aggiornamento a cascata dell'istogramma solo se la differenza di carico supera una certa soglia.

1.4 Scelta degli *overlay* da sviluppare

Dopo aver elencato i principali studi sugli *overlay* delle reti *p2p* e sugli algoritmi di distribuzione omogenea del carico abbiamo scelto due strutture da realizzare: SkipCluster e GGrid. Per la realizzazione abbiamo adottato il simulatore di reti *peer-to-peer* PeerSim[11], come verrà descritto dettagliatamente nei capitoli successivi.
SkipCluster è stato scelto in quanto è uno studio molto recente, risale infatti al 2010, con una struttura organizzata. GGrid è stato scelto in quanto realizzato dall'Università di Bologna, e, soprattutto, in quanto vi sono stati aggiornamenti che intrinsecamente portano ad una distribuzione quasi del tutto uniforme del traffico di lavoro. In questo

modo si evita una seconda struttura di *overlay* che deve costruire e mantenere i valori di carico di tutti i *peer* della rete, evitando così ulteriori messaggi di sistema. Vedremo gli sviluppi dalla versione base alle versioni successive nel capitolo dedicato a GGrid.

Capitolo 2

SkipCluster

Il primo *overlay* che andiamo ad analizzare ed a testare è, come annunciato, SkipCluster, presentato dall'Università di Shangai nel 2011. Vedremo in questo capitolo le caratteristiche tecniche dell'architettura e dell'algoritmo di SkipCluster, estrapolate dalla rivista Computer Communications.

2.1 Introduzione

SkipCluster nasce come evoluzione di SkipNet, derivante a sua volta da SkipList[2]. Quest'ultimo è un *overlay* nato nel 1990 con la caratteristica di rappresentare i *peer* su alcune liste in diversi livelli. Il livello più basso di SkipList è il livello 0, costituito da un elenco di tutti i *peer* della rete. Salendo di livello si creano liste con meno *peer* fino ad avere un livello con lista vuota. Per ottenere un overaly di SkipList perfetto, ad ogni livello successivo si prendono i *peer* in maniera alternata dal livello sottostante. In questo modo, il routing inizia dal livello più alto, che permette di raggiungere i *peer* più lontani in pochi passi, per poi scendere sui livelli sottostanti diminuendo la distanza di collegamento tra i *peer*. Dal momento che non tutti i *peer* hanno collegamenti ad alti livelli, questo porta ad un aumento del numero di *hop*, in quanto, ad esempio, se un messaggio viene spedito da un *peer* presente solo al livello 0, questo porterebbe ad uno spostamento minimo del messaggio anche nella parte iniziale di inoltro, oltre che

nella parte finale. La figura 2.1 mostra un'esempio di architettura di SkipList.

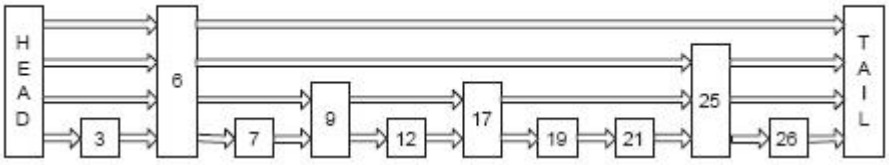


Figura 2.1: SkipList perfetta

L'evoluzione di SkipList è stata presentata nel 2003 con il nome di SkipNet[2]. Questa architettura prevede un anello completo di tutti i *peer* al livello 0. Ogni volta che si sale di livello l'anello viene diviso in due anelli similari per dimensione. Questa procedura viene ripetuta finchè non si raggiunge l'ultimo livello, in cui ogni *peer* era parte di un anello che comprende solo se stesso. La novità apportata da SkipNet rispetto a SkipList è che, questa volta, ogni *peer* ha almeno un collegamento con tutti i livelli, dando la possibilità di compiere grandi balzi verso i *peer* più lontani già dall'invio del messaggio, avvicinando subito il messaggio almeno a metà della distanza con il *peer* destinatario. In SkipNet, un *peer*, oltre ad essere riconosciuto attraverso il proprio identificativo all'interno della rete, è costituito da un altro codice che permette, oltre che a distinguerlo, di velocizzare le operazioni di routing. Infatti, il codice in questione, è costituito da una serie di 0 ed 1. Ognuna di queste cifre stabilisce, per ogni livello, se il *peer* appartiene all'anello di destra o di sinistra nel livello sottostante. Infatti, ad ogni livello, gli anelli hanno idealmente dei codici univoci che permettono di percorrere la strada dall'anello di livello 0, senza alcuna cifra identificativa, agli anelli di livello superiore, a cui viene aggiunta la cifra 0 o 1 se si tratta dell'anello di sinistra o di destra. Ovviamente, l'ultimo anello, è costituito da un singolo *peer*, e quindi questo prende lo stesso codice identificativo dell'anello di appartenenza. La figura 2.2 mostra un esempio di SkipNet con 8 *peer*.

Dopo aver descritto SkipList e SkipNet quali predecessori di SkipCluster, nel prossimo capitolo entreremo nel dettaglio di quest'ultimo *overlay*.

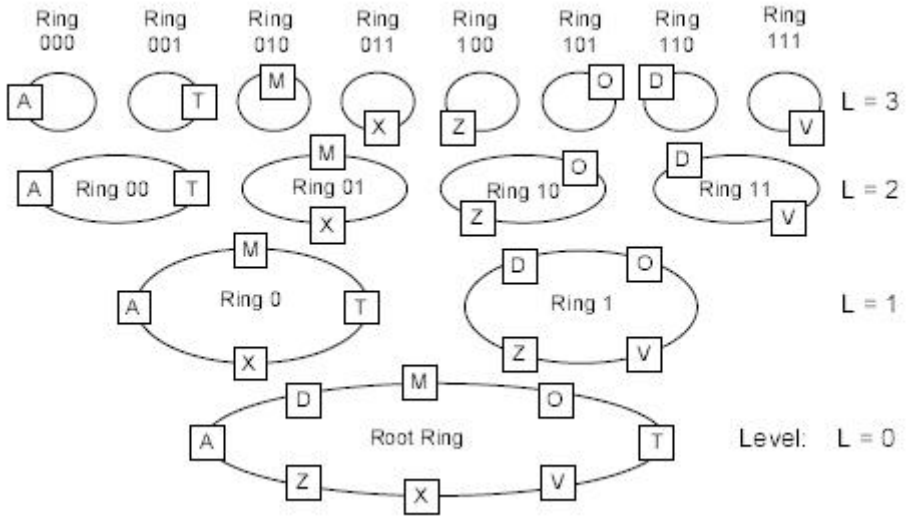


Figura 2.2: SkipNet con 8 *peer*

2.2 Architettura di rete

L'idea base di SkipCluster, come suggerisce la parola stessa, è quella di suddividere i *peer* in *cluster*. Per fare questo gli ideatori hanno diviso l'architettura in due macro-livelli: il livello alto ed il livello basso. Nel livello basso abbiamo i singoli *cluster*, isolati tra loro, mentre nel livello alto vediamo i *cluster* come entità atomica che viene collegata agli altri *cluster*.
Iniziamo con l'osservare il livello basso dell'architettura. Risulta importante stabilire, dal momento della costituzione della rete, la dimensione del *cluster*, ossia, qual è il numero massimo di *peer* appartenenti ad uno stesso *cluster*. Se consideriamo ora un identificativo dei *peer*, a livello della rete, costituito da una stringa binaria, possiamo dividere l'identificatore in due parti: la prima parte indica il *cluster* di appartenenza ed il secondo il posto all'interno del *cluster*. Infatti, se indichiamo una dimensione di k *peer* per *cluster*, gli ultimi $log(k)$ bit saranno sufficienti per identificare il *peer* in modo univoco all'interno di un *cluster*. I bit più significativi, invece, indicheranno il *cluster* di appartenenza che, di conseguenza, adotterà lo stesso valore come co-

dice identificativo di tutto il *cluster*. I *peer* all'interno di un singolo *cluster* si disporranno in un anello ordinati in base al loro identificatore interno al *cluster*. Il *peer* con l'identificatore intra-*cluster* minore sarà chiamato super-peer e sarà il responsabile delle comunicazioni con il livello superiore. Ogni volta che un *peer* vuole inviare un messaggio ad un destinatario che non fa parte di quel *cluster*, il *peer* dovrà inviare il messaggio al super-peer del *cluster* di appartenenza, che inoltrerà il messaggio al super-peer responsabile del *cluster* di appartenenza del *peer* destinatario, che provvederà ad inoltrare il messaggio al destinatario finale del messaggio. Per scambiarsi i messaggi all'interno del *cluster*, i *peer* utilizzano una struttura particolare. Ogni *peer* divide in due ipotetiche semirette, nelle due direzioni di navigazione dell'anello, gli altri *peer* dello stesso *cluster*. Le semirette vengono poi divise in segmenti di dimensione crescente in maniera esponenziale e viene preso come collegamento vicino l'ultimo *peer* effettivamente presente di ogni segmento. Questa operazione avviene in entrambe le direzioni dell'anello, quindi i *peer* avranno due serie di vicini, l'una con i *peer* con identificativo intra-*cluster* maggiore, l'altra con i *peer* con identificativo minore del *peer* che cerca i propri vicini. Il *peer*, inoltre, manterrà un collegamento con il proprio super-peer per rendere più rapide le comunicazioni esterne al *cluster* di appartenenza. Per capire meglio possiamo osservare il disegno in figura 2.3.
Passiamo ora ad osservare il livello alto dell'architettura. In questo livello, come abbiamo detto in precedenza, ogni *cluster* è rappresentato in maniera atomica, indipendentemente dai *peer* presenti all'interno del *cluster*. Per semplificare l'architettura, viene scelto il super-peer come rappresentante del *cluster*. In questo livello ogni super-peer sarà identificato dall'identificativo del *cluster* che rappresenta. La struttura di alto livello è costituita allo stesso modo di SkipNet, con la differenza che, invece che essere composta da tutti i *peer* della rete, l'architettura di alto livello è composta dai soli super-peer della rete. La figura 2.4 mostra la somiglianza con SkipNet dal punto di vista dell'architettura.

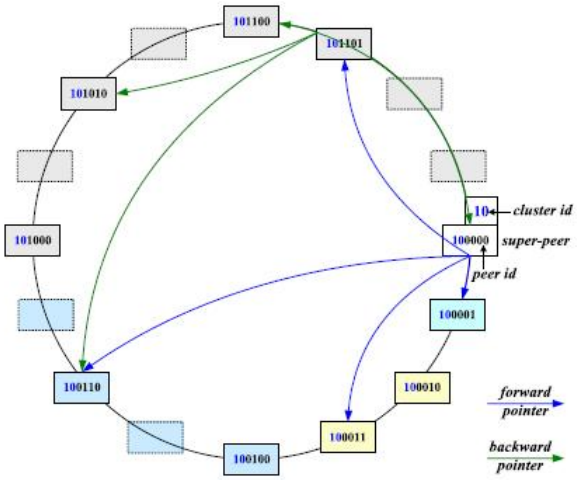


Figura 2.3: SkipCluster: organizzazione intra-*cluster*

2.3 *Join* e *leave* dei *peer*

L'inserimento e la disconnessione dei *peer* da SkipCluster è un momento delicato in quanto potrebbe creare delle modifiche alla struttura della rete. Si potrebbero infatti creare o rimuovere interi *cluster*, potrebbe cambiare il super-peer di riferimento di un singolo *cluster*, oppure, semplicemente, si potrebbero aggiungere o eliminare singoli *peer* internamente ad un *cluster*. Vediamo ora nel dettaglio come avviene l'ingresso di un *peer* nella rete e, in seguito, come avviene una disconnessione.
Per potersi connettere alla rete, un *peer* deve come prima operazione trovare il proprio *cluster* di destinazione a cui dovrà fare parte. Per fare questo, è necessario innanzitutto eseguire una *query* nel livello alto della infrastruttura, con la richiesta di ottenere il riferimento al super-peer responsabile del *cluster* identificato dalla parte più significativa dell'identificatore del *peer* che sta eseguendo il *join*. In base alla risposta ottenuta possiamo trovarci in una delle seguenti situazioni:

- il *cluster* di appartenenza non è ancora presente nella rete;

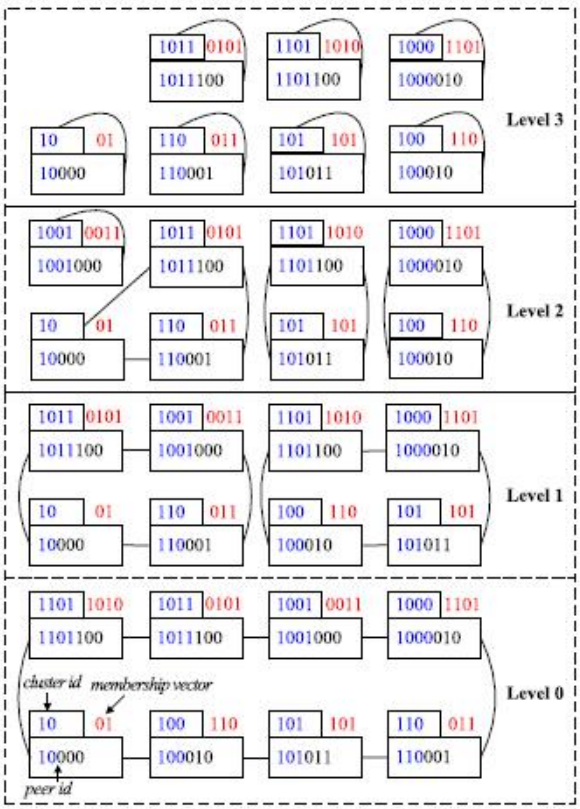


Figura 2.4: SkipCluster: organizzazione intra-*cluster*

- il *cluster* è presente ed il *peer* che si sta connettendo verrà situato al proprio interno;
- il *cluster* è presente ed il *peer* in ingresso sostituirà il super-peer attuale.

Se la *query* avrà esito negativo, quindi se non è stato trovato nessun super-peer con lo stesso identificatore di *cluster*, vorrà dire che non vi è ancora nessun *cluster* con quel determinato identificatore. Di conseguenza, il *peer* che sta eseguendo la connessione alla rete, sarà il responsabile della creazione del *cluster* relativo. Per fare ciò si setterà automaticamente come super-peer del *cluster* composto da se stesso ed eseguirà le operazioni per ottenere i contatti con altri *peer* della

rete. Dal momento che non vi sono altri *peer* nel proprio *cluster*, il nodo non avrà bisogno di ottenere le connessioni interne al *cluster*, ma necessiterà solamente delle connessioni di livello alto. Per fare ciò ipotizziamo che la rete sia costituita solamente dai super-peer, collegati con una struttura SkipNet. Le operazioni di connessione saranno le stesse che si effettuano durante la connessione di reti con architettura SkipNet. Il *peer* si inserirà, quindi, sull'anello di livello zero della rete, per poi ottenere i collegamenti con i livelli superiori.
Nel caso in cui, invece, la *query* di *join* otterrà esito positivo, il *peer* saprà che è già presente il suo *cluster* di destinazione e proverà a connettersi ad esso. Anche in questa situazione dovremmo distinguere due casi: il *peer* ha un identificatore interno al *cluster* maggiore di quello del super-peer, oppure, ha un identificatore minore di quello dell'attuale super-peer.
Nel primo caso, ossia quando il *peer* che sta eseguendo l'ingresso nella rete ha un identificatore intra-peer maggiore a quello del super-peer, il *peer* in fase di *join* cercherà la propria posizione all'interno dell'anello costituito dai soli *peer* dello stesso *cluster*. Una volta entrato nell'anello, e quindi una volta ottenuti i collegamenti con i pripri vicini, di destra e di sinistra, dividerà idealmente l'anello in due semirette, una con i *peer* di identificatori minori del proprio id, una con quelli con identificatori maggiori. A questo punto sarà sufficiente eseguire alcune *query*, con target di destinazione calcolato in modo da ottenere gli ultimi *peer* dei segmenti di dimensione esponenziale, per ciascuna direzione. Le *query* permetteranno la creazione dei collegamenti di livello basso della architettura. Il *peer* avrà anche, in seguito alla *query* di connessione iniziale, il collegamento con il proprio super-peer. Quest'ultimo collegamento, oltre che ad essere utilizzato per le *query* che escono dalla competenza del *cluster* al quale il *peer* appartiene, è utile per forzare l'aggiornamento dei *link* di livello basso. Infatti, l'ingresso di un *peer* nell'anello del *cluster*, causerà la rigenerazione dei *link* intra-*cluster* dell'intero anello. Il *peer* che ha eseguito il *join* invierà un messaggio al super-peer che trasporterà la richiesta di aggiornamento dei *link* interni all'anello del *cluster*. Tranne il successore ed il predecessore, infatti, i segmenti potrebbero essere cambiati. Il super-peer, essendo comunque parte dell'anello del *cluster*, eseguirà l'aggiornamento dei collegamenti intra-*cluster* ed inoltrerà al proprio

successore il messaggio di aggiornamento dei *link* interni al *cluster*. Il messaggio che causa l'aggiornamento dei *link* circolerà su tutto l'anello portando il *cluster* in una situazione di stabilità con tutti i collegamenti corretti in base alla definizione dell'architettura.
Vediamo ora il caso in cui il *peer* che entra nella rete abbia un identificativo interno al *cluster* minore dell'identificatore del super-peer. Come abbiamo detto, per definizione, il *peer* con identificatore minore all'interno del *cluster*, è chiamato super-peer. Nel caso in questione, quindi, il *peer* che sta eseguendo il *join*, dovrà diventare il nuovo super-peer, sostituendo quello precedente. Al momento della *query* iniziale, il super-peer che risponde alla richiesta può già osservare il verificarsi di questa situazione. Se il super-peer nota che il *peer* che sta cercando di entrare nella rete dovrà essere il nuovo responsabile del *cluster*, nel messaggio di risposta, gli invierà anche la propria tabella di routing di alto livello. In questo modo il *peer* che si sta collegando si setterà automaticamente come super-peer del *cluster* e, ricevendo la tabella di routing di alto livello, invierà ai suoi vicini di tutti i livelli alti, il nuovo indirizzo su cui inoltrare le richieste inter-*cluster*. In questo modo non sarà necessario rieseguire il *join* ad alto livello per lo stesso *cluster*. A questo punto, il nuovo super-peer si costruirà le tabelle di routing di basso livello e forzerà la costruzione di quelle di tutti i *peer* appartenenti al *cluster*, aggiornando anche il loro collegamento con il nuovo super-peer, identicamente a quanto visto nella situazione precedente.
Le procedure e le casistiche di disconnessione sono simili a quelle viste per la connessione:

- il *peer* che si disconnette è interno al *cluster* e non è un super-peer;
- il *peer* che si disconnette è il super-peer del proprio *cluster*;
- il *peer* che si disconnette è l'unico presente nel proprio *cluster*.

Se il *peer* che vuole disconnettersi è un *peer* interno al *cluster*, e quindi non un super-peer, questo notificherà l'evento ai vicini ed al super-peer prima di disconnettersi dalla rete. Quest'ultimo esegue la procedura di aggiornamento dei *link* interni al *cluster* esattamente allo stesso modo in cui si eseguiva l'aggiornamento a seguito di un *join*. Se il

peer che vuole eseguire la disconnessione è il super-peer e vi sono altri *peer* all'interno del *cluster*, questo invierà la propria tabella di routing al *peer* che lo segue nell'anello del *cluster* ed eseguirà una disconnessione. Il *peer* che riceva la tabella si setterà come nuovo super-peer, ereditando la tabella di alto livello. A questo punto, così come accade al cambio di un super-peer in fase di *join*, il nodo invierà le richieste di aggiornamento di alto livello e farà partire la procedura di aggiornamento dei *link* di basso livello. Nell'ultimo caso, ossia quando il *peer* che intende lasciare la rete è il super-peer ed unico nodo di un *cluster*, questo notificherà l'evento solamente ad alto livello, in cui verrà eseguita la procedura di disconnessione allo stesso modo in cui viene effettuata da SkipNet.
Negli esperimenti effettuati non sono state trattate le disconnessioni improvvise causate da, ad esempio, spegnimento delle macchine. Questo non permetterebbe di eseguire le procedure di disconnessione descritte. Quando però un *peer* si accorge, a causa della mancata risposta, che un proprio vicino non è più parte della rete, farà partire lui stesso le procedure. Se a disconnettersi bruscamente è un super-peer, il nuovo super-peer, non avendo ricevuto la tabella di alto livello, dovrà eseguire un *join* portando al ricalcolo di tutti i *link* andati persi dalla disconnessione.

2.4 Applicazione di HiGLoB

Viste le capacità di bilanciamento di HiGLoB descritte nel capitolo precedente, abbiamo applicato tale algoritmo all'architettura di SkipCluster.
Dal momento che utilizziamo una distribuzione uniforme dei dati nei vari *peer* che compongono la rete, abbiamo deciso di applicare SkipCluster in modo che si bilanciasse il numero di *peer* presente in ogni *cluster*. Un *cluster* di dimensioni elevate comporta, infatti, un elevato numero di collegamenti all'interno dell'anello del *cluster* stesso. Questo porterebbe ad avere una elevata disparità di numero di collegamenti dei vari *peer*, aumentando, di fatto, la memoria media necessaria ad ogni *peer* per mantenere la struttura. Inoltre sbilancierebbe il sistema in termini di operazioni di assestamento a seguito di connessioni

o disconnessioni nei *cluster* ad elevata densità di *peer*.
Avendo scelto di bilanciare il numero di *peer* per *cluster*, le zone di interesse sono proprio i singoli *cluster* che, per costruzione, non si sovrappongono. Inoltre, dal momento che solo i super-peer conoscono la struttura di alto livello, ossia a livello di *cluster*, abbiamo deciso di demandare ogni operazione specifica di HiGLoB nelle mani dei super-peer. Dal momento che lo scopo dell'esperimento è quello di bilanciare il numero di *peer* per ogni *cluster*, i *peer* interni al *cluster* sono considerati come oggetti che verranno scambiati tra i vari super-peer per bilanciare il sistema.
Vediamo, quindi, nel dettaglio, come abbiamo realizzato i due componenti fondamentali del framework proposto da Kian Lee Tan, adattandoli per lavorare nella struttura SkipCluster.
Il primo componente fondamentale è il costrutture dell'istogramma. Ogni *peer* dovrà, infatti, conoscere la situazione riguardante il numero di *peer* per ogni *cluster* della rete. Per fare questo, i super-peer invieranno dei messaggi nella rete in cui comunicheranno il numero di *peer* situati all'interno del proprio *cluster* e preleveranno informazioni riguardo gli altri *cluster*. Nello specifico, ogni volta che un nuovo *cluster* si viene a creare, ossia ogni volta che un *peer* che esegue il *join* si trova ad essere unico *peer*, e di conseguenza super-peer, del *cluster* di appartenenza, il nodo invierà al proprio vicino, inteso come il successore nell'anello di livello 0, un messaggio in cui cominicherà la propria presenza. Il nodo che riceve tale messaggio acquisirà l'informazione sul *cluster* appena creato e memorizzerà sul messaggio il *cluster* da esso gestito con il numero di *peer* presenti all'interno. Il messaggio sarà poi inviato al nodo successore nell'anello. Questa procedura aggiornerà tutti i super-peer e, una volta che il messaggio tornerà al mittente, cioè dopo un giro completo dell'anello, permetterà al *cluster* appena creato di prelevare le informazioni di carico di tutti gli altri *cluster*.
Come abbiamo spiegato in precedenza, ogni volta che un *peer* si connette alla rete esegue una *query* nel livello alto. Se il *cluster* è già stato creato, sarà il super-peer del *cluster* a cui dovrà collegarsi a rispondere alla richiesta di *join*. In questo modo il super-peer si accorge di un nuovo ingresso. Se il *peer* che sta eseguendo la connessione non diventerà il nuovo super-peer, l'attuale *peer* responsabile del *cluster* creerà

un messaggio che circolerà nell'anello di livello 0 andando ad aggiornare il grafico di tutti i super-peer, inserendo l'informazione sulla nuova quantità di nodi gestita dal super-peer che ha inviato il messaggio.
Se il *peer* che si sta collegando sarà il nuovo super-peer, il super-peer che lo ha preceduto fino a quel momento, oltre ad inviargli la tabella di routing di alto livello, gli invierà anche il proprio istogramma di carico. Il nuovo super-peer, a questo punto, invierà nell'anello 0, come in precedenza, un messaggio che notificherà l'aumento di dimensioni del proprio *cluster*.
La stessa procedura si ripete nella fase di disconnessione di un *peer*. Se il *peer* che si disconnette è interno ad un *cluster*, il relativo super-peer notificherà il cambio di dimensione agli altri super-peer come in precedenza. Se a disconnettersi è il super-peer ed il relativo *cluster* contiene almeno un altro nodo, oltre che ad inviare la tabella di routing di alto livello, il super-peer invierà al *peer* candidato anche l'istogramma. Il nuovo super-peer notificherà il cambio di dimensione del proprio *cluster* con un messaggio di update degli istogrammi. Infine, se a disconnettersi è un super-peer, unico nodo del proprio *cluster*, questo invierà un messaggio particolare in cui informerà gli altri super-peer di procedere all'eliminazione della zona dell'istogramma che rappresenta il *cluster* appena rimosso dalla rete.
Con queste procedure ogni super-peer sa esattamente la dimensione di tutti i *cluster* e può eseguire gli algoritmi di bilanciamento dei *peer*. Queste operazioni sono gestite dal secondo componente fondamentale del framework HiGLoB: il componente di bilanciamento. Per realizzare questo componente si devono trattare due problemi: quando entrerà in azione e come realizzerà il bilanciamento. Il primo problema è di facile risoluzione. Si esegue, poi, un controllo che fa eseguire le operazioni di bilanciamento alla ricezione dei messaggi di creazione e aggiornamento dell'istogramma. In particolare, quando un *peer* riceve un messaggio di aggiornamento o creazione dell'istogramma potrà calcolare la media dei *peer* per ogni *cluster* e valutare se il numero di nodi gestiti dal proprio *cluster* è superiore della media calcolata. Nel caso in cui si verifichi questa ipotesi il *peer* procederà al bilanciamento chiedendo lo spostamento di un certo numero di *peer* ad un *cluster* meno carico. Il super-peer, dopo aver calcolato quanti *peer* dovrà spostare, valore pari alla differenza tra i propri *peer* e la media

dei *peer* per *cluster*, inoltrerà il messaggio di creazione/aggiornamento dell'istogramma ricevuto in precedenza al successore nell'anello. Nel messaggio che inoltrerà il nodo vi inserisce anche il numero di *peer* che intende spostare ed in che zona. Nel messaggio sarà presente quindi una lista di *peer* che stanno per essere spostati ed in che zone. In questo modo, i *peer* che ricevono un messaggio di aggiornamento dell'istogramma, dovranno calcolare la media ed il *peer* meno carico aggiungendo anche le informazioni di passaggio dei *peer*. Questo evita un eccessivo spostamento di *peer*. L'esempio più significativo è la procedura di creazione di una zona dell'istogramma. Infatti, se un *peer* riceve l'inserimento di una zona, il relativo *cluster* sarà inizialmente composto da un solo *peer*. Il super-peer che riceverà il messaggio, probabilmente invierà un numero di *peer* al nuovo arrivato ed inoltrerà il messaggio. Se non si inglobasse nel messaggio l'informazione sui *peer* in movimento, il super-peer che riceverà il messaggio inoltrato invierà, con grande probabilità, ancora nodi al nuovo *cluster*, e così via. Al nuovo clustrer arriverebbero un numero elevato di *peer* che, probabilmente, sarà superiore alla media e provocherà ancora lo spostamento di nodi. In questo modo si avrebbe un eccessivo aumento di traffico di sistema prima di arrivare all'equilibrio. Inserendo nel messaggio di HiGLoB il numero di *peer* in movimento si ha dunque una rapida convergenza all'equilibrio.
Vediamo ora come opera con precisione il componente che esegue il bilanciamento vero e prorpio. Se un nodo, alla ricezione di un messaggio di HiGLoB, si accorge di essere sovraccarico, calcolerà il numero di *peer* da spostare in modo che quelli residui siano pari alla media, ed individuerà il *cluster* meno carico al quale invierà il numero di nodi intenzionato ad inviargli. Il nodo meno carico, alla ricezione della richiesta di spostamento appena inviata, calcolerà quanti nodi può ricevere senza che superino il massimo numero di *cluster* per *peer* impostato. Inoltre, il nodo, dovrà mantenere una lista con l'identificatore dei nodi appartenenti al proprio *cluster*. In questo modo, il *peer* meno carico, invierà a quello che ha inviato la richiesta di spostamento, direttamente gli identificativi dei *peer* che accetta nel proprio *cluster*. Gli identificativi inviati dovranno avere la caratteristica di appartenere al *cluster* di destinazione e di non essere già presenti sulla rete. Il nodo sovraccarico riceverà, quindi, una risposta alla sua richiesta

con i nuovi identificativi da assegnare ai *peer* che intende spostare. Il super-peer sceglierà, quindi, i *peer* da rimuovere dal proprio *cluster* in maniera casuale ed invierà loro un messaggio con all'interno il nuovo identificativo che dovranno assumere. I *peer* che ricevono questo messaggio eseguiranno una disconnessione temporanea dalla rete, cambieranno il proprio identificatore coon quello assegnato dal super-peer, ed eseguiranno un *re-join*. Con il nuovo identificatore, i *peer* si connetteranno direttamente all'interno del nuovo *cluster* bilanciando così il carico di *peer* per *cluster*.
Grazie all'utilizzo di HiGLoB, come vedremo dai risultati, otterremo una diminuzione del numero medio di *link*, diminuendo così la memoria media necessaria per mantenere la struttura di SkipCluster.
Ovviamente è interessante anche l'applicazione di HiGLoB in scenari senza distribuzione uniforme del carico, per osservare le migliorie apportate dall'algoritmo. Ma tralasciamo questa applicazione e ci limiteremo, in questo elaborato, ad osservare HiGLoB in ambiti diversi, per notare se vi sono comunque miglioramenti della struttura.
Nelle figure di seguito vedremo in dettaglio il codice delle principali funzioni che implementano la nostra versione di HiGLoB.

```
Hashtable<String,Integer> movingPeers=null;
if(msg.isDisconnection){

    //aggiornamento a seguito della disconnessione di un intero cluster
    this.histogram.PeersPerCluster.remove(msg.startClusterID);

    //eseguo il bilanciamento
    movingPeers=Balance(msg);
    if(!msg.endNode.equals(localnode)){
        Node next=this.routingTable.getNeighbor(Direction.RIGHT, 0);

        //se sto spostando alcuni peer ne memorizzo il numero nel messaggio
        if(movingPeers!=null)
            msg.peersInMoving.put(movingPeers.keys().nextElement(),
                    movingPeers.values().iterator().next());

        //invio al vicino di destra il messaggio di aggiornamento dell'istogramma
        sendMessage(next,msg);
    }
    return;
}
if(msg.startClusterID.equals(this.getClusterID())){
    //se il messaggio di aggiornamento ha completato il giro dell'anello tornando al mittente
    //memorizzo i dati relativi agli altri cluster memorizzati dagli altri super-peer
    for(String clusters:msg.histogramStack.keySet())
        this.histogram.PeersPerCluster.put(clusters, msg.histogramStack.get(clusters));
    //Eseguo il bilanciamento
    Balance(msg);
}else{
    //aggiorno l'istogramma con i dati del mittente del messaggio
    this.histogram.PeersPerCluster.put(msg.startClusterID, msg.nPeers);
    //aggiungo i dati relativi al cluster di cui sono responsabile nel messaggio
    msg.histogramStack.put(getClusterID(), this.histogram.nPeers);
    //eseguo il bilanciamento
    movingPeers=Balance(msg);
    //se devo spostare alcuni peer ne memorizzo il numero nel messaggio
    if (movingPeers!=null)
        msg.peersInMoving.put(movingPeers.keys().nextElement(), movingPeers.values()
                .iterator().next());
    //invio il messaggio di aggiornamento dell'istogramma al vicino di destra
    Node next=this.routingTable.getNeighbor(Direction.RIGHT, 0);
    sendMessage(next,msg);
}
```

Figura 2.5: Funzione che descrive il comportamento di un *peer* che riceve un messaggio di aggiornamento dell'istogramma

```
Hashtable<String,Integer> peersToMove=new Hashtable<String,Integer>();
int mediaPeers=this.histogram.nPeers;
int counter=1;
String zonaSottocaricata=this.getClusterID();
int npeerSottocaricata=this.histogram.nPeers;
//ciclo l'istogramma per capire qual'è il cluster con meno peer
for(String key : this.histogram.PeersPerCluster.keySet()){
    int movingPeers=0;
    //se vi sono peer in fase di spostamento verso il cluster
    //li sommo a quelli che possiede attualmente
    if(msg.peersInMoving.containsKey(key))
        movingPeers=msg.peersInMoving.get(key);
    mediaPeers+=this.histogram.PeersPerCluster.get(key)+movingPeers;
    counter++;
    if(this.histogram.PeersPerCluster.get(key)<npeerSottocaricata){
        zonaSottocaricata=key;
        npeerSottocaricata=this.histogram.PeersPerCluster.get(key)+movingPeers;
    }

}

//calcolo la media dei peer per cluster
mediaPeers=(int) Math.round((float)mediaPeers/counter);
//se il mio cluster ha un numero di peer maggiore di un decimo della media
//significa che sono sovraccarico ed eseguo il bilanciamento
if(this.histogram.nPeers>(mediaPeers+((int)this.ClusterSize/10)) &
        this.histogram.nPeers-mediaPeers>((int)this.ClusterSize/10)){

    //numero di peer da disconnettere
    int peersToDisconnect=this.histogram.nPeers-mediaPeers;

    //memorizzo il numero di peer da disconnettere e la zona di destinazione (cluster con meno peer)
    peersToMove.put(zonaSottocaricata, peersToDisconnect);

    //invio al cluster sottocaricato un messaggio con numero di peer da spostare
    //per ricevere i nuovi id da assegnare ai peer del cluster attuale
    Node next=this.routingTable.getNeighbor(Direction.RIGHT, 0);
    AskIDToBalance askMsg=new AskIDToBalance(localnode,next);
    askMsg.targetID=zonaSottocaricata;
    askMsg.nPeerRequest=peersToDisconnect;
    askMsg.destination=SkipClusterUtil.getMembershipVectorFromID(zonaSottocaricata, pid);
    sendMessage(next,askMsg);
    return peersToMove;
}else{
    return null;
}
```

Figura 2.6: Funzione che bilancia un *cluster* sovraccarico di peer attraverso la richiesta di spostamento di peer su di un *cluster* sotto caricato

```
if (msg.isResponse){
    //Se è un messaggio di risposta ciclo gli id inviati dal super-peer del cluster destinazione
    for(int i =0;i<msg.avaiableIDs.size();i++){
        if(i>=this.histogram.usedID.size())
            break;
        //invio un messaggio con indicati i nuovi id e membershipvector per ogni peer da spostare
        AskRejoinMsg rejoinMsg=new AskRejoinMsg(localnode,localnode);
        rejoinMsg.destination=this.getMembershipVector();
        rejoinMsg.targetID=this.histogram.usedID.get(i);
        rejoinMsg.newID=msg.avaiableIDs.get(i);
        rejoinMsg.newVector=msg.newVector;
        sendMessage(rejoinMsg);
    }

}else{
    //calcolo gli id disponibili all'interno del cluster
    //da assegnare ai peers che dovranno spostarsi
    String minID=getClusterID();
    String maxID=getClusterID();

    for (int i=0;i<this.getIntraClusterBitLength();i++){
        minID+="0";
        maxID+="1";
    }

    if(intraClusterRight.size()>0){
        String rightNeighbor=getRightNeighbor();
        Node rightNode=intraClusterRight.get(rightNeighbor);
        intraClusterRight.clear();
        intraClusterRight.put(rightNeighbor, rightNode);
    }

    int maxBound=SkipClusterUtil.binaryStringToInt(maxID);

    //per ogni peer richiesto
    for (int i=0;i<msg.nPeerRequest;i++){
        //controllo di non superare la dimensione massima del cluster
        if(this.histogram.usedID.size()+2>this.ClusterSize){
            break;
        }
        //cerco un id disponibile (non presente attualmente) da assegnare ai nuovi peer
        int newID=SkipClusterUtil.binaryStringToInt(this._ID.toString())+1;
        boolean alreadyExist;
        do{
            alreadyExist=false;
            newID=newID+1;
            if(newID>maxBound)
                break;
            for(int j=0;j<this.histogram.usedID.size();j++){
                if(newID==SkipClusterUtil.binaryStringToInt(this.histogram.usedID.get(j))){
                    alreadyExist=true;
                }
            }
```

```
        //ricontrollo di avere id univoco
        temp=checkUniqueName(temp,pid);

        newID=SkipClusterUtil.binaryStringToInt(temp);

        //se l'id rientra nel cluster aggiungo al messaggio di risposta l'id da assegnare
        if(newID<maxBound){
            String newIDString=Integer.toBinaryString(newID);
            while(newIDString.length()<this._ID.toString().length())
                newIDString="0"+newIDString;
            msg.avaiableIDs.add(newIDString);
            this.histogram.usedID.add(newIDString);
        }
    }
    //invio il messaggio di risposta al mittente con i nuovi id
    msg.isResponse=true;
    msg.newVector=this.getMembershipVector();
    SkipCluster sender=(SkipCluster)msg.startNode.getProtocol(pid);
    msg.targetID=sender._ID.toString();
    msg.destination=sender._membershipVector;
    sendMessage(msg.startNode,msg);
}
```

Figura 2.7: Funzione in risposta alla richiesta di spostamento di *peer* sul *cluster* di cui il super-peer è responsabile. In questa funzione il super-peer invia al richiedente gli identificatori validi per inserire i *peer* nel proprio *cluster*

```
        //ricontrollo di avere id univoco
        temp=checkUniqueName(temp,pid);

        newID=SkipClusterUtil.binaryStringToInt(temp);

        //se l'id rientra nel cluster aggiungo al messaggio di risposta l'id da assegnare
        if(newID<maxBound){
            String newIDString=Integer.toBinaryString(newID);
            while(newIDString.length()<this._ID.toString().length())
                newIDString="0"+newIDString;
            msg.avaiableIDs.add(newIDString);
            this.histogram.usedID.add(newIDString);
        }
    }
    //invio il messaggio di risposta al mittente con i nuovi id
    msg.isResponse=true;
    msg.newVector=this.getMembershipVector();
    SkipCluster sender=(SkipCluster)msg.startNode.getProtocol(pid);
    msg.targetID=sender._ID.toString();
    msg.destination=sender._membershipVector;
    sendMessage(msg.startNode,msg);
}
```

Figura 2.8: In questa funzione il *peer* scelto per lo spostamento su di un altro *cluster* si disconnette dalla rete, cambia il proprio identificatore e si riconnette sul *cluster* di destinazione

Capitolo 3

GGrid

In questo capitolo presenteremo GGrid, fulcro di questo elaborato. In primo luogo, andreamo ad analizzare GGrid nella sua versione base. Andremo in seguito ad osservare come cambia il suo comportamento apportando modifiche in merito alla memorizzazione ed alla struttura, per poi arrivare ad una versione che presenta un traffico di rete ben bilanciato tra i nodi, grazie all'utilizzo di una tecnica nuova, basata sui collegamenti casuali.

3.1 GGrid in versione originale

La versione base di GGrid nasce da una collaborazione dell'Università di Bologna con l'Università di Chicago. GGrid viene definito dai creatori come una struttura dati multi-dimensionale distribuita su reti *peer-to-peer*, che organizza gli oggetti automaticamente nei diversi nodi sparsi nella rete.

3.1.1 La struttura

Lo spazio dati su cui opera GGrid possiamo immaginarlo come un iper-rettangolo con tante dimensioni quanti sono gli attributi dei dati che vi saranno inseriti. La struttura distingue due tipi di nodi appartenenti alla rete: i C-peer e gli S-peer. Questa distinzione è necessaria

in quanto, probabilmente, non tutti i nodi della rete detengono almeno un dato. Per definizione, i nodi che non sono responsabili di nessun dato vengono chiamati C-peer, mentre chi colleziona almeno un dato viene detto S-peer. Tale divisione permette di realizzare un indice distribuito più snello e semplice, dal momento che sarà necessario mantenere nell'indice solamente gli S-peer, interessati dalle *query* di ricerca, inserimento, eliminazione. Inizialmente, viene assegnato ad un nodo l'intero iper-rettangolo. Questo nodo sarà l'unico S-peer della rete. Mano a mano che verranno inseriti i dati, l'iper-rettangolo verrà diviso in modo da distribuire tra i diversi *peer* le zone di responsabilità. Ogni regione potrà mantenere, al proprio interno, un numero massimo di dati. Questo parametro viene definito *bucket-size*. Viene anche associato un numero minimo di elementi che una regione può contenere attraverso un secondo parametro, che, dopo svariati esperimenti, è stato fissato ad un terzo del valore del *bucket-size* per avere la massima efficienza dalla struttura.
I due parametri servono da guida per eseguire le operazioni di divisione e collassamento delle regioni. Come abbiamo detto, infatti, inizialmente vi è una singola regione, chiamata regione radice, o root, responsabile dell'iper-rettangolo che racchiude l'intera struttura. Mano a mano che vengono inseriti i dati vengono eseguiti i controlli per mantenere il vincolo del numero massimo di elementi ammissibili per ogni regione. Quando il numero di elementi di una certa regione supera il valore massimo consentito, si procederà ad una operazione di split, ossia di partizione. La partizione verrà effettuata su di una dimensione e la nuova regione sarà responsabile della metà dell'iper-rettangolo lungo la dimensione di split. Al contrario, se il numero di elementi di una regione scende al di sotto del numero minimo impostato come parametro del sistema, si procederà ad un collasso della regione stessa con la regione padre da cui è stata generata. La figura 3.1 mostra il caso di split con un iper-rettangolo di dimensione 2 nel momento in cui la regione root abbia un numero di elementi superiore a 3.

Grazie a questo tipo di divisione, GGrid acquisisce due proprietà molto importanti, che aiutano a semplificare la ricerca degli elementi sulla rete. La prima è la proprietà spaziale, la quale stabilisce che le regioni collocate allo stesso livello non si intersecano mai. La seconda è la proprietà di copertura in base alla quale ogni regione è inclusa

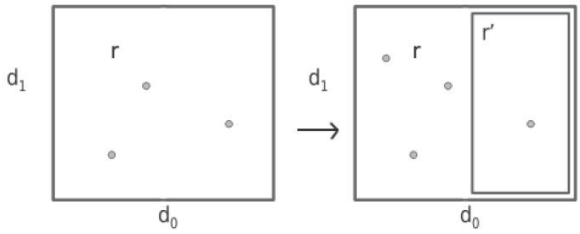


Figura 3.1: Split della regione principale con *bucket-size* pari a 3 dopo l'inserimento del quarto elemento

in una ed una sola regione padre, ad eccezione della regione radice (o root), che è inclusa in se stessa.
Per rappresentare le divisioni e la loro parentela padre-figlio è stata utilizzata una struttura ad albero.
Per poter procedere occorre assegnare un identificatore univoco alle regioni che si creano e, anche in questo caso, ci viene in aiuto la struttura ad albero scelta. La regione root verrà identificata con il carattere *. A questo punto, alla regione creata in seguito ad un'operazione di split, verrà assegnato, come codice univoco, l'identificatore della regione padre, seguito dal carattere 0 o 1 a seconda che la regione sia responsabile rispettivamente della zona sinistra o destra dell'iper-rettangolo. Ovviamente, potrebbero esservi zone che, a causa del basso numero di dati, non subiscono alcuna divisione. Di conseguenza, nell'albero delle regioni potrebbero esserci nodi intermedi ai quali non è associata alcuna regione dell'iper-rettangolo. Per memorizzare meglio gli identificatori si è scelto, inoltre, di utilizzare una notazione più semplice composta da due valori: il primo rappresenta il valore, in base 10, dell'identificatore assegnato alla regione (il carattere * viene codificato come 0); il secondo valore indica l'altezza dell'albero a cui si trova la regione stessa. Per comodità si utilizza la notazione π per indicare il valore in decimale dell'identificatore della regione ed il carattere l per identificarne l'altezza. Vediamo un esempio di quanto descritto in figura 3.2.

Ogni regione verrà associata ad un solo S-peer, che a sua volta potrà contenere altre regioni, mentre un C-peer non mantiene nessuna

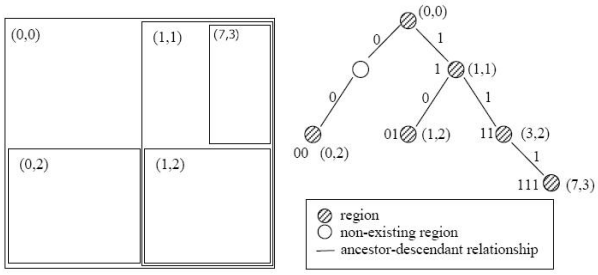


Figura 3.2: Esempio di una struttura GGrid con relativo albero delle regioni

regione. Il numero di regioni che un S-peer potrà contenere può essere impostato come parametro della struttura. Inoltre, i C-peer avranno solamente il collegamento con un altro nodo chiamato seme, o seed, in quanto non dovranno eseguire operazioni di *routing* dal momento che non appartengono all'albero, e quindi, ad un indice distribuito. I nodi di tipo S-peer, facendo parte dell'albero, dovranno mantenere i collegamenti con il padre ed i figli che ha generato per poter creare un indicizzazione distribuita della struttura.

Vediamo ora come verranno effettuate le operazioni di lookup, ossia di ricerca degli elementi nella struttura, lasciando al paragrafo successivo il compito di far luce sull'ingresso e rimozione di *peer* e dati dalla rete.
Per poter ricercare il dato nella struttura di GGrid è necessario ricavare qual'è la regione che dovrebbe memorizzare il dato e utilizzare l'albero delle regioni per raggiungere la regione di destinazione. Ricavare la regione obiettivo significa ricavare i valori di π ed l. Supponiamo, in un primo caso, di conoscere il valore dell'altezza dell'albero ed avere un iper-rettangolo unitario. Ogni dimensione del dato da inserire dovrà essere moltiplicato per 2^{li} dove li è la parte intera del quoziente tra la profondità massima dell'albero ed il numero di dimensioni. Unendo, a questo punto, il valore binario della parte intera della moltiplicazione appena effettuata di tutte le dimensioni, nell'ordine, otterremo il valore di π ricercato. Sarà sufficiente navigare l'albero per raggiungere la destinazione. Una delle due ipotesi sopra effettuate, però, difficilmen-

te viene soddisfatta. Si tratta della conoscenza globale dell'altezza dell'albero. Questo valore non è conosciuto da tutti i *peer*, che, al massimo conoscono la propria altezza e quella dei figli eventualmente generati. Utilizzando un valore di l piuttosto elevato, si otterrà una regione profonda che non esiste nell'albero, ma questo non è un problema. Infatti, se, navigando l'albero, si raggiunge un *peer* che non ha figli, questo sarà anche responsabile di tutte le ipotetiche regioni figlie che si potrebbero creare al di sotto del nodo stesso. Se si utilizzasse, invece, un valore di l minore dell'altezza attuale dell'albero, non si sarebbe più in grado di ottenere la destinazione specifica in quanto il *peer* raggiunto potrebbe avere delle ulteriori divisioni di regioni, e quindi di responsabilità. Per i nostri esperimenti abbiamo ipotizzato un valore di l pari a 32, il che significa ipotizzare di avere a che fare con un numero di regioni pari circa a 8 miliardi, numero sufficientemente grande utilizzabile anche in reti *peer*-to-peer a livello mondiale.
Per eseguire il *routing*, ogni *peer* ha la possibilità di inviare il messaggio al proprio padre o ai propri figli, in base alla regione ricercata, per raggiungere il destinatario del messaggio.
Anche per eseguire le operazioni di inserimento ed eliminazione dati sarà sufficiente trovare la regione destinataria, responsabile dell'identificatore dell'elemento in qeustione, che effettuerà l'operazione richiesta. La sola differenza è che queste operazioni potrebbero causare una divisione od un collassamento delle regioni, ma questo lo vedremo nel prossimo paragrafo.

3.1.2 *Join* e *leave* dei *peer*

In GGrid, il *join* dei *peer* non implica nessuna modifica alla struttura, in quanto, inizialmente, un nodo sarà identificato come un C-peer e non sarà responsabile di nessuna regione. In questo modo, un nodo che entra nella rete dovrà avere solamente un collegamento con un S-peer a cui inviare i messaggi, che, a sua volta, inoltrerà in base all'indice distribuito costituito dall'albero delle regioni. La modifica della struttura avviene quando ad essa viene aggiunto un S-peer. Quando una regione supera il numero massimo di elementi ed è costretta ad effettuare un'operazione di split, infatti, viene preso come candidato ad ospitare la nuova regione, un C-peer della rete. Questo, ricevendo una

regione in carico, diventerà automaticamente un S-peer ed entrerà a far parte dell'albero delle regioni. Il solo messaggio di rete da scambiare, in questo caso, è quello contenente gli elementi che dovranno essere spostati. Il nodo che riceverà queste informazioni otterrà anche l'identificatore della regione di cui diventerà responsabile ed il collegamento con il *peer* padre. Con questo semplice passaggio avviene il *join* corretto della struttura.
Per quanto riguarda l'operazione di disconnessione possiamo distingere due casi. Se il nodo che intende disconnettersi è un C-peer, non essendo parte dell'indice distribuito e non avendo elementi memorizzati nella propria locazione, non è necessaria alcuna operazione di *leave*. Se, invece, il nodo in questione è un S-peer, questo dovrà inviare alla regione padre i propri dati prima di lasciare la rete.
Anche l'operazione di eliminazione potrebbe essere considerata un operazione di *leave* in un caso particolare. Infatti, se, a seguito di una operazione di eliminazione, si ottiene una regione con un numero di elementi inferiore al minimo impostato, la regione collasserebbe, passando i dati alla regione padre. Se l'S-peer che ha eseguito il collasso della propria regione non ha altre regioni in carico, questo diventerebbe un C-peer e non farebbe più parte dell'albero delle regioni, eseguendo una sorta di *leave* dall'indice distribuito nonostante esso si trovi ancora sulla rete.

3.1.3 Punti critici

Il punto critico della versione base di GGrid viene identificato dall'analisi di distribuzione del carico. Infatti, essendo una struttura ad albero, i nodi più vicini alla radice saranno attraversati da un alto numero di messaggi. Per capire meglio entriamo nel dettaglio del percorso dei messaggi a partire dalle foglie, per raggiungere la radice. Le foglie saranno attraversate solo dai messaggi che avranno come destinatari le foglie stesse e non eseguiranno il *routing* di nessun messaggio tranne di quelli che loro stessi creano. I nodi appartenenti al livello superiore alle foglie, invece, oltre che raccogliere i messaggi di cui sono destinatari, eseguono il *routing* che permette ai messaggi di raggiungere i destinatari del nodo inferiore e permettono, ai nodi foglia, di inviare i messaggi in altre zone dell'albero. Mano a mano che si sale

sull'albero si vedrà che il traffico aumenterà in quanto, oltre a gestire i messaggi di cui un nodo è destinatario, dovrà eseguire il *routing* di tutti i messaggi inviati dai livelli inferiori. Quindi, maggiore è il numero di livelli sottostanti ad un nodo, maggiore è il *routing* che si troveranno a gestire. Fino ad arrivare al nodo radice, che smisterà i messaggi di tutti i nodi inferiori permettendo di raggiungere i lati oppposti dell'albero.

3.2 GGrid con learning

3.2.1 Differenze con la versione base

Un primo accorgimento per migliorare le prestazioni di GGrid, risiede nel proporre una soluzione che non preveda costi aggiuntivi di gestione. La nuova versione di GGrid è detta con 'Learning' in quanto alla rete viene data la capacità di 'imparare' la struttura della rete stessa. Per fare ciò abbiamo bisogno che ogni messaggio memorizzi al prorpio interno la lista dei nodi visitati. In questo modo, ogni *peer* che esegue il *routing* del messaggio, può creare collegamenti con tutti i nodi presenti nella lista contenuta in esso. In una rete stabile un elevato numero di *query* porterebbe, a tutti i *peer*, una conoscenza dell'intero sistema. In questo modo ogni *peer* potrebbe ricostruirsi un proprio albero di GGrid internamente ed avere i collegamenti con tutti i *peer* della rete. La struttura ad albero, in cui i figli di un nodo sono costituiti dall'identificativo del nodo padre seguito da un solo carattere, '1' o '0', permetterebbe una memorizzazione di molti nodi richiedendo poco spazio. Infatti, ogni nodo può essere memorizzato con la coppia indirizzo-carattere aggiuntivo rispetto al proprio padre. Con la conoscenza dell'intera rete, ogni nodo, potrà inviare i messaggi al diretto destinatario senza passare per *peer* intermedi. Ovviamente, questo accade se si ha un elevato numero di *query* uniformemente distribuito. Se un nodo viene raggiunto da poche *query* è probabile che il suo indirizzo non sia conosciuto da tutti i *peer* della rete, ma solo da una buona parte. Quindi, il numero di *hop*, che idealmente può essere pensato pari ad 1 (il destinatario viene contattato in maniera diretta), sarà leggermente maggiore a causa della non perfetta uniformità delle *query* generate nel sistema.

3.2.2 Punti critici

L'aumento del numero di *hop* non è un vero e proprio problema. Infatti, l'aumento da 1 a circa 2 non comporta grandi ritardi nel raggiungimento del destinatario del messaggio.
Il problema da sottolineare riguarda, ancora una volta, la distribuzione del traffico. Se si lavorasse nella situazione ideale, nella quale tutti i *peer* hanno la conoscenza completa della rete, ovviamente, anche la distribuzione del traffico sarebbe ideale. Infatti, utilizzando sempre i collegamenti diretti, i messaggi creerebbero traffico solamente sul nodo di destinazione, quindi, con una distribuzione uniforme di *query* si avrebbe una distribuzione uniforme del traffico.
Nel momento in cui, un determinato *peer*, non avesse la conoscenza completa della rete, utilizzerebbe dei nodi intermedi per consegnare i messaggi.
Il problema si accentua durante l'ingresso di nuovi S-peer nella rete. Inizialmente, un nodo che ha appena effettuato il *join*, mantiene il collegamento solamente con un nodo chiamato seme. Quindi, solamente alla ricezione di messaggi di cui è esso stesso destinatario, il nodo potrebbe accrescere la propria conoscenza. Fino a che non raggiunge una conoscenza quasi completa della rete, il nodo invierà i messaggi ai soli *peer* intermedi che conosce. Come abbiamo detto in precedenza, inizialmente, il nodo conosce solo il *peer* seme, quindi caricherà quest'ultimo con ogni sua richiesta. Questo porterebbe ad una distribuzione del carico non uniforme.
In ambiti in cui la rete ha alta dinamicità, e, precisamente, quando vi è una frequenza di ingresso di nodi nella rete piuttosto elevata rispetto al numero di *query*, la distribuzione non si ripartisce più in modo uniforme.

3.3 Verso una migliore distribuzione del traffico

Il miglioramento apportato dalla versione con Learning di GGrid, diminuisce drasticamente il numero di *hop* medi del sistema, portando però ad una distribuzione del carico di lavoro non uniforme sui *peer*

della rete. Dal momento che la distribuzione uniforme è un punto chiave di questi sistemi proviamo a fare un passo indietro, partendo dalla versione base di GGrid, per sviluppare dei miglioramenti incentrati sulla distribuzione del carico.
Per fare questo abbiamo provato ad applicare due tipologie di strati di *overlay* alla struttura base di GGrid. Vedremo nei prossimi paragrafi gli *overlay* applicati, con i rispettivi pregi e difetti.

3.3.1 Aggiunta di un *overlay* alla CHORD

Un *overlay* molto utilizzato in questo momento per le reti *peer*-to-peer è CHORD[8]. La scelta di questo *overlay* da applicare a GGrid è stata guidata dai risultati di buona distribuzione del carico che arrivano dagli esperimenti effettuati in letteratura.
Questo *overlay* permette di disporre i nodi su di un anello, ordinati in modo crescente in base al proprio identificativo. Ogni nodo ha poi un collegamento con i *peer* che si trovano a distanza esponenziale da esso. Quindi, un nodo avrà come vicini il successore, il secondo nodo più vicino a lui, il quarto, l'ottavo, e così via, come si vede nell'esempio mostrato in figura 3.3. Nella figura si notano i collegamenti del nodo con identificatore pari a N8.

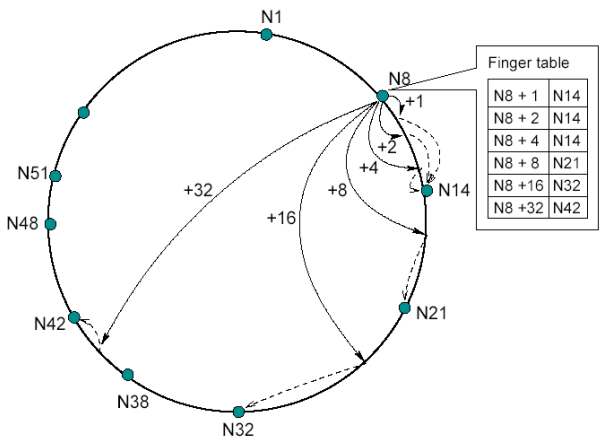


Figura 3.3: Esempio di una rete con *overlay* CHORD

In CHORD, l'algoritmo di *routing* permette di raggiungere velocemente la zona limitrofa al destinatario del messaggio. I messaggi, infatti, vengono inviati al *peer* più lontano il cui identificatore non superi l'obiettivo da raggiungere. Vista la notorietà dell'*overlay* CHORD in letteratura, non entreremo nel merito dei dettagli implementativi dell'architettura, rimandando il lettore alle note bibliografiche.

Come cambia la struttura

Una volta definiti i principi di CHORD, vediamo come è stato possibile applicarlo alla struttura GGrid, mantenendo gli stessi principi costruttivi.
Innanzitutto, CHORD opera su di un anello ordinato di *peer*, mentre GGrid ha una struttura ad albero. Per poter trasformare l'albero in anello occorre definire un ordinamento dei nodi presenti nella struttura. Ovviamente i *peer* che faranno parte dell'anello saranno i soli S-peer, esattamente come accade per la costruzione dell'albero in GGrid versione base. L'ordinamento dei *peer* è stato scelto utilizzando un criterio grafico: vengono, nell'ordine, prima i figli di sinistra, poi quelli di destra, e successivamente, il padre. Ovviamente il criterio è ricorsivo durante la discesa dell'albero, dalla radice alle foglie. In figura 3.4 vediamo un esempio di ordinamento di un albero di GGrid.

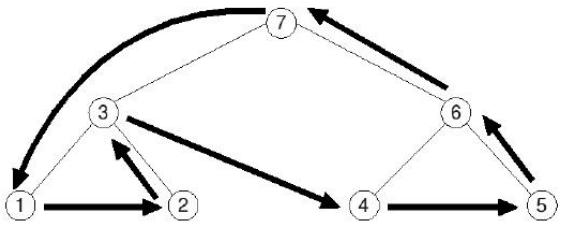
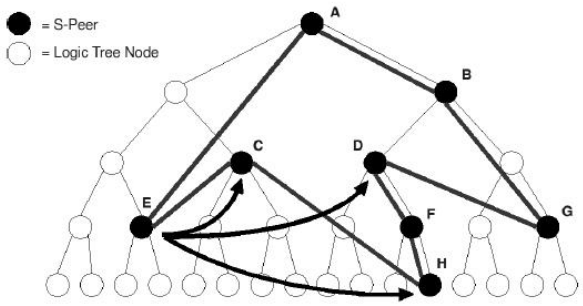


Figura 3.4: Ordinamento dell'albero delle regioni di GGrid

Una volta definito l'ordinamento, è necessario collegare i nodi nel modo opportuno per avere l'anello, parte base della struttura di CHORD. A questo punto sarà sufficiente creare i collegamenti esponenziali come sono definiti dell'architettura CHORD. In figura 3.5 è mostrato un

esempio di come si presenterà la struttura finale della nuova versione di GGrid.



Node	$\Pi(R_p)$	Log-Link	Node	$\Pi(R_p)$	Log-Link
A	*	E-C-F	**E**	*001	C-H-D
B	*1	A-E-H	**F**	*101	D-G-A
C	*01	H-F-G	**G**	*111	B-A-C
D	*10	G-B-E	**H**	*1011	F-D-B

Figura 3.5: Sovrapposizione della struttura CHORD all'*overlay* GGrid

Punti critici: *join* e *leave* dei *peer*

Nonostante, grazie alla struttura CHORD, si sia riusciti ad ottenere un buon risultato per quanto riguarda la distribuzione del carico, emerge un punto critico degno di noto: il mantenimento della struttura. Infatti, ogni volta che un nodo esegue un *join* nell'*overlay* CHORD, questo deve innanzitutto trovare il posto giusto sull'anello, successivamente deve costruirsi i *link* di distanza esponenziale. Il problema della ricerca dei *link* del *peer* appena entrato non desta preoccupazione, infatti, un *peer*, invece che cercare manualmente tutti i *link*, potrebbe sfruttare un meccanismo intrinseco in questa tipologia di collegamenti. Il *link* n-esimo del *peer*, infatti, è il collegamento a distanza n-1 del *peer* precedentemente trovato. In questo modo, un *peer*, una volta trovato il successore, e quindi il primo nodo della lista di collegamenti esponenziali, è sufficiente che, per cercare il suo secondo collegamento, chieda il primo successore al suo primo collegamento. Chiederà, a seguire, il secondo collegamento al secondo *peer* per ottenere il proprio terzo collegamento, e così via. In questo modo il costo di creazione della tabella

di *routing* di un *peer* che si collega all'anello ha una complessità del tipo O(log n), dove n è il numero di nodi presenti nell'anello. L'ingresso di un *peer* nella rete, però, non si limita solamente alla creazione dei *link* del *peer* appena entrato. Infatti, i *link* esponenziali della maggior parte dei nodi potrebbero cambiare in seguito all'ingresso di un *peer* che non era stato calcolato in precedenza. Questo fa sì che tutti i *peer* dell'anello controllino i propri collegamenti dopo il verificarsi di un *join* nella rete. Questo controllo va eseguito anche al momento della disconnessione di un *peer*.
Queste operazioni creano un traffico, causato dal mentenimento della struttura, molto elevato. Questo punto critico ha portato alla scelta di un secondo *overlay* da sovrapporre a GGrid, per cercare di ottenere circa gli stessi risultati, in termini di distribuzione del traffico, del sistema appena descritto, ma con un costo di mantenimento della rete molto minore a fronte delle dinamicità della rete.

3.3.2 GGrid diventa un piccolo mondo

Come vedremo nell'appendice, negli anni Novanta, è stata ripresa una teoria sperimentata da Milgram nel 1967. L'esperimento mette in luce le reti piccolo mondo e le sue caratteristiche riguardo al basso numero di *hop* necessari per collegare due elementi lontani in una rete con un elevato numero di oggetti. La teoria viene ripresa dagli studiosi Watts e Strogatz che sottolineano l'importanza delle reti piccolo mondo, osservando che sono molto frequenti in natura. Le caratteristiche osservate da questi studi sono principalmente il basso numero di *hop* e la naturale distribuzione del traffico piuttosto equa. Il modello elaborato per le reti piccolo mondo si adatta perfettamente alle reti *peer*-to-peer, e, viste le caratteristiche presentate in letteratura, è stato scelto come *overlay* da sviluppare sopra l'architettura GGrid.

La nuova struttura

La caratteristica fondamentale delle reti piccolo mondo sancisce che ogni nodo della rete dovrà essere collegato con un insieme di vicini ed un numero di collegamenti casuali con alcuni nodi sparsi sulla rete. Nella nostra implementazione abbiamo deciso di stabilire i vicini

del nodo come il predecessore ed il successore dell'anello creato come descritto per l'architettura GGrid con l'ausilio di CHORD. L'anello, infatti, ha come caratteristica essenziale, la certezza di risultato, in quanto, vi è sempre almeno un collegamento a cui inviare un messaggio per avvicinarlo alla destinazione. Una volta stabiliti i vicini è stato sufficiente creare alcuni collegamenti lontani per ricreare l'*overlay* della rete piccolo mondo. Abbiamo utilizzato un numero di collegamenti lontani pari al valore logaritmico del numero di *peer* collegati alla rete. Per ottenere ogni collegamento sono state eseguite *query* di ricerca casuali all'interno della rete che, una volta raggiunto il nodo destinatario, lo collegavano con il nodo mittente. Caratteristica importante è che i *link* lontani siano distribuiti equamente nella rete. In una rete stabile questa caratteristica viene rispettata dal momento che si creano *query* casuali e, quindi, con distribuzione uniforme. In figura 3.5 possiamo notare un esempio della rete piccolo mondo (o small-world) che applicheremo all'architettura GGrid.

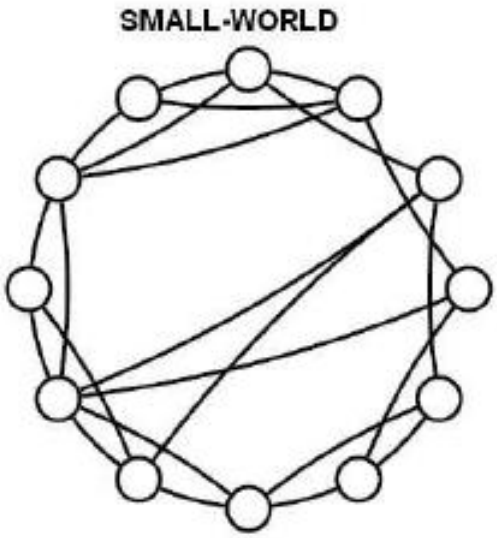


Figura 3.6: Rete piccolo mondo che verrà sovrapposta all'architettura GGrid

Join e *leave* dei *peer*

Un *peer* che si collega nella rete piccolo mondo ha come scopo iniziale quello di collocarsi all'interno di un anello di S-peer, ordinato come descritto precedentemente per la struttura CHORD. Una volta inseriti nell'anello, i *peer* in fase di *join* dovranno eseguire le *query* casuali per ottenere i collegamenti con i destinatari delle stesse.
Notiamo, però, che in caso di alta dinamicità della rete, il vincolo che prevede la presenza di *link* lontani distribuiti uniformemente nel piccolo mondo, potrebbe essere violato. Infatti, un *peer* che si connette in una rete stabile, non ha la possibilità di essere contattato come *link* lontano dei *peer* già inseriti nel sistema. Per ovviare a questo problema è stato deciso che i messaggi relativi alle *query* utilizzate per la creazione dei *link* lontani, create all'ingresso di un *peer* sulla rete, creassero un collegamento tra il *peer* destinatario e il nodo che ha emesso il messaggio. In questo modo, si viene a creare un collegamento supplementare che non comporta costi aggiuntivi ma che aiuta il bilanciamento e la distribuzione dei *link* casuali, detti anche random, sulla rete.
Il grande vantaggio di questa topologia di rete, rispetto all'*overlay* CHORD sovrapposto in precedenza, sta nel fatto che l'ingresso di un *peer* non invalida i *link* di nessun *peer* nella rete. Infatti, essendo *link* casuali, non hanno una struttura logica, quindi non devono essere ricalcolati. Il vantaggio che deriva da questa caratteristica si traduce in un risparmio di molti dei messaggi necessari alla ricostruzione dei *link* di tutto l'anello. Anche un'operazione di disconnessione non creerà traffico aggiuntivo. Se, infatti, vi è un *peer* collegato in maniera casuale con un *peer* che si è disconnesso, appena il nodo notificherà l'assenza del *peer*, ricreerà un *link* casuale in sostituzione di quello perduto. Il numero di messaggi è ancora esiguo.
Il grande vantaggio di questo *overlay* sta nel fatto, dunque, che la connessione o la disconnessione di nodi dalla rete non comportano aggiornamenti in cascata, diversamente da quanto accadeva nella struttura CHORD descritta precedentemente.
Questi vantaggi saranno visibili nel prossimo capitolo, che mostrerà i risultati degli esperimenti mettendo a confronto, oltre a SkipCluster, tutte le versioni di GGrid elencate in questo capitolo.

Capitolo 4

Esperimenti e risultati

4.1 PeerSim: il simulatore di reti *peer-to-peer*

I nostri esperimenti vedono protagoniste le reti *peer-to-peer*. Per eseguire gli esperimenti, quindi, è necessario ricreare, su di una macchina, un'ambiente capace di simulare una rete *peer-to-peer*. Abbiamo analizzato i simulatori software di reti *peer-to-peer* presenti attualmente nel panorama accademico. Le possibili scelte sono molteplici. Ogni simulatore mostra aspetti più o meno diversi della rete e hanno differenti gradi di personalizzazione. L'idea alla base di questi programmi è quella di offrire un vero e proprio *framework* che permetta agli sviluppatori di implementare un proprio protocollo di scambio dei dati e di personalizzare le caratteristiche dei nodi della rete. Tra i simulatori analizzati i due principali simulatori analizzati sono stati PeerSim e PlanetSim. Entrambi scritti in Java, mostrano un *framework* molto diverso. PlanetSim offre tre livelli implementativi che permettono di suddividere l'*overlay* da implementare in tre diversi sotto-problemi legati rispettivamente ai livelli di applicazione, di nodo e di rete. La struttura permette di organizzare bene il lavoro costringendo, però, lo sviluppatore a seguire con rigore le API messe a disposizione e, quindi, rendendo necessario uno studio approfondito della complessa struttura su cui è costruito PlanetSim. Per i nostri esperimenti abbiamo deciso di utilizzare il simulatore PeerSim. La scelta è stata, ancora una volta,

dettata dal "luogo di nascita" del software. Infatti, mentre PlanetSim non è un prodotto italiano, PeerSim è sviluppato, come GGrid, da un docente dell'Università di Bologna in collaborazione con docenti di altre nazionalità. Inoltre, dal punto di vista tecnico, PeerSim offre la possibilità di creare *overlay* molto più liberamente e di applicarli, in maniera ordinata, ai nodi che compongono la rete.
Scendendo nello specifico, il componente fondamentale di PeerSim è il nodo. Definendo, infatti, una classe che implementa l'interfaccia del nodo fornita da PeerSim, è possibile gestire eventi quali l'arrivo di messaggi o eventi interni definiti dall'utente. Inoltre è possibile associare ai nodi della rete più *overlay* contemporaneamente. Nei nostri esperimenti abbiamo collegato ad ogni nodo l'*overlay* specifico dell'ambiente da testare, come GGrid o SkipCluster, ed un *overlay* di trasporto fornito dal simulatore in modo nativo. Abbiamo inoltre creato dei controlli, che il sistema esegue periodicamente, secondo un parametro impostato su di un file di configurazione, che permettono la generazione di *query*, la scrittura dell'immagine della rete su database, e l'eventuale connessione o disconnessione di nodi. Ogni parametro, dei nodi e dei controlli, viene letto da un file di testo impostato come file di configurazione. Per ogni altro dettaglio tecnico si consiglia di visitare il sito *http://peersim.sourceforge.net/* che permette una consultazione dettagliata del simulatore.

4.2 Scenari e misure osservati

Vediamo in questa sezione quali sono le variabili di interesse che evidenzieremo dai risultati ottenuti ed osserviamo in che ambiti abbiamo eseguito i nostri test.

4.2.1 Le misure osservate

Il parametro fondamentale su cui ci soffermeremo maggiormente è la distribuzione del traffico. Come abbiamo descritto nei primi capitoli, questo parametro è di grande importanza in quanto indica quanto uniformemente è distribuito il traffico di lavoro sui *peer* che compongono la rete. Osserveremo inoltre altri importanti parametri quali la media del:

- numero di *hop*;
- numero di messaggi medi scambiati per mantenere la struttura (chiamati messaggi di sistema);
- numero medio di collegamenti che danno anche indicazione della memoria occupata dalle strutturte dati dell'architettura.

4.2.2 Gli scenari sperimentati

Abbiamo testato gli *overlay* sviluppati in diversi scenari applicativi. In tutti gli scenari partiamo da una rete costituita da un solo elemento e colleghiamo, molto rapidamente, 10000 nodi. Questi nodi si collegano con tempi casuali alla rete entro un tempo limite impostato da file di configurazione. I nodi creano la struttura desiderata scambiando messaggi con gli altri *peer* presenti nella rete. Alla fine della connessione iniziamo ad eseguire periodicamente 100000 *query*. Le osservazioni, e quindi le scritture su database, sono eseguite ogni 10000 *query*. Se questi sono i parametri comuni a tutti gli scenari, vediamo ora le particolarità che ogni ambiente testato presenta.
Il primo scenario presenta solamente l'esecuzione delle *query*, senza apportare modifiche alla struttura creata dopo l'inserimento dell'ultimo nodo entrato in rete. Al termine dell'esecuzione i nodi saranno ancora 10000.
Nel secono scenario supponiamo che, dopo la costruzione della rete iniziale, ci siano inserimenti di *peer* eseguiti costantemente ogni 100 *query*. Questo aumenterà la rete di circa 1000 nodi.
Nel terzo scenario supponiamo, una volta completata la costruzione della rete iniziale, di eseguire connessioni o disconnessioni ogni 100 *query* con la probabilità del 50%, ossia ogni 100 *query* eseguiremo in maniera casuale una connessione oppure una disconnessione. Il numero di nodi varierà nel tempo rimanendo circa intorno ai 10000 nodi iniziali.
Nell'ultimo scenario utilizzeremo un teorema ideato dal professore dell'Università di Bologna, Gianluca Moro, che vincola ad eseguire un numero di connessioni pari alla radice quadrata del numero di *query* eseguite fino ad un certo istante.

In tutte queste simulazioni supponiamo un carico distribuito uniformemente sui *peer* che compongono la rete.
Per la memorizzazione dei risultati abbiamo utilizzato il motore di database PostgreSQL, salvando i dati su tre tabelle mostrate nella figura 4.1.

4.3 I risultati

In questa sezione andremo a suddividere i risultati in base allo scenario sperimentato e commenteremo i grafici relativi alle misure registrate.

```
-- Tabella per le statistiche dei nodi di GGrid

CREATE TABLE ggridpeers (
    sim text NOT NULL,
    at_msg bigint NOT NULL,
    ip text NOT NULL,
    connected boolean NOT NULL,
    s_peer boolean NOT NULL,
    treedepthfound integer NOT NULL,
    requestsent integer NOT NULL,
    responsereceived integer NOT NULL,
    requesttimeout integer NOT NULL,
    requesterror integer NOT NULL,
    averagehops double precision NOT NULL,
    dispatchedexactdatamsg integer NOT NULL,
    ownerpi text,
    log_link integer,
    c_peer_link integer,
    s_peer_link integer,
    random_peer_link integer,
    conta_traffico integer,
    conta_sistema_loglink integer,
    conta_sistema_random integer,
    conta_link_strutturali_usati integer,
    conta_link_log_random_usati integer,
    conta_sistema_struct integer,
    nregion integer DEFAULT 0,
    nfullregion integer DEFAULT 0
);
```

```
--Tabella per la quantità delle regioni dei nodi GGrid

CREATE TABLE regions (
    sim text NOT NULL,
    at_msg bigint NOT NULL,
    ip text NOT NULL,
    pi text,
    nregions integer DEFAULT 0
);

-- Tabella per le statistiche dei nodi di SkipCluster

CREATE TABLE skipclusterpeers (
    sim text NOT NULL,
    at_msg bigint NOT NULL,
    id text NOT NULL,
    clusterid text NOT NULL,
    membershipvector text NOT NULL,
    connected boolean NOT NULL,
    responsereceived integer NOT NULL,
    averagehops double precision NOT NULL,
    intracluster_link integer,
    interclusterlink integer,
    conta_traffico integer,
    conta_sistema integer,
    higloblink integer DEFAULT 0,
    npeerincluster integer DEFAULT 0
);
```

Figura 4.1: Tabelle utilizzate per la memorizzazione dei risultati degli esperimenti

4.3.1 Rete stabile

Come anticipato, in questo scenario abbiamo eseguito inizialmente la connessione di 10000 *peer* alla rete. Dopo un breve periodo di assestamento della prima fase, eseguiamo 100000 *query* selezionando casualmente il *peer* che effettua la *query* ed il dato ricercato nella rete. Ogni 10000 *query* eseguiremo una fotografia del sistema e tracceremo i grafici sotto riportati.
In questo scenario, i nodi rimangono costanti a 10000.

Dal grafico in figura 4.2 possiamo osservare che la versione base di SkipCluster necessita di pochi messaggi per creare la struttura,

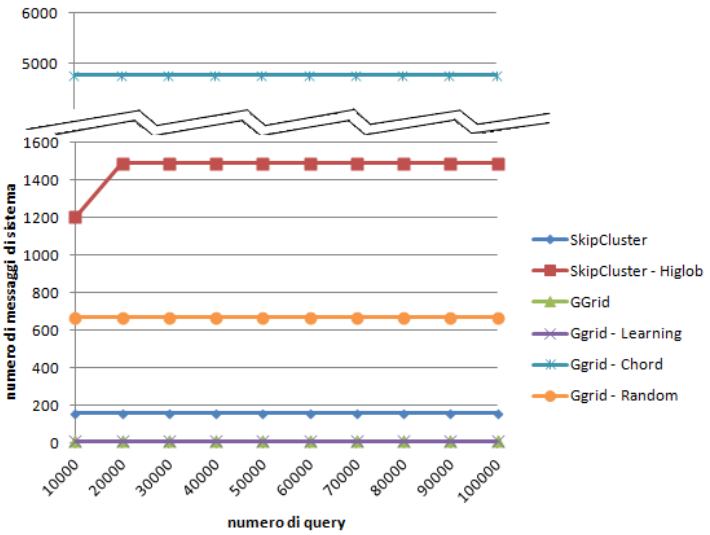


Figura 4.2: Traffico di sistema

e che, ovviamente, l'introduzione di HiGLoB incrementa in maniera piuttosto evidente tale valore. Per quanto riguarda GGrid, osserviamo che la versione base ha un costo quasi nullo grazie alla struttura ad albero binario di cui è composto. L'unico costo di mantenimento viene riscontrato nella fase di delegazione delle regioni che portano, necessariamente, alla creazione di nuovi nodi nell'albero degli S-peer. Il costo di GGrid con Learning ha lo stesso valore della versione base in quanto il Learning viene eseguito in *piggyback* nei messaggi di *query*. Notiamo, infine, che, come annunciato in precedenza, la struttura Chord presenta un lato negativo in termini di numero di messaggi di aggiornamento, risultanti troppo elevati. Questo ricade sul costo di mantenimento della struttura. Infine, osserviamo come la versione Random di GGrid necessita di un costo non eccessivamente elevato rispetto agli altri *overlay* testati.

Il grafico di figura 4.3, mostra, invece, il numero medio di *hops* per *peer*. Ogni *peer* misura la media degli *hops* necessari al raggiungimento del dato ricercato. Inizialmente il numero di messaggi di *query*

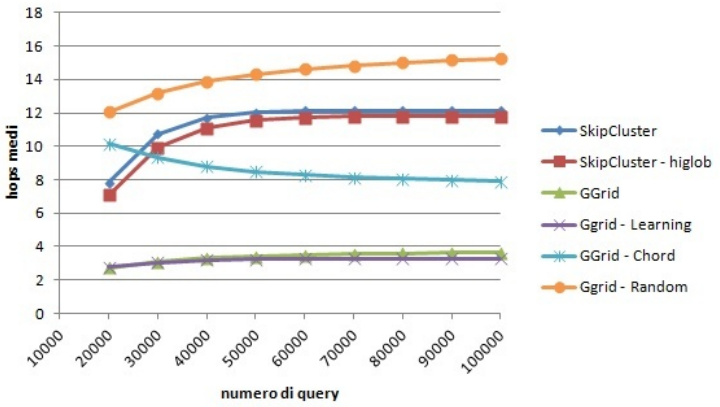


Figura 4.3: Numero medio di *hops* per *peer*

inviati da ciascun *peer* è molto basso, quindi il valore misurato sarà interessante solo dopo una fase iniziale di assestamento. Ovviamente, più *query* si eseguono, più ci si avvicina alla media prevista. Si nota, infatti, che la versione di GGrid con Learning, tende al valore teorico 2. Questo valore è giustificato dal fatto che, dal momento che la rete non cambia la propria composizione, con un numero elevato di *query*, ogni *peer* avrà conoscenza dell'intera struttura, impiegando quindi solo un passaggio per raggiungere la soluzione ed uno per il ritorno. Anche GGrid in versione base mostra un *hop* medio piuttosto basso. Entrambe le versioni di SkipCluster presentano un *hop* medio di poco inferiore al valore logaritmico della dimensione della rete. La differenza tra le due versioni di GGrid, alla Chord e con *link* Random, porta, in questa osservazione, a notare un vantaggio per la prima versione, anche se la differenza tra i due non è così eccessiva da penalizzare pesantemente nessun *overlay*. Infine, da questo grafico, notiamo che tutte le implementazioni i GGrid, al di fuori di quella con *link* casuali, presentano un *hop* medio minore di quello offerto da SkipCluster

Dal grafico in figura 4.4 otteniamo informazioni sulla quantità di memoria necessaria al mantenimento di ciascuna struttura. Infatti, un numero elevato di *link* comporta necssariamente una elevata quantità di memoria per poterli memorizzare. Notiamo che GGrid in versione

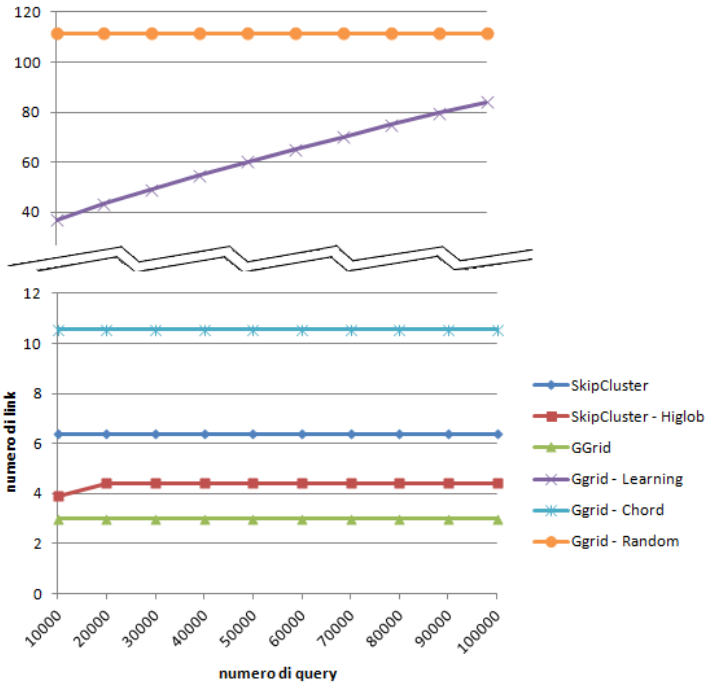


Figura 4.4: Numero di *link* necessari al mantenimento della struttura

Random necessita di un numero di *link* abbastanza elevato. Il numero di *link* non è logaritmico rispetto alla dimensione della rete, in quanto abbiamo dovuto memorizzare i *link* con gli ultimi nodi entrati sulla rete per distribuire in maniera più uniforme i *link* generati in maniera casuale. Il valore ottenuto è comunque un valore fisso. Il dato peggiore relativamente al numero di *link* è rappresentato da GGrid in versione Learning. Questo cresce con l'aumentare delle *query*. Infatti, ad ogni *query*, ogni *link* conosce e crea *link* con gli altri nodi attraverso cui la *query* è passata. Questo porta, a regime, ad avere un numero di *link* pari alla dimensione della rete. Nel grafico si nota l'andamento crescente dei *link* che, come detto in precedenza, arriverà a 10000 dopo un numero elevato di *query*. Si nota, inoltre, che il numero di *link* memorizzati per mantenere la struttura Chord è, come ci si aspettava per costruzione, all'incirca pari al logaritmo del numero di

elementi della rete. GGrid in versione base ha il minor numero di *link* per nodo essendo costituito da una struttura ad albero binario. Per quello che riguarda SkipCluster notiamo che l'introduzione di Higlob diminuisce di un terzo il numero di *link* di struttura, portando ad un valore vicino a quello osservato per GGrid in versione base.

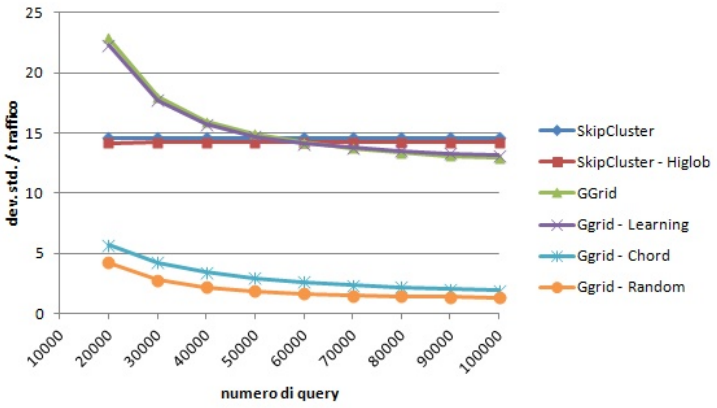


Figura 4.5: Distribuzione del traffico

Nell'ultimo grafico, ossia quello in figura 4.5, abbiamo misurato la distribuzione del traffico media per *peer* come il rapporto tra la deviazione standard e la media del numero di messaggi smistati da ciascun nodo della rete. Notiamo che le due versioni che si distinguono in maniera accentuata dalle altre sono GGrid nelle versioni Chord e Random. Il valore, in questo caso, si aggira attorno all'1, e questo indica una distribuzione del traffico quasi uniforme.

4.3.2 Rete con soli ingressi

Anche questo scenario è composto da due fasi. Nella prima fase eseguiamo il collegamento dei 10000 nodi che compongono la rete, attendendo un breve periodo che permette di stabilizzare la rete. Nella seconda fase, diversamente dal primo scenario, eseguiamo l'introduzione di un *peer* nella rete ogni 100 *query*. Questo causa l'aumento di nodi in maniera proporzionale al numero di *query* effettuate. L'andamento della crescita è visibile dal grafico mostrato in figura 4.6

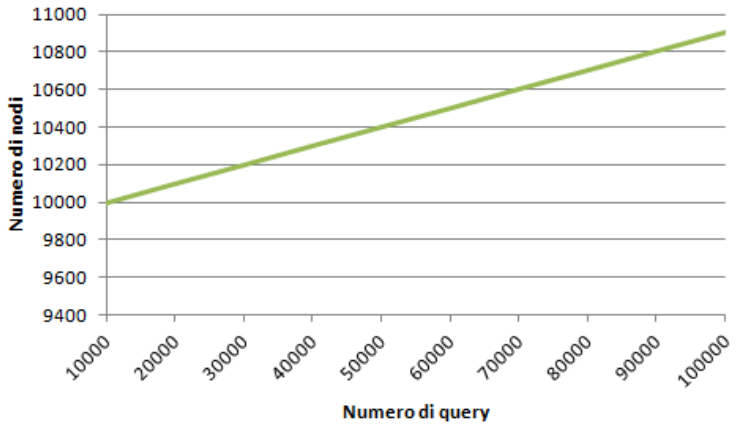


Figura 4.6: Numero di nodi della rete all'aumentare delle *query*

Esattamente come abbiamo effettuato nell'esperimento precedente analizziamo in primo luogo il numero di messaggi scambiati tra i *peer* che compongono la rete per mantenere la struttura dell'*overlay* associato. In figura 4.7 possiamo notare che, come in precedenza, l'*overlay* che necessita un numero di messaggi di gran lunga più elevato è ancora GGrid in versione Chord. In questo scenario notiamo che la media di messaggi di sistema cambiati, diminuisce leggermante nel tempo nonostante il numero di *peer* aumenti. Questo è giustificato dal fatto che, una volta che la rete si è stabilizzata, il numero di messaggi di sistema causati dall'ingresso sporadico di nuovi *peer* è minore dei messaggi scambiati nella fase iniziale, in cui la rete riceve un elevato incremento di dimensione in un tempo breve. La diminuzione di questo parametro, in questo scenario, indica, quindi, che il numero medio di messaggi di sistema è leggermente più basso rispetto a quanto misurato nella fase iniziale, ma rimane comunque un valore molto elevato rispetto alle altre soluzioni sperimentate. Anche in questo scenario troviamo SkipCluster con HiGLoB come successore a GGrid in versione Chord, che necessita quindi di un numero elevato di messaggi di sistema rispetto a SkipCluster in versione base in quanto deve mantenere aggiornato l'istogramma fondazionale di HiGLoB. Notiamo, poi, che, come nello scenario precedente, GGrid in

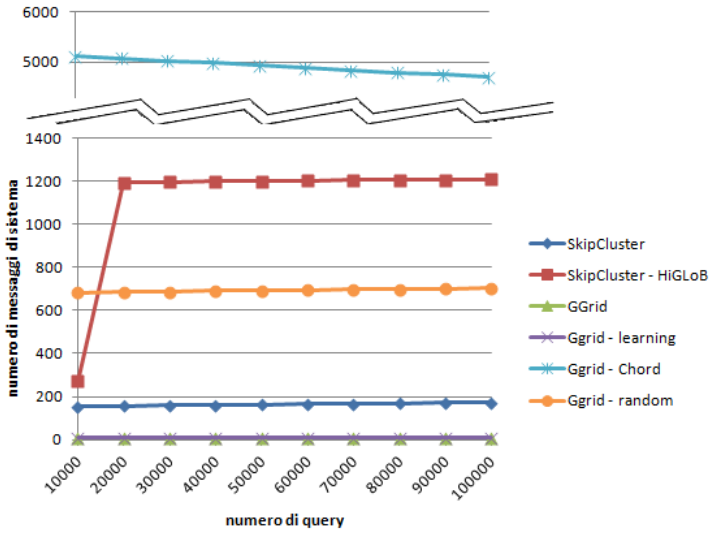


Figura 4.7: Traffico di sistema

versione Random mantiene un numero di messaggi non troppo elevato rispetto alle altre soluzioni, che aumenta di poco con il crescere della dimensione della rete. Anche SkipCluster presenta circa gli stessi valori osservati nel precedente esperimento, anche se ora si nota un aumento abbastanza evidente del numero di messaggi al crescere della rete. Infine, il numero di messaggio scambiati da GGrid in versione base ed in versione Learning rimangono ancora una volta molto bassi e non crescono significativamente con l'introduzione di nuovi *peer* sul sistema.

Dal grafico in figura 4.8 possiamo notare l'andamento del numero di *hop* medio osservato da ogni *peer* al crescere della rete. Notiamo che i valori, in generale, aumentano con l'aumentare della dimensione della rete. In particolare GGrid in versione Random è quello che mostra un numero di *hop* maggiore, attorno a 16, in leggero aumento con l'ingresso di *peer* sulla rete. Inoltre, rispetto allo scenario precedente, notiamo che la differenza tra SkipCluster e SkipCluster con HiGLoB è più accentuata. Una rete con un numero medio di *peer* per *cluster* equilibrato comporta, quindi, una reazione migliore, in termini di *hops*,

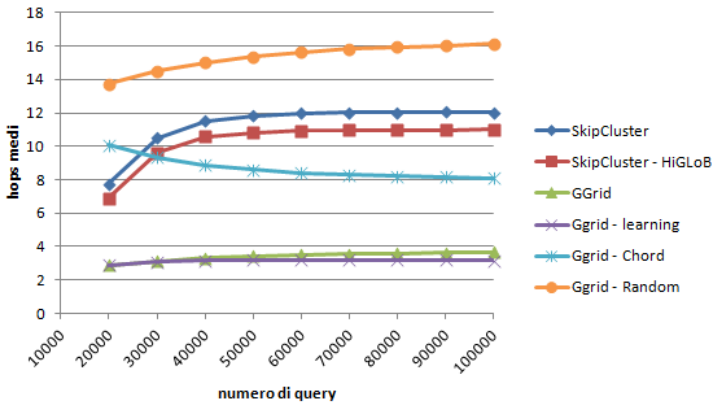


Figura 4.8: Numero medio di *hop*s per *peer*

all'aumento del numero di nodi sul sistema rispetto ad uno scenario con distribuzione non uniforme dei *peer* sui *cluster*. L'*overlay* GGrid in versione Chord presenta pressoché le stesse caratteristiche dello scenario precedente. Infine notiamo che ancora una volta, GGrid in versione base ha un *hop* molto basso, che si assesta intorno a 4, mentre l'introduzione del Learning tende ad un valore ideale pari a 1. In questo scenario, in cui la rete cresce continuamente, non sarà però possibile raggiungere il valore ideale neanche a regime in quanto ogni volta che si collega un nuovo *peer* questo non è presente, inizialmente, sulla struttura di quello già presenti.

Nel grafico in figura 4.9 abbiamo rappresentato il numero di *link* medio per *peer* con altri nodi della rete. Questi *link* sono necessari al mantenimento della struttura e forniscono, quindi, una misura anche sulla quantità di memoria riservata per la struttura. Il numero di *link* di GGrid in versione Random cresce leggermente con l'ingresso di nuovi *peer*. Questo perché, come spiegato nel capitolo 3, per mantenere una distribuzione sempre uniforme dei collegamenti casuali, ogni *peer* destinatario di un messaggio per la costruzione di un collegamento casuale, memorizza il collegamento in direzione opposta se il *peer* che ha inviato il messaggio sta costruendo per la prima volta la propria tabella di *routing*, ossia, se il *peer* è appena entrato nella rete. Il numero di collegamenti è abbastanza elevato ma rimane stabile. Non

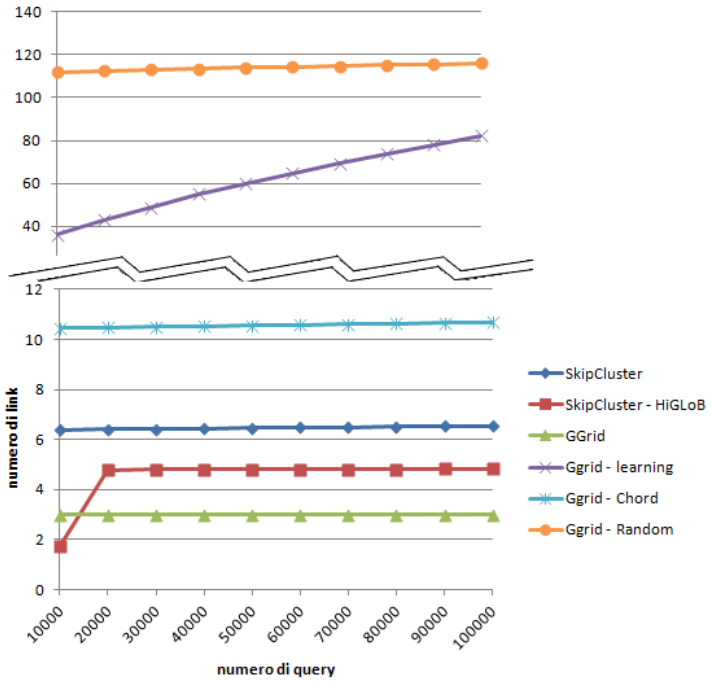


Figura 4.9: Numero di *link* necessari al mantenimento della struttura

si può dire altrettanto della versione Learning di GGrid. Il grafico evidenzia una retta con forte pendenza che raggiungerà, come si può dedurre dalla teoria alla base del Learning, il valore del numero di *peer* della rete. A regime, quindi, il numero di collegamenti aumenterà di tanto quanto aumenterà la dimensione della rete. Ad ogni *peer* in più nella rete, quindi, corrisponderà un collegamento in più per ogni *peer*. GGrid in versione Chord, per costruzione, ha un numero di *link* pari al logaritmo del numero di nodi della rete. L'aumento del numero di nodi aumenta perciò il numero di *link* in maniera logaritmica anzichè lineare. Questo permette di avere un aumento significativo solo con aumento spropositato di ingressi nella rete. L'aumento di un collegamento per *peer*, ad esempio, si avrà solo quando la rete raddoppierà le proprie dimensioni. Anche SkipCluster ha, all'interno di ogni *cluster*, collegamenti logaritmici, ed è per questo che l'aumento di *link* non è

significativo fintantochè la dimensione della rete raddoppi. HiGLoB porta, come in precedenza, un numero minore di collegamenti necessari distribuendo in modo più equilibrato, i *peer* nei diversi *cluster*. Dal punto di vista di occupazione di memoria, però, l'istogramma di HiGLoB occuperà uno spazio pari circa al numero di *cluster*, ossia di poco superiore a 100. Infine notiamo che GGrid in versione base maniene un bassissimo numero di collegamenti per nodo a causa della sua struttura ad albero binario.

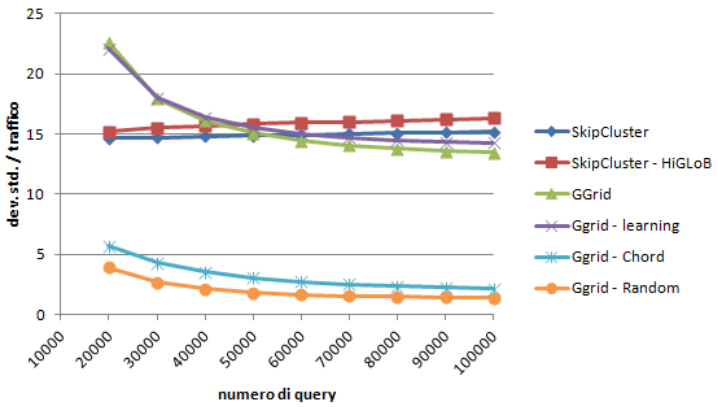


Figura 4.10: Distribuzione del traffico

Infine, in figura 4.10, notiamo l'andamento della distribuzione del traffico di rete nei diversi *overlay*. In questo scenario, rispetto al precedente, osserviamo che l'aumento di *peer* nella rete porta ad un aumento della disparità di traffico smaltito per *peer*. Gli *overlay* che mostrano una maggiore differenza in termini di distribuzione del traffico di rete sono quelli composti da SkipCluster. Anche GGrid, seppur in minor quantità, risente negativamente dell'introduzioni di nuovi nodi nella rete. Ancora una volta notiamo che, mentre la media degli altri *overlay* è attorno a 15, GGrid in versione Chord ed in versione Random mostrano una distribuzione del traffico che tende a 1, portando una distribuzione del traffico di rete quasi uniforme. Dei due migliori *overlay* sotto questo parametro evidenziamo che i *link* casuali portano ad un valore più basso rispetto all'organizzato sistema Chord.

4.3.3 Rete con ingressi ed uscite

Il terzo scenario sarà composto dalla fase iniziale, corrispondente all'ingresso di 10000 nodi nella rete, seguito da una fase in cui vengono eseguite 100000 *query*. Ogni 100 *query*, inoltre, si eseguirà un'operazione di connessione o disconnessione di un nodo con probabilità del 50%. Il numero di nodi si aggirerà intorno al valore iniziale di 10000 dal momento la probabilità di connessione è la stessa di disconnessione. Il grafico in figura 4.11 mostra la dimensione della rete ogni 10000 *query*. Rispetto al grafico dello scenario precedente non vi è la linearità tra i punti che costituiscono la figura. Tra ogni fotografia il numero di nodi cambia in maniera casuale. Non è qundi da intendere che, ad esempio tra 60000 e 80000, il numero di nodi cresca uniformemente da 9983 a 10023. All'interno di ogni intervallo, la dimensione della rete resterà variabile senza un andamento prefissato.

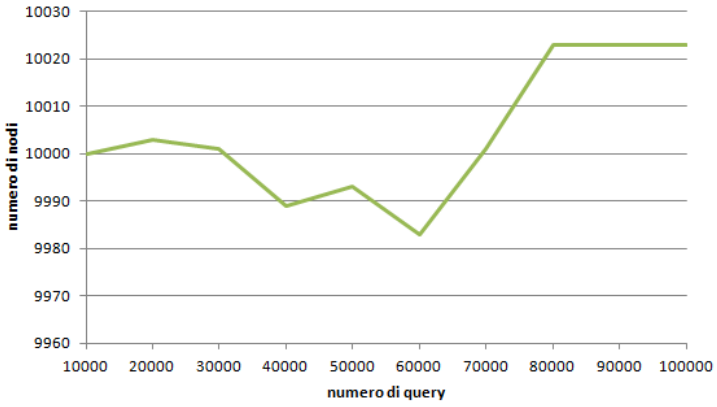


Figura 4.11: Numero di nodi della rete all'aumentare delle *query*

Questo scenario presenta circa le stesse caratteristiche di quelli precedentemente analizzati. Anche qui, come si osserva dalla figura 4.12, l'*overlay* con traffico di sistema maggiore è GGrid in versione Chord. La struttura organizzativa obbliga lo scambio di un elevato numero di messaggi per mantenere l'architettura coerente con i vincoli imposti da Chord. Segue poi SkipCluster con HiGLoB. In questo *overlay* HiGLoB penalizza SkipCluster in quanto per mantenere ag-

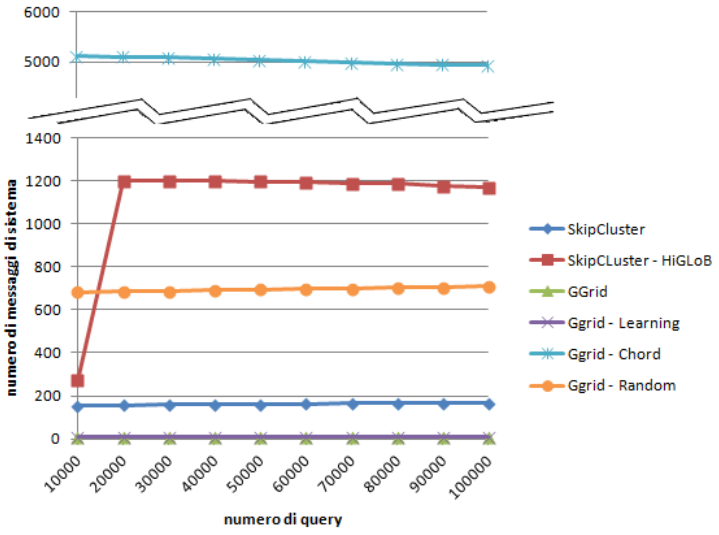


Figura 4.12: Traffico di sistema

giornato l'istogramma dei *cluster* necessita di un elevato scambio di messaggi di sistema. Ggrid in versione Random crea un traffico di rete non eccessivamente elevato, ma sempre decisamente più elevato degli *overlay* in versione base. SkipCluster presenta un traffico di rete abbastanza ridotto, mentre, come evidenziato precedentemente, un bassissimo costo di mantenimento.

Dal grafico in figura 4.13 notiamo che il numero medio di *hops* segue circa lo stesso andamento degli altri scenari nei diversi *overlay* sperimentati. Dal momento che vi sono inserzioni ed eliminazioni, le curve rappresentate, presentano un andamento non rettilineo. Questo dipende dalla quantità e dalla posizione dei collegamenti persi ed entrati. In linea di massima troviamo la stessa situazione osservata negli scenari precedenti, con GGrid in versione Random che presenta il più alto *hop* medio, seguito dalle due versioni di SkipCluster e GGrid in versione Chord. Anche questa volta GGrid in versione base e con Learning presentano un numero di *hops* molto basso per le stesse motivazioni date in precedenza.

La figura 4.14 mostra il numero di *link* medio per *peer* necessari al

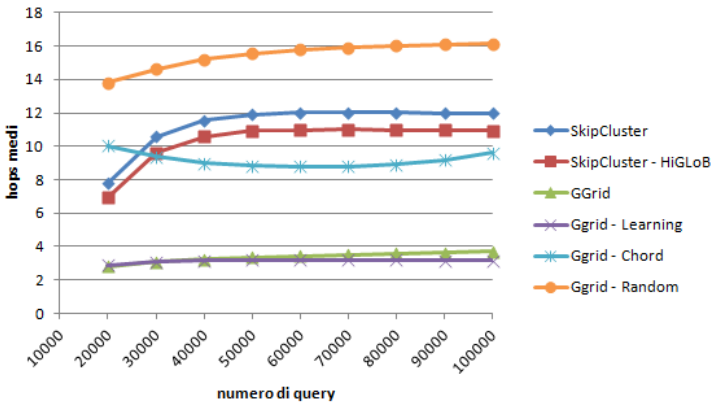


Figura 4.13: Numero medio di *hops* per *peer*

mantenimento della struttura rappresentata dall'*overlay*. Come evidenziato negli scenari precedenti, l'*overlay* con il più alto numero di collegamenti è GGrid in versione Learning, che a regime ha un numero di collegamenti che segue il numero di *peer* presenti nella rete. A seguire troviamo GGrid in versione Random che mantiene un numero di collegamenti piuttosto costante, intorno ai 100. Troviamo poi GGrid in versione Chord, che risente in maniera minimale la dinamicità della rete. Troviamo poi SkipCluster con un numero di collegamenti piuttosto basso e la relativa versione con HiGLoB che diminuisce ancora una volta questo valore. Infine, come ribadito precedentemente, troviamo GGrid in versione base, la cui struttura non permette un numero di collegamenti maggiore di 3 (un padre e due figli).

Anche l'ultimo grafico, presente in figura 4.15, è simile a quello relativo alla distribuzione del traffico degli scenari precedenti. Da evidenziare, in questo scenario, che la versione Chord di GGrid, mostra un più elevato valore di distribuzione, che indica che la rete cala leggermente le proprie prestazioni dal punto di vista della distribuzione del traffico di rete quando si hanno disconnessioni e connessioni. Quindi, GGrid in versione Chord è più vulnerabile alla dinamicità della rete per quanto concerne l'equa distribuzione del traffio di rete.

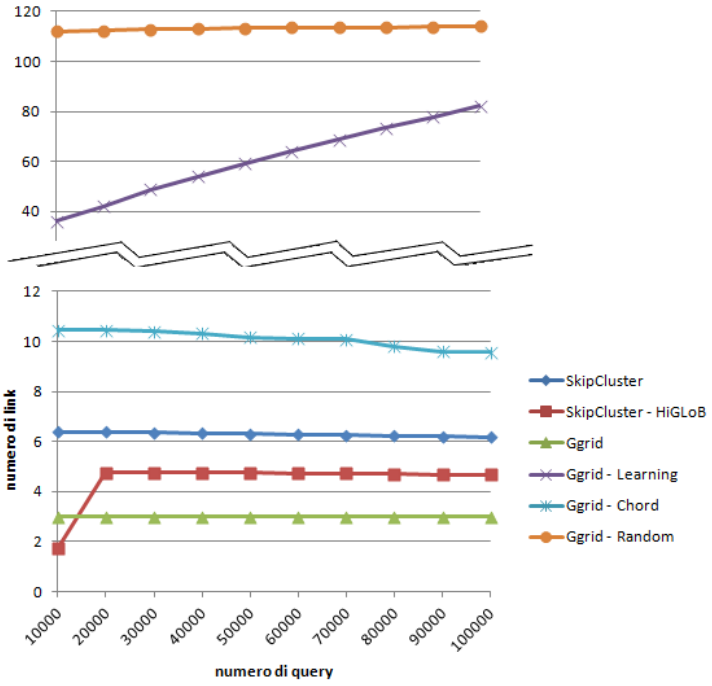


Figura 4.14: Numero di *link* necessari al mantenimento della struttura

4.3.4 Rete con ingressi esponenziali

In quest'ultimo scenario vediamo in azione un teorema dimostrato dal prof. Gianluca Moro in cui, dopo una fase di connessione di 10000 nodi iniziale, le connessioni successive sono pari al valore della radice delle *query* eseguite nel sistema. Il grafico in figura 4.16 mostra la curva della dimensione della rete con l'aumentare delle *query* eseguite su di essa.

Dal grafico in figura 4.17 si evince ancora che il costo di GGrid in versione CHord è di gran lunga superiore a quello degli altri *overlay*. Anche in questo caso il costo si abbassa con i successivi inserimenti in quanto questi hanno una frequenza relativamente bassa. La cifra rimane però molto più elevata degli altri *overlay* osservati. Come per gli altri scenari osserviamo poi che SkipCluster con HiGLoB ricopre un costo molto più elevato di SkipCluster in versione base, causato dalla

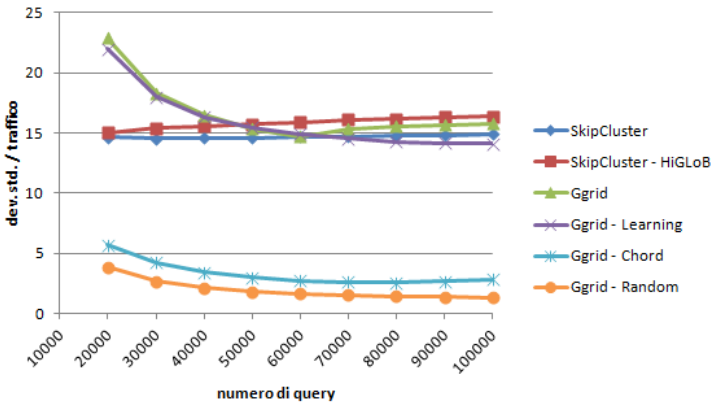


Figura 4.15: Distribuzione del traffico

costruzione e dal mantenimento dell'istogramma e dalle operazioni di bilanciamento effettuate. La versione Random di GGrid mostra ancora una volta un costo non eccessivo mantenuto costante nonostante gli ingressi di nuovi nodi. Infine, come per gli altri scenari, GGrid in versione base e con Learning forniscono un costo molto contenuto grazie alla sua struttura ad albero binario.

Anche per quanto riguarda il numero di *hop* (presente in figura 4.18) medio lo scenario presenta circa le stesse caratteristiche evidenziate negli scenari precedenti. Tutti gli *overlay* presentano un *hop* che rimane sullo stesso ordine di grandezza. Osserviamo ancora una volta che GGrid in versione Random mostra l'*hop* più elevato, seguito, a distanza di 4 unità, da SkipCluster. La versione HiGLoB di SkipCluster si distacca dalla sua versione base diminuendo di circa una unità il numero di *hops*. La versione Chord di GGrid mostra un *hop* medio pari circa alla metà di quello offerto dalla versione Random. Infine, GGrid in versione base, e meglio ancora GGrid in versione Learning, offrono un *hop* medio molto basso, rispettivamente di circa 4 e circa 3.

Il grafico relativo al numero di collegamenti necessari al mantenimento, presente in figura 4.19, mostra che l'*overlay* GGrid con Learning presenta ancora la caratteristica di una retta pendente, che tende al valore della dimensione della rete. Questo fattore critico comporta

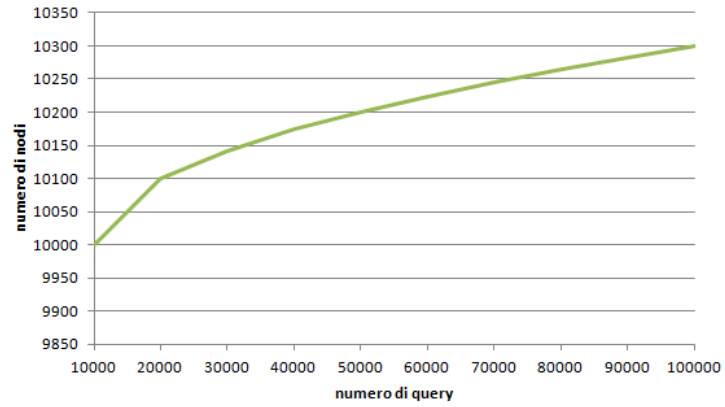


Figura 4.16: Numero di nodi della rete all'aumentare delle *query*

quindi anche un'elevata quantità di spazio di memorizzazione. Osserviamo poi che GGrid in versione Random ha un numero di *link* abbastanza elevato rispetto agli *overlay* rimanenti ma costante e, quindi, non critico come per il caso con Learning. Proseguendo nell'analisi del grafico notiamo che GGrid in versione Chord necessita di circa $\log n$ collegamenti, dove n è il numero di nodi del sistema. SkipCluster in versione HiGLoB presenta un miglioramento rispetto alla sua versione base in quanto crea circa 2 collegamenti in meno per *peer*. Infine, possiamo osservare che GGrid, grazie alla struttura di cui è composto, necessita di un numero di collegamenti più basso di tutti gli altri *overlay*.

Osserviamo, per concludere, il grafico della distribuzione del traffico di rete in figura 4.20, che mostra che tutti gli *overlay*, al di fuori di GGrid in versione Chord e Random, hanno una distribuzione poco equilibrata del traffico. Nel dettaglio, la migliore performance rispetto a questo parametro spetta a GGrid in versione Random, che tende tale valore a 1, fornendo una distribuzione del traffico di rete quasi completamente equilibrata.

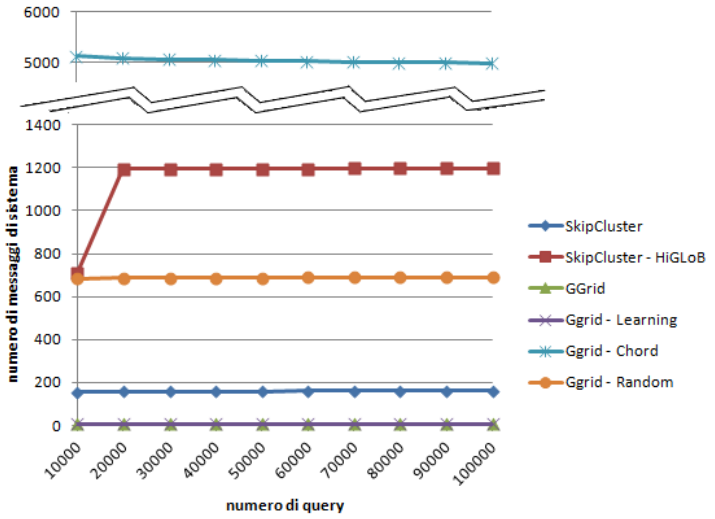


Figura 4.17: Traffico di sistema

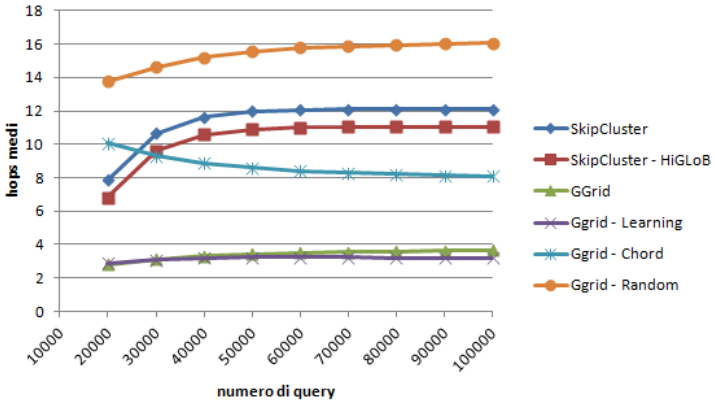


Figura 4.18: Numero medio di *hops* per *peer*

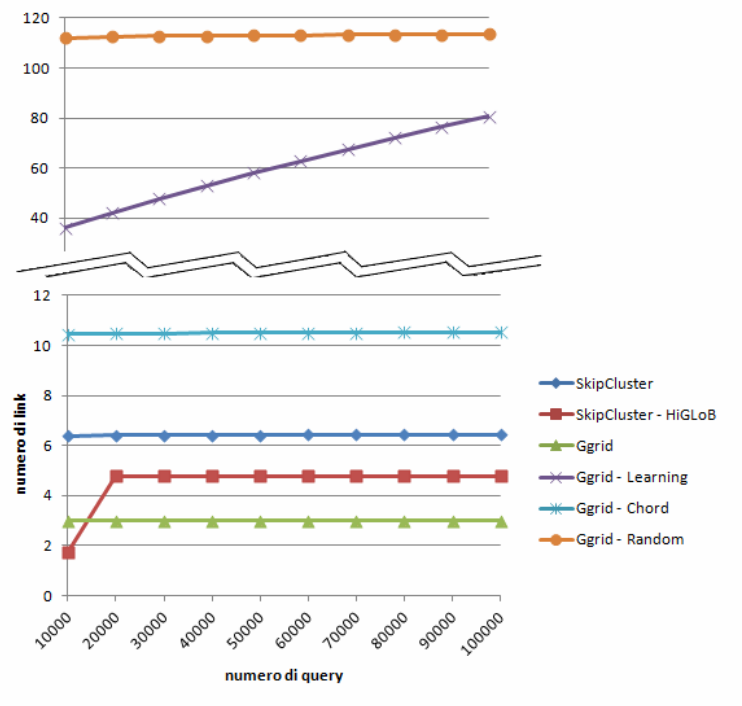


Figura 4.19: Numero di *link* necessari al mantenimento della struttura

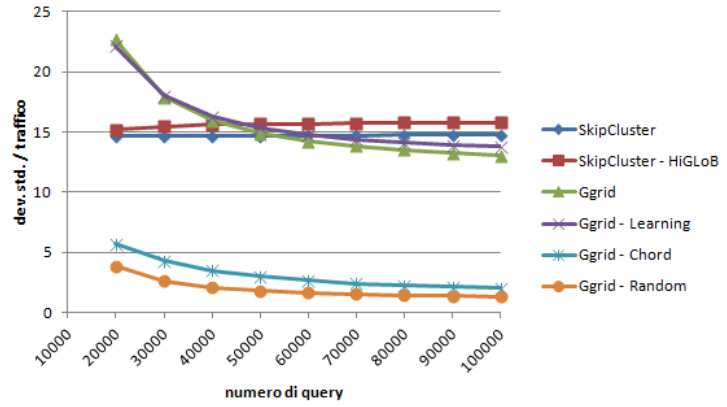


Figura 4.20: Distribuzione del traffico

Capitolo 5

Conclusioni

Dopo aver analizzato i risultati degli esperimenti condotti, siamo in grado di trarre alcune conclusioni, nonchè a presentare spunti per lavori futuri che sperimenteranno quanto realizzato in ambienti sempre più realistici e pratici.
Innanzitutto riportiamo le principali caratteristiche degli *overlay* sviluppati indipendentemente dallo scenario utilizzato. Ovviamente riferiremo i valori ottenuti alla dimensione sperimentata, in quanto, ogni *overlay* varia le proprie performance in maniera sensibile al numero di *peer* collegati alla rete.
Partendo da SkipCluster in versione base, notiamo che il costo di mantenimento non è eccessivamente elevato, circa un cinquantesimo della dimensione della rete. Questo significa che l'ingresso e l'uscita dei nodi da questa rete comportano un costo di mantenimento relativamente basso. Inoltre, la struttura presenta un *hop* medio pari al valore logaritmico del numero di *peer* presenti e connessi alla rete. Anche il numero di *link* necessari al mantenimento della struttura è molto basso, pari circa alla metà del valore logaritmico della dimensione. Il quoziente tra la deviazione standard ed il traffico di lavoro medio è anch'esso in valore logaritmico. Quest'ultimo parametro rappresenta l'unica debolezza di SkipCluster, nonostante mostri buone performance rispetto agli altri valori. La versione HiGLoB di SkipCluster, mostra, rispetto alla sua versione base appena descritta, un numero di collegamenti medio per nodo minore di circa un terzo. Questo valore permette di mantenere un numero di connessioni minore diminuendo,

quindi la banda. Nonostante ciò, avremo un'occupazione di memoria maggiore alla versione base di SkipCluster in quanto ogni nodo dovrà memorizzare l'istogramma dei *cluster*, calcolata come circa un centesimo della dimensione della rete. Inoltre, per mantenere l'istogramma aggiornato, questa versione dovrà utilizzare un numero di messaggi pari circa al doppio di quelli utilizzati per il mantenimento della versione base di SkipCluster.
Spostando la nostra attenzione su GGrid, notiamo che la versione base mostra già notevoli vantaggi rispetto a SkipCluster. Innanzitutto ha un costo di mantenimento ridotto al minimo in quanto l'inserimento o la disconnessione di un *peer* comportano un cambiamento in un nodo dell'albero che mantiene, al massimo tre collegamenti. Anche il numero di collegamenti è un punto di forza di GGrid. Questo valore è ridotto all'osso grazie alla struttura ad albero binario utilizzata. Il numero medio di *hop* è anch'esso di gran lunga inferiore rispetto a quello di SkipCluster e, a regime, si stima arriverà al valore logaritmico della dimensione della rete. Per quanto riguarda la distribuzione del traffico, invece, GGrid ottiene le stesse performance di SkipCluster, mostrando quindi lo stesso problema di bilanciamento. GGrid in versione Learning mostra le stesse caratteristiche in termini di traffico di sistema, in quanto il valore aggiunto viaggia in *piggyback*, ossia a cavallo di messaggi generati per il mantenimento e l'operatività di GGrid in versione base. Il miglioramento apportato consta nella diminuzione del numero di *hop*, che, a regime, arriverà circa ad uno. Per contro, lo svantaggio sta nel numero di collegamenti che cresce linearmente alla dimensione della rete. Questo dato ha un doppio svantaggio: richiede una grande quantità di memoria dei nodi e, in caso si implementassero funzioni di *ping* per i collegamenti attivi, un'elevata occupazione di banda per testare i tanti collegamenti. Anche in questo caso la distribuzione del traffico è piuttosto impari tra i vari nodi della rete.
Per quanto riguarda GGrid in versione Chord, notiamo subito che necessita di un elevatissimo numero di messaggi di sistema, pari circa alla metà della dimensione della rete. Questa struttura porta un *hop* medio pari circa alla metà del logaritmo del numero di nodi connessi ed un numero di collegamenti medio per *peer* pari, come ci si può aspettare per costruzione, al valore logaritmico del numero di nodi. Nonostante questa versione degradi le performance rispetto alle ver-

sioni base e Learning di GGrid, possiamo osservare che il traffico è distribuito in maniera quasi uniforme, portando il quoziente tra deviazione standard e media del traffico di lavoro pari, circa, a 2.
L'ultima versione di GGrid, in versione Random, nasce per cogliere i vantaggi di entrambe le architetture: versione base e Chord di GGrid. Si può infatti notare che il numero di messaggi di sistema è piuttosto ridotto rispetto a quello richiesto dalla versione Chord, e si posiziona a circa un quindicesimo del numero di nodi della rete. Il numero di *hop* aumenta, portandosi ad un paio di unità rispetto al valore logaritmico del numero di nodi, così come il numero di collegamenti, che si posiziona ad un centesimo della dimensione della rete. Questo degrado di performance viene però compensato dal rapporto deviazione standard su media del traffico di lavoro che rimane circa ad uno, esprimendo una distribuzione del traffico di rete che può essere considerato uniforme a meno di una impercettibile approssimazione. Quest'ultima versione di GGrid permette quindi un'ottima distribuzione del traffico a scapito di un lieve calo delle altre performance misurate.
Gli esperimenti effettuati sino ad ora avevano un carico uniforme, considerato unitario. Nei prossimi lavori sarà interessante valutare le performance in reti con distribuzione del carico non uniforme, considerando più elementi per *peer*, aumentando il numero di regioni ed il bucket-size massimo per ogni nodo. Queste simulazioni potranno misurare le performance delle architetture in casi reali. Inoltre, sarà interessante valutare la possibilità di realizzare HiGLoB per la distribuzione del carico sia in SkipCluster che in GGrid.
Le analisi e le considerazioni presentate in questa tesi hanno, quindi, lo scopo principale di mostrare le capacità di GGrid in ambienti ideali, confrontandolo con uno dei più recenti *overlay* realizzati, SkipCluster. L'elaborato crea innumerevoli spunti per la prosecuzione dello studio e dell'analisi di GGrid in ambienti *peer-to-peer* realistici.

Appendice A

Le reti piccolo mondo

La teoria dei piccoli mondi [10,15,16,17,18] nasce come ramo della teoria dei grafi e deve la sua esistenza alla fusione di studi riguardanti numerose discipline e trova applicazione in innumerevoli campi della vita pratica.
I primi esperimenti riguardo questa teoria iniziarono negli anni '60. Il contributo più importante fu quello Stanley Milgram [14], psicologo e sociologo statunitense ricordato soprattutto per i suoi studi riguardanti la determinazione del comportamento individuale. Nel dettaglio, Milgram si prefisse l'obbiettivo di studiare il comportamento umano quando entrava in relazione con altri individui, tramite catene di conoscenze. Risale al 1967 il primo vero esperimento condotto in merito. L'esito delle ricerche fu alquanto strabiliante.

Per testare la teoria, Milgram selezionò casualmente un gruppo di americani del Midwest e chiese loro di mandare una lettera ad un estraneo che abitava nel Massachusetts, a diverse migliaia di chilometri di distanza. Ognuno di essi conosceva il nome del destinatario, la sua professione e la zona in cui risiedeva ma non l'indirizzo preciso. Fu, quindi, chiesto a ciascuno dei partecipanti all'esperimento di mandare la propria lettera ad una persona da loro conosciuta che a loro giudizio avesse il maggior numero di possibilità di conoscere il destinatario finale. Quella persona avrebbe fatto lo stesso, e così via fino a che la lettera non sarebbe giunta realmente al destinatario. Contrariamente alle aspettative (i promotori dello studio si aspettavano che la catena comprendesse perlomeno un centinaio di intermediari), ci vollero in

media tra i cinque e i sette passaggi per conosegnare la lettera al destinatario. La eco mediatica fu clamorosa. L'espressione "sei gradi di separazione" venne assodata, di fatto, nella cultura popolare.

Nel 1973, il sociologo Mark Granovetter, rimasto affascinato dal risultato dell'esperimento di Milgram, continuò lo studio delle reti costituite da persone collegate tramite la loro semplice conoscenza approfondendo, quindi, la teoria delle reti sociali. Una rete sociale consiste di un qualsiasi gruppo di persone connesse tra loro da diversi legami sociali, che vanno dalla conoscenza casuale, ai rapporti di lavoro, ai vincoli familiari. I collegamenti che si instaurano tra di loro rappresentano le conoscenze che ogni individuo ha con il resto della società.
Per spiegare meglio la rete sociale ipotizziamo di inserire un individuo in una rete e di collegarlo ad altre persone, amici o parenti, che appartengono alla stessa rete. Ipotizziamo, anche, che il collegamento tra individui si crea se l'uno conosce, o anche semplicemente saluta, l'altro. Partendo da questo presupposto, potremmo collegare un determinato individuo con i propri parenti e tutti i suoi conoscenti allo stesso modo. Se pensiamo che in una società le relazioni non sono tutte uguali, ma si distinguono dal grado di conoscenza, noteremo che la teoria appena esposta pecca di un'importante dettaglio. Per questo, Granovetter, distingue in due tipologie i legami che possono legare due persone, prossime a loro stesse: legami forti e legami deboli. Per fare un esempio, osserviamo un individuo della società. Egli avrà una sua cerchia di amici, coi quali nutre un legame forte, ed alcuni conoscenti, che, ad esempio, vede una volta l'anno, con i quali ha instaurato legami deboli. Se osserviamo ora, con attenzione, il gruppo di amici con il quale l'individuo ha un legame forte, noteremo che, ogni persona del gruppo, avrà approssimativamente lo stesso legame forte con gli altri componenti degli amici. Più semplicemente, è molto probabile che due amici dell'individuo preso in esame, siano anche amici tra loro. Questa proprietà caratterizza solo i legami forti. Nei legami deboli, infatti, un amico di un conoscente, difficilmente sarà nostro amico. Nella figura A.1 analizziamo come due gruppi di amici siano collegati con un legame debole. I legami deboli fungono, dunque, da ponte tra piccole porzioni di mondo costituito da legami forti.

Alla luce di queste considerazioni, Mark Granovetter, spiegò che

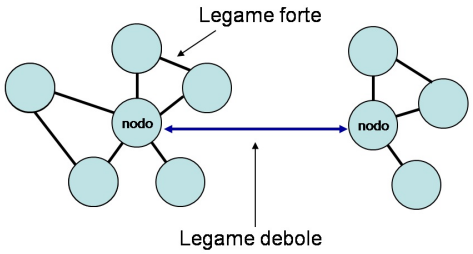


Figura A.1: Esempio di rete sociale con legami deboli e forti

- a differenza dell'espressione che li definisce - i legami più importanti, sono proprio i legami deboli. Chiariamo l'importanza dei legami deboli con un esempio pratico. Supponiamo che un individuo voglia mettere in circolazione la notizia riguardo al fatto che sta cercando lavoro. Se lo facesse unicamente nel suo gruppo di amici, il messaggio circolerebbe nel gruppo e gli amici, fortemente in contatto tra loro, si ripeterebbero la notizia vicendevolmente. Sfruttando i legami deboli, la notizia potrebbe arrivare ad un conoscente che potrebbe, a sua volta, avere un legame forte con un gruppo che sta offrendo lavoro. Con un ulteriore esempio mostreremo, inoltre, quanto sia più critica l'eliminazione di un legame debole rispetto a quella di un legame forte. Se due amici spezzano un legame forte che esiste tra loro, infatti, essi potrebbero collegarsi attraverso un terzo amico del gruppo, aggiungendo solamente un passaggio e lasciando comunque un percorso che li collega. Se eliminassimo un legame debole, un individuo perderebbe anche il gruppo di soggetti collegati tra loro tramite legami forti. Si interromperebbe, dunque, il collegamento tra due piccoli mondi, portando all'interuzione di collegamenti con interi gruppi di persone. A seguito di queste osservazioni, Granovetter, rinominò questa caratteristica "la forza dei legami deboli".

Negli anni a seguire, furono eseguiti diversi esperimenti a conferma

che quanto stabilito da Milgram e Granovetter potesse essere applicato a tutte le reti sociali. Degno di nota, risalente ai primi anni '90, è quello che è stato rinominato l'*oracolo di Kevin Bacon*, sviluppato da alcuni studenti dell'Università della Virginia. Kevin Bacon è un famosissimo attore cinematografico; il programma sviluppato permette di selezionare qualsiasi altro personaggio del cinema e di stabilire il numero di collegamenti tra Bacon e l'attore scelto. I collegamenti sono, ovviamente, costituiti dalle collaborazioni nei film. Emerse, anche in questo caso, che Bacon distava da qualsiasi altro attore ancora in media sei passaggi. Ovviamente Bacon venne scelto in maniera del tutto casuale e, di certo, questo attore non ha partecipato a più film rispetto ad altri attori presenti nel database del programma. Per testare il programma è possibile consultare il sito internet *www.oracleofbacon.org*.
La teoria dei sei gradi di separazione trova un riscontro veritiero anche in termini matematici. Se pensiamo che in una vita un individuo colleziona 50 conoscenze tra amici e parenti osserviamo che, con soli due passaggi, potrebbe conoscere il quadrato delle persone, ossia 2500, con tre passaggi 125000 e così via, fino ad arrivare, in sei passaggi a circa sedici mila milioni di individui. Dal momento che questo valore supera di gran lunga il numero di persone presenti attualmente nel mondo, è facile pensare che la teoria sia provata con successo. Quello che non è stato considerato è che gruppi di persone legate da legami forti sono anche collegate tra di loro, diminuendo il numero di persone che si possano conoscere ad ogni passaggio.

Lo studio della teoria venne ripreso con gran fermento nel 1998 da parte di due studiosi, Duncan Watts e Steve Strogatz. Fino ad allora, in letteratura, possiamo trovare solo osservazioni dei risultati, ma non vi erano spiegazioni delle caratteristiche che la rete sociale doveva mantenere. Fu proprio questo l'obiettivo dello studio che si prefissero: trovare il modo di costruire una rete piccolo mondo in modo da poter permettere la creazione, in qualsiasi ambito, di reti con le stesse qualità. I due studiosi partirono da un grafo ad anello completamente ordinato, in cui ogni vicino è connesso con un numero limitato di persone più vicine. In questa configurazione, attraverso simulazioni svolte in un computer, rilevarono che i gradi di separazione erano molto più elevati di quelli delle reti sociali. Una prima conclusione stabiliva che

una rete piccolo mondo non poteva essere una rete completamente ordinata. Lo studio proseguì sostituendo, ad alcuni dei collegamenti esistenti, alcuni collegamenti casuali, che connettevano, quindi, in maniera casuale, due punti del grafo. Mano a mano che aggiungevano collegamenti, il grado di separazione della rete diminuiva. Un altro fattore che tenevano sotto controllo era il coefficiente di aggregazione. Questo coefficiente, in percentuale, fornisce la stima di quanti dei nodi collegati ad un nodo comune, siano collegati anche tra di loro. Per capire meglio possiamo dire, trattandosi di reti sociali, quanti amici sono amici tra di loro. Nelle società questo fattore è, come abbiamo detto in precedenza parlando dei legami forti, si avvicina al 60%. In una rete completamente ordinata, il fattore di coefficiente rilevato era del 67%. Questo valore è piuttosto elevato per rappresentare una rete sociale. Infatti, indicherebbe che due persone su tre di quelle che un individuo conosce si conoscono anche tra di loro. Inserendo i collegamenti casuali il coefficiente di aggregazione si abbassa. Ovviamente, proseguendo l'esperimento, arrivarono al punto finale in cui la rete era costituita solo da collegamenti casuali. Questa rete mostrava un grado di separazione molto basso ma, di contro, offriva un coefficiente di aggregazione anch'esso molto basso, pari a circa 0.001%. Questo indicherebbe che quasi nessuna persona conosce l'amico del proprio amico. In una rete sociale questo è impossibile in quanto, come abbiamo detto, in un gruppo di amici è molto probabile che due amici di una terza persona si conoscano tra loro. Viste le simulazioni effettuate, Watts e Strogatz dimostrarono che una rete piccolo mondo è una rete che si trova a metà strada tra una rete completamente ordinata ed una completamente casuale, come si vede in figura A.2. Questo modello di rete prende, appunto, il nome di Modello di Watts e Strogatz.

Nella figura A.3 evidenziamo i risultati degli esperimenti di Watts e Strogatz. Dalla figura si osserva che maggiore è il numero di collegamenti causali, chiamati anche *shortcuts*, minore è la distanza. Il coefficiente di aggregazione, inceve, rimane piuttosto elevato fino ad una soglia critica che lo fa scendere, in maniera piuttosto brusca, portando ad un distaccamento dalle reali condizioni in cui operano le reti sociali.

Rispetto alle reti ben organizzate, le reti casuali mostrano cam-

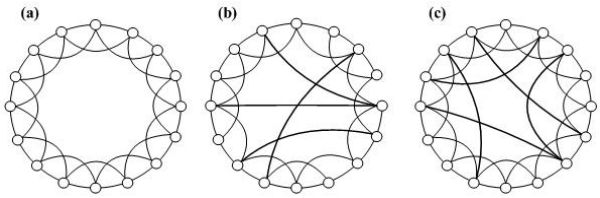


Figura A.2: a) rete ordinata; b) modello di Watts e Strogatz; c) rete casuale

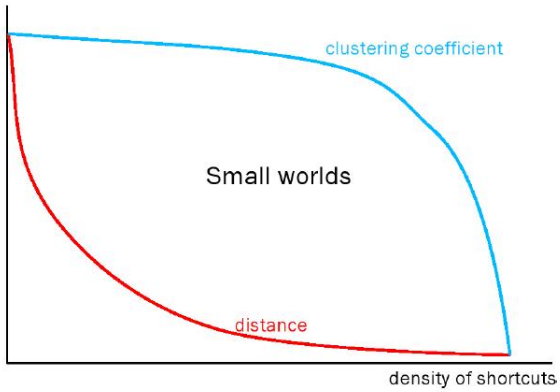


Figura A.3: Risultati emersi dagli esperimenti di Watts e Strogatz: dimensione e coefficiente di aggregazione all'aumentare dei collegamenti casuali

mini brevi in termini di distanza tra due nodi qualsiasi ed una alta resistenza in caso di guasti. Le reti ordinate, infatti, oltre a presentare un cammino più lungo, in caso di guasti, dovrebbero essere ristrutturate per permettere una riorganizzazione che rispecchi i vincoli di struttura imposti dal tipo di rete scelto. Anche la distribuzione dei gradi, ossia la distribuzione di probabilità di connessione dei diversi nodi della rete, è migliore in una rete casuale piuttosto che in una rete ordinata. Di contro, come vediamo dal grafico in fugura A.3, il livello di clusterizzazione scende bruscamente quando ci si avvicina troppo alle reti casuali. Le reti *small world*, quindi, si collocano al centro tra reti ordinate e casuali, ereditando i pregi, seppur in maniera legger-

mente attenuata, di entrambe le reti. Una rete *small world* presenterà, quindi, le seguenti caratteristiche:

- cammini brevi;
- distribuzione equa dei gradi;
- alta resistenza in caso di guasti;
- buon coefficiente di clusterizzazione.

Tutte queste caratteristiche hanno pilotato la nostra scelta facendoci optare per questo tipo di struttura da applicare come *overlay* in GGrid. Osservando i risultati troveremo i pregi elencati confermando le teorie di Watts e Strogatz.

Bibliografia

[1] Ouksel A. M., Moro G.: *GGrid: A Class of Scalable and Self-Organizing Data Structures for Multi-dimensional Querying and Content Routing in P2P Networks*, Technical Report no. DEIS-LIA-002-04, February 2004, University of Bologna, University of Illinois at Chicago

[2] Harvey N. J. A., Jones M. B. Saroiu S., Theimer M., Wolman A.: *SkipNet: a scalable overlay network with practical locality properties*, USITS'03 Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, 2003

[3] Xu M., Zhou S., Guan J.: *A new effective hierarchical overlay structure for Peer-toPeer networks*, Computer COmmunications, October 2010

[4] Girdzijauskas S., Galuba W., Darlagiannis V., Datta A., Aberer K.: *Fuzzynet: Ringless routing in a ring-like structured overlay*, Peer-toPeer Networking and Application, vol. 4, num. 3, 2011

[5] Sarunas G., Datta A., Aberer K.: *Oscar: Small-world overlay for realistic key distributions*, The Fourth International Workshop on Databases, Information Systems and Peer-toPeer Computing, 2006

[6] Bharamble A. R., Agrawal M., Seshan S.: *Mercury: Supporting Scalable Multi-Attribute Range Queries*, Jenuary 2004

[7] Vu Q. H., Ooi B. C., Rinald M., Tan K.-L.: *Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems*, IEEE

Transactions on knowledge and data engineering, Vol. 21, No. 4, April 2009

[8] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan: *Chord: A scalable peer-to-peer lookup service for internet applications*, Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, 2001

[9] M. Buchanan: *Nexus: Small Worlds and the New Science of Networks*, W.W. Norton & Co, New York, 2002

[10] D. Watts, S. H. Strogatz,*Collective dynamics of small-world networks*, Nature, June 1998

[11] PeerSim: A Peer-to-Peer Simulator, http://peersim.sourceforge.net/

[12] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, I. Stoica, *Load Balancing in Structured P2P Systems*, Lecture Notes in Computer Science, 2003, Volume 2735/2003, 68-79

[13] A. Brinkmann, Y. Gao, M. Korzeniowski, D. Meister, *Request Load Balancing for Highly Skewed Traffic in P2P Networks*, 2011 IEEE Sixth International Conference on Networking, Architecture, and Storage

[14] S. Milgram, *The small world problem*, Psychology today, Volume 2, 60-67, 1967

[15] Amaral, L.A.N. and Scala, A. and Barthélémy, M. and Stanley, H.E., *Classes of small-world networks*, Proceedings of the National Academy of Sciences, Volume 97, 21, 2000

[16] Kleinberg, J.M. and others, *Navigation in a small world*, Nature, Volume 406, 845, 2000

[17] Latora, V. and Marchiori, M., *Efficient behavior of small-world networks*, Physical Review Letters, Volume 87, 19, 2001

[18] Wang, XF and Chen, G., *Synchronization in small-world dynamical networks*, International Journal of Bifurcation and Chaos in Applied Sciences and Engineering, Volume 12, 187-192, 2002

[19] Aspnes J, Kirsch J, Krishnamurthy A *Load balancing and locality in range-queriable data structures*, PODC2004, 2004

[20] Freedman MJ, Lakshminarayanan K, Rhea S, Stoica I, *Non-transitive connectivity and dhts*, Proceedings of the 2nd conference on Real, Large Distributed Systems, pp. 10-10, 2005, USENIX Association, Berkeley, CA, USA

[21] A. Rowstron, P. Druschel, *Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems*, Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms, 2001

[22] E. Anceaume, R. Ludinard, A. Ravoaja, F. Brasileiro, *PeerCube: a hypercubebased P2P overlay robust against collusion and churn*, Proceedings of 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008

[23] B. Kroll, P.W., *Distribuiting a search tree among a growing number of processors*, Proceedings of the ACM International Conference on Management of Data (SIGMOD'94), Minneapolis, MN, USA, ACM Press (1994), 265-276

[24] Ouksel, A.M., Moro, G., Litwin, W., *GGF: A Generalized Grid Files for Distribuited Environments*, Technical Report UIC-IDS-CRIM/TECH-REPORT, No. 2002-05, University of Illinois at Chicago, DEIS University of Bologna, 2002

[25] Moro, G., Monti, G., *WGrid: a CrossLayer Infrastructure for MultiDimensional Indexing, Querying and Routing in Ad-Hoc and Sensor Networks*, P2P 2006: Sixth IEEE International Conference on PeerToPeer Computing, Cambridge, UK, IEEE Computer Society, 2006

[26] Monti G., Moro G, *W*Grid: A Robust Decentralized Crosslayer Infrastructure for Routing and MultiDimensional Data Management in Wireless AdHoc Sensor Networks*, P2P 2007: Seventh IEEE international conference on PeertoPeer computing, Galway, Ireland, 2007