UNIVERSITÀ DEGLI STUDI DI CAGLIARI
FACOLTÀ DI SCIENZE MM. FF. NN.

CORSO DI LAUREA SPECIALISTICA IN
TECNOLOGIE INFORMATICHE

RANGE QUERIES IN RETI PEER TO PEER STRUTTURATE

Relatore:
Prof. Riccardo Scateni

Tesi di Laurea Specialistica di:
Alessandro Soro

A.A. 2005 2006

Abstract

Il presente lavoro illustra gli algoritmi e le strutture dati necessarie per l'implementazione di una struttura dati ditribuita per la memorizzazione di collezioni di dati che supporti l'esecuzione efficiente di query su intervalli. La Skip List Distribuita usa come overlay di appoggio la Distrbuted Hash Table (DHT) e la arricchisce con la possibilità di effettuare range queries, pur mantenendone sostanzialmente inalterate le doti principali: scalabilità, fault tolerance, self maintenance e auto bilanciamento. Gli algoritmi di inserimento e ricerca, come pure l'esecuzione di query su intervalli, richiedono l'esecuzione di un numero di lookup sulla sottostante DHT proporzionale a $log(N)$ in una lista di N entry, consentendo l'archiviazione di grandi quantità di dati sfruttando spazio di memorizzazione, spesso inutilizzato, disponibile ai margini della rete Internet; Infine, si dimostra con valutazioni sperimentali che la DSL distribuisce in modo uniforme il carico di lavoro, tanto per quanto rigurda lo spazio di memoria, quanto per l'impegno di CPU, su tutti i nodi che formano la rete.

Ringraziamenti

Le idee, gli studi e le sperimentazioni descritte nel presente lavoro sono state in gran parte supportate dal CRS4, Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna, nell'ambito del progetto "Distributed Agent-Based Retrieval Toolkit" (DART), progetto di ricerca industriale cofinanziato dal Ministero dell'Università e della Ricerca Scientifica.
Un ringranziamento particolare va al Dr. Pietro Zanarini, direttore del programma Information & Communication Technology, e al Dr. Gavino Paddeu, capo area della linea di ricerca Network Dstributed Applications.
Un ringraziamento anche a tutti i colleghi dell'area NDA per i preziosi consigli e suggerimenti.

Indice

Elenco delle figure

1 Introduzione

Le architetture informatiche distribuite note come peer to peer sono progettate per consentire agli utenti di condividere risorse, come ad esempio contenuti digitali, storage o potenza di calcolo, senza alcuna intermediazione o controllo da parte di autority o servizi di coordinamento centralizzati. Nate e sviluppate nello specifico contesto della rete Internet, le architetture peer to peer sono ispirate a semplici e fondamentali principi: la capacità di adattarsi ad un ambiente estrememente dinamico, pur mantenendo sostanzialmente inalterate le proprie caratteristiche di affidabilità e connettività.

In una architettura peer to peer le risorse, oltreché remote, possono considerarsi *disperse*, in quanto la disponibilità di una spe-

cifica risorsa si considera in genere (relativamente) indipendente dalla disponibilità del nodo che la ha inizialmente condivisa.
Le tecniche specifiche per messe in atto a questo scopo variano secondo le particolari architetture e si basano da un lato sulla ridondanza delle risorse e delle connessioni, dall'altro sulla decentralizzazione degli algoritmi di routing e discovery.
Fra le più significative infrastrutture peer to peer progettate, le hash table distribuite (DHT) spiccano per la notevole scalabilità e robustezza e non a caso vengono oggi utilizzate per il discovery di risorse condivise in reti di grandissime dimensioni, come infrastruttura per varie diffusissime applicazione peer to peer.
Il maggiore limite delle DHT consiste nel fatto che esse sono intrinsecamente limitate all'esecuzione di query *exact match*, nelle quali cioè è nota a priori la chiave sotto cui la risorsa è stata memorizzata. Se tale assunto è abbastanza plausibile in una vasta gamma di servizi, ad esempio di *file sharing*, lo stesso non può dirsi per altrettante, importanti classi di applicazioni, in cui la chiave è nota in modo approssimato, o viene espressa sotto forma di vincoli, ad esempio un intervallo di valori.
Tale limitazione si può tuttavia superare per mezzo di struttu-

re dati appositamente progettate, che memorizzano un indice distribuito dei contenuti usato come guida nella ricerca di intervalli.

Il presente lavoro illustra una panoramica degli algoritmi e delle strutture dati utili a questo scopo e propone un algoritmo originale che, rispetto a quelli esistenti, è specificamente progettato per ridurre al minimo gli interventi di manutenzione sulla strttura dati, aumentando significativamente la robustezza e la facilità di implementazione.

Nel capitolo 2 vengono discusse in maggiore dettaglio le architetture peer to peer, e le principali classificazioni, in base al grado di decentralizzazione e si introdice la distinzione tra reti destrutturate e strutturate; in particolare su queste ultime si basa il presente lavoro. Sono inoltre presentati alcuni esempi di applicazioni basati su tali architetture.

Il Capitolo 3 esamina più in profondità le hash table distribuite, esempio emblematico di architetture peer to peer strutturate, con particolare riferiemento alla rete Kademlia.

Nel capitolo 5 vengono introdotte le skip list: strutture dati per la gestione di collezioni ordinate di dati, con prestazioni di inse-

rimento e ricerca simili agli alberi bilanciati, ma caratterizzate da una estrema semplicità di implementazione e manutenzione, e particolarmente adatte ad essere implementate in ambiente distribuito.

Nel capitolo 4 vengono esposti gli obiettivi del presente lavoro e viene presentato lo stato dell'arte. Sebbene non manchino in letteratura numerosi spunti per la soluzione dei problemi qui esposti, una risposta definitiva e generalmente accettata non è ancora stata formulata. I Lavori presi in esame sono risultati di ricerche recenti e, come sarà evidenziato, lasciano aperti numerosi interrogativi legati alla scalabilità e alla affidabilità delle soluzioni proposte.

Il capitolo 6 descrive in dettaglio una implementazione distribuita delle skip list e mostra in che modo essa permette l'esecuzione efficiente e affidabile di query su intervalli in reti peer to peer strutturate. Vengono descritti gli algoritmi di inserimento, lookup, e ricerca, e le strutture dati interessate; sono infine presentati alcuni risultati sperimentali che confermano la validità dell'approccio proposto.

2 Reti Peer to Peer

Una rete peer to peer, nella definizione più classica, è un sistema informatico totalmente distribuito in cui tutti i nodi ricoprono compiti e svolgono attività equivalenti. Tale definizione radicale esclude di fatto molti sistemi peer to peer esistenti, che ripartiscono i task in modo disomogeneo, ad esempio adottando dei cosiddetti super-nodi. Una definizioe meno tecnica e piu' concettuale descrive il peer to peer come un paradigma di calcolo o una classe di applicazioni che sfrutta a proprio beneficio le risorse (storage, cpu, o presenza umana) disponibili ai margini della rete Internet, ovverosia presso cosiddetti client. Tale definizione è molto più utile per individuare le caratteristiche e il campo d'azione del fenomeno peer to peer, che negli ultimi

anni ha guadagnato, nel bene e nel male, una attenzione e una base di utenti senza precedenti: oggi oltre la metà del traffico Internet è imputabile a software peer to peer di file sharing, e il volume di contenuti condivisi è stimato in centinaia di terabyte.

2.1 Applicazioni delle reti peer to peer

Le architetture peer to peer trovano applicazione in una vasta gamma di servizi, caratterizzati da una sostanziale indipendenza da autority di coordinamento.

File sharing e Content Distribution

La maggior parte dei sistemi peer to peer oggi disponibili rientrano in questa categoria, fino al punto che in taluni ambienti la tecnologia peer to peer è stata recepita come sinonimo di pirateria informatica e violazione di copyright. In realtà le applicazioni di questo tipo spaziano dai più semplici strumenti di file sharing a complesse piattaforme per la pubblicazione, catalogazione, ricerca e distribuzione di contenuti digitali. Esempi

di questa classe di applicazione includono Napster, Publius [30], Gnutella, Kazaa, Freenet [5].

Comunicazione e Collaborazione

In questa tipologia si inquadrano le infrastrutture costruite per agevolare la comunicazione diretta, generalmente interattiva, tra applicazioni o utenti della rete Internet. Esempi classici di tali applicazioni sono chat e instant messaging, ma in questo campo si stanno affermando applicazioni sempre più evolute di telepresenza e collaborazione a distanza, anche in seguito all'affermarsi di tipologie di organizzazione del lavoro a distanza, dall'outsourcing al telelavoro.

Calcolo Distribuito

La possibilità di distribuire job di calcolo su CPU remote, in particolare sfruttando i tempi di inattività dei computer presenti ai margini della rete ha stimolato la nascita di questo tipo di applicazioni, di cui il progetto SetiAthome è sicuramente l'esempio più noto. Le applicazioni di calcolo distribuito richiedono un moderato impegno di arbitraggio per dividere un job in fram-

menti che vengono successivamente inviati ai computer peer per l'elaborazione. I risultati sono successivamente convogliati verso un repository centralizzato. Questo genere di applicazioni ha tuttavia messo in luce una caratteristica tipica degli utenti di applicazioni peer to peer: la disponibilità a condividere risorse (in questo caso cicli di CPU) senza un diretto tornaconto, ma solo in virtù di un senso di appartenenza a una *community*.

Database

Se le applicazioni di ile sharing mettono sostanzialmente a disposizione degli utenti una immensa collezione di dati non strutturata, un grande impegno si sta riversando nella progettazione di veri e propri sistemi di Database basati sulle tecnologie peer to peer. Esempi di tali applicazioni includono PIER [14], un query engine capace di distribuire su migliaia di computer query relazionali in modo scalabile; Il sistema Piazza [11] applica il paradigma peer to peer alle problematiche del Semantic Web, permettendo di creare servizi distribuiti su nodi che forniscono dati e/o *schema*; Edutella [18] è un progetto open source che ha come obiettivo la definizione di standard per la pubblicazione di

metadati e la risoluzione di query su reti peer to peer.

2.2 Architetture Peer to Peer

È opportuno premettere che una rete peer to peer assolutamente decentralizzata non può essere realizzata su vasta scala data la necessità di uno o più nodi noti, detti rendezvous, che di fatto eseguono un bootstrap della rete e funzionano da punto di ingresso, per i nodi che vogliano connettersi alla rete. Al di la di questo le reti peer to peer sono comunemente classificate in base alla relazione esistente tra la posizione delle risorse nella rete e la topologia della rete stessa, e all'uso più o meno sistematico di nodi di coordinamento, o super-nodi.

2.2.1 Centralizzate e Decentralizzate

La distinzione tra reti peer to peer centralizzate e totalmente decentralizzate è relativa alla presenza o meno di un servizio di coordinamento. Alcune architetture sono infatti basate su una infrastruttura centrale che svolge in genere un servizio di indi-

cizzazione delle risorse, lasciando ai peer l'onere di distribuire le risorse stesse.

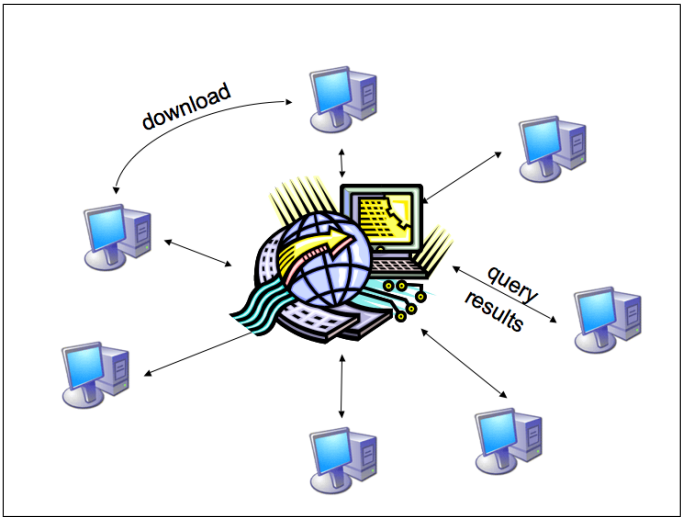

Figura 2.1: Organizzazione di una rete peer to peer centralizzata.

In figura 2.1 è rappresentata l'organizzazione di una architettura peer to peer centralizzata. I client si connettono ad un server centrale che ha il compito di mantenere:

- una tabella degli utenti connessi (indirizzo IP, porta, eventuali informazione sulla connessione come bandwidth, ecc.)

- una tabella delle risorse condivise da ogni utente, eventualmente corredata di metadati.

Al momento della connessione i client contattano il server centrale pubblicando la lista delle risorse che essi possono distribuire o condividere. Le query vengono inoltrate al server centrale che individua nelle proprie tabelle il o i peer che condividono una risorsa rispondente ai vincoli. La successiva comunicazione fra i client avviene propriamente peer to peer, con una o più connessioni dirette tra il peer che ha richiesto la risorsa e i peer che la distribuiscono. Se il vantaggio del modello centralizzato risiede nella semplicità e affidabilità del protocollo, il principale problema sta nella sua vulnerabilità alla censura. Infatti, sebbene la presenza di un server centrale lasci intravvedere dei limiti di scalabilità, numerose applicazioni hanno dimostrato nella pratica la possibilità di realizzare indici centralizzati di grandissime dimensioni, un esempio per tutti: i Web search engine. Napster, uno dei primi esempi di tale architettura, è in effetti incorso in una azione legale, e non in limiti tecnologici.

Un esempio di sistema peer to peer puramente decentralizzato è invece costituito dalla rete Gnutella. Gnutella costruisce

un overlay network, cioè una rete di nodi la cui topologia è indipendente dalla sottostante rete Internet, dotato di suoi propri meccanismi di routing, che permette agli utenti di condividere e distribuire risorse. Non esiste alcun controllo centralizzato dell'attività della rete, e ogni utente esegue una applicazione che funziona contemporaneamente come client e come server, chiamata *servent*.

La comunicazione tra i servent è regolata da un protocollo che definisce quattro tipi di messaggi:

ping una richiesta diretta a un certo host affinché si *annuci*;

pong un messaggio di risposta a un *ping*. Contiene l'indirizzo IP e la porta del mittente del messaggio e il numero e la dimensione dei file condivisi;

query una richiesta di risorse, contiene una search string, e indicazioni sui requisiti di bandwidth;

query hits un messaggi di risposta a una *query*, contiene indirizzo IP e porta ai quali connettersi per il download, informazioni di bandwidth e il numero di file che corrispondono alla query;

Un esempio di comunicazione in una rete peer to peer puramente decentralizzata è illustrato in figura 2.2.

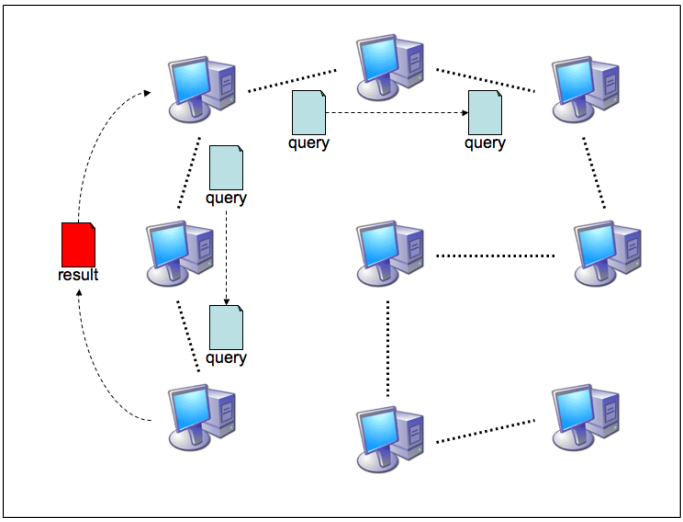

Figura 2.2: Organizzazione di una rete peer to peer puramente decentralizzata.

Dopo essersi connesso alla rete, ogni nodo invia un messaggio *ping* ad ognuno dei suoi vincini, cioè a tutti i nodi dei quali conosce direttamente l'indirizzo IP e la porta. Questi, a loro volta, rispondono con un messaggio *pong*, identificandosi, e propagano il ping ai propri vicini. Il discovery di una risorsa sulla rete avviene inoltrando un messaggio *query*. Tale messaggio vie-

ne propagato nella rete da un nodo all'altro, fino all'esaurimento di un time-to-live. Le eventuali risposte sono inoltrate al nodo che ha originato la query seguendo il cammino inverso. Quando un nodo riceve un messaggio *query hit*, che indica che la risorsa cercata è stata localizzata su un certo peer, esso stabilisce una connessione diretta per il download. La scalabilià del sistema è garantita dal time-to-live dei messaggi, che impone di fatto un *orizzonte*, oltre il quale i messaggi non possono propagarsi, evitando il collasso della rete.

In una via di mezzo si collocano i sistemi parzialmente centralizzati: essi impiegano il concetto di super peers, a cui viene temporaneamente affidato il compito di inoltrare le query originate in una determinata porzione della rete. I super peer sono eletti in base alle capacità di calcolo e bandwidht disponibile e permettono di migliorare le prestazioni complessive della rete, riducendone i tempi di risposta, senza però introdurre elementi centralizzati che possono essere soggetti a controllo o censura, o costituire un *single point of failure*. La struttura di una rete peer to peer parzialmente centralizzata è illustrata infigura 2.3

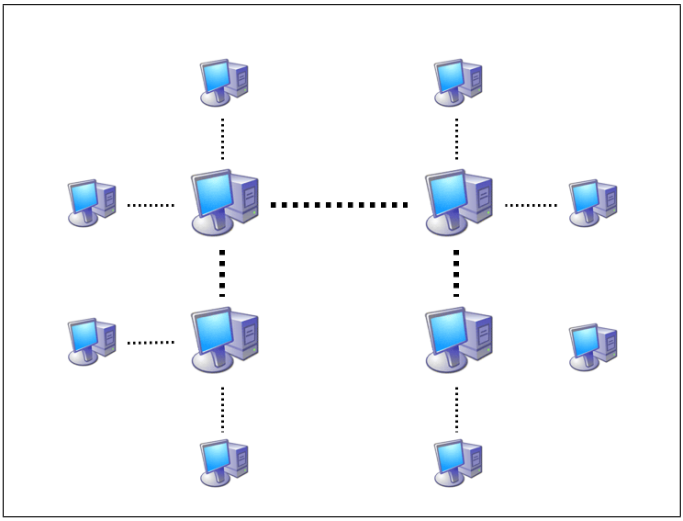

Figura 2.3: Organizzazione di una rete peer to peer parzialmente centralizzata.

2.2.2 Strutturate e Destrutturate

Con questa disitinzione si intende evidenziare l'esistenza o meno di una relazione tra la topologia della rete e la localizzazione di una risorsa sulla rete stessa. Nelle reti destrutturate non esiste alcuna relazione tra la posizione di una risorsa e la topologia della rete. La localizzazione di una risorsa in seguito a una query avviene con tecniche che spaziano dal flooding alla propagazione

in profondità o in ampiezza, seguendo la topologia della rete.

Nelle reti strutturate invece la posizione delle risorse è controllata e le risorse vengono memorizzate su nodi scelti in modo deterministico secondo un preciso algoritmo che fornisce un mapping tra i contenuti e i nodi, o piu' precisamentetra i rispettivi identificatori, sotto forma di tabelle di routing distribuite che permettono di instradare in modo efficiente le query verso nodi che memorizzano la risorsa cercata. L'esempio più significativo nel panorama delle reti peer to peer strutturate è costituito dalle Hash Table Distribuite, di cui nel capitolo 3 sono illustrati due esempi: Kademlia e Pastry.

3 Distributed Hash Tables

Una Distributed Hash Table (DHT) è una struttura dati distribuita usata nei più recenti sistemi peer to peer, che consente di memorizzare coppie $\langle chiave, valore \rangle$ in modo efficiente, affidabile e robusto. Poiché la primitiva di *lookup*, che consente di recuperare un valore memorizzato nota la chiave, ha complessità logaritmica rispetto al numero di nodi connessi alla rete, le DHT sono estremamente scalabili: è sufficiente un passo in più nelle operazioni di lookup per fare fronte a un raddoppio delle dimensioni della rete. Tuttavia, come conseguenza dell'algoritmo di routing usato per allocare i contenuti e instradare le query, le DHT non consentono di effettuare query su intervalli, limitandosi a una ricerca exact match, in cui cioè è noto a priori

l'identificatore della risorsa cercata (la chiave, appunto). In generale alle risorse viene assegnato un idetificatore, basato spesso sull'hash crittografico del loro contenuto, e successivamente vengono instradate verso uno o piu' nodi secondo una logica predeterminata e dipendente dall'identificatore stesso. La natura pseudo casuale dell'ID della risorsa garantisce un bilanciamento della distribuzione delle risorse tra i nodi ma al tempo stesso viene persa ogni informazion relativa all'ordinamento reciproco delle risorse.

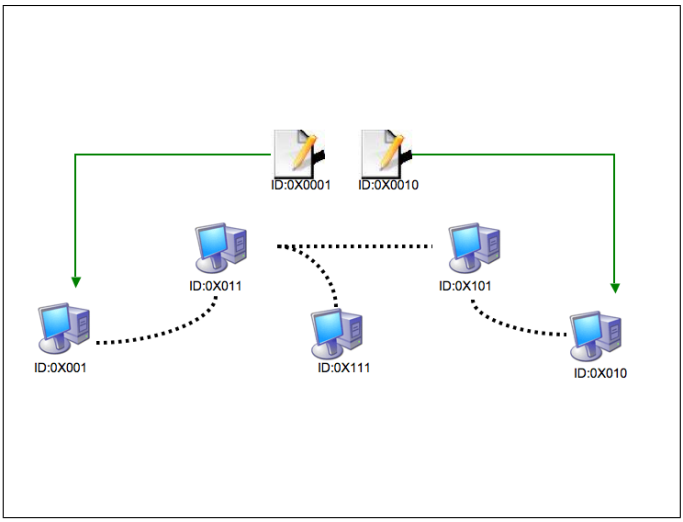

Figura 3.1: Routing in una Hash Table Distribuita.

Tra gli esempi più significativi di tale tecnologia si possono citare Chord [28], Tapestry [31], Can [26] e Kademlia [17].

Sebbene differrenti nei dettagli, queste tecnologie sono molto simili nella sostanza, e sono accomunate da alcune caratteristiche fondamentali:

1. Il sistema opera in modo completamente decentralizzato, non esiste una *autority* di coordinamento per il routing, l'assegnazione dei contenuti o degli indirizzi e in particolare non esiste un *single point of failure*; il sistema potrebbe continuare a funzionare anche in presenza di malfunzionamenti estesi o attacchi sistematici e massivi

2. Il sistema è scalabile, intendedo con ciò che il numero di nodi può crescere in modo virtualmente indefinito, e con esso il traffico, senza comprometterne il funzionamento, e il carico di lavoro viene ripartito equamente fra i nodi che costituiscono la rete

3. La rete è supposta fortemente dinamica; in ogni momento nuovi nodi possono aggiungersi ed altri possono disconnettersi senza che ciò abbia un impatto significativo sulle pre-

stazioni o sull'affidabilità, la coerenza dei dati è assicurata da un certo grado di *ridondanza*

Nel seguito sono descritte in dettaglio le architettura Kademlia, sulla quale si basa l'argomento del presente lavoro di tesi, e Pastry. Le altre non differiscono in modo significativo.

3.1 Kademlia

Kademlia [17] si distingue nel panorama dei sistemi peer to per basati su DHT per una serie di apprezzabili caratteristiche:

1. I nodi che partecipano alla rete scambiano un numero contenuto di messaggi di coordinamento, riducendo il sovraccarico di traffico necessario alle operazioni di sistema, le informazioni di routing e configurazione si diffondono in modo automatico come effetto collaterale delle normali operazioni di lookup

2. le query sono propagate in modo parallelo e asincrono, ciò che permette di operare in condizioni di elevata laten-

za o malfunzionamenti della rete evitando rallentamenti o ritardi dovuti all'attesa di timeout

3. L'algoritmo di routing usato in Kademlia si dimostra robusto alle principali tipologie di attacchi e la sua efficienza può essere provata nei casi tipici di funzionamento della rete Internet

Una risorsa memorizzata sulla rete Kademlia è identificata da una chiave a 160 bit, determinata in modo pseudo causale (ad esempio eseguendo l'algoritmo SHA-1 sul contenuto della risorsa stessa). Le chiavi sono opache, cioè non sono correlate in alcun modo con la risorsa per la quale sono state generate. Ad ogni nodo che partecipa alla rete Kademlia viene assegnato un *node ID*, un identificatore, scelto all'interno dello spazio di 160 bit delle chiavi. Una particolare coppia $\langle chiave, valore \rangle$ viene memorizzata su un nodo il cui *node ID* è *vicino* alla chiave data. La funzione XOR bitwise è usata come metrica della distanza all'interno dello spazio di indirizzamento a 160 bit delle chiavi: date due chiavi x e y interpretate come numeri interi

$$d(x, y) = x \oplus y$$

Si dimostra facilmente che la funzione XOR è una valida metrica di distanza, sebbene non euclidea:

$\forall x : d(x, x) = 0$

$\forall x, y : d(x, y) \geq 0$ sse $x \neq y$

$\forall x, y : d(x, y) = d(y, x)$

vale inoltre la disuguaglianza triangolare:

$\forall x, y, z : d(x, y) + d(y, z) \geq d(x, z)$

3.1.1 *k*-buckets

Come già accennato, ogni nodo è a propria volta identificato da un *node ID* a 160 bit che può quindi essere confrontato con le chiavi che identificano le risorse salvate. Ogni nodo mantiene una tabella di routing nella quale memorizza gli indirizzi di altri nodi nella forma di triple $\langle indirizzoIP, portaUDP, nodeID \rangle$. La tabella di routing di ogni nodo ha la forma di una lista di 160 elementi in cui, alla posizione i-esima, sono memorizzati gli indirizzi di nodi noti, la cui distanza dal nodo stesso è compresa tra 2^i e 2^{i+1}; ogni posizione nella lista è chiamata un k-bucket e le entry al suo interno sono ordinate per tenere conto di quali nodi hanno generato del traffico nel periodo recente (e hanno

dunque maggiore probabilità di essere ancora attivi). Ad esempio il k-bucket[0] continete l'indirizzo del nodo (se esiste ed è noto) che dista dal nodo dato per un solo bit, il meno significativo; il k-bucket[n] contiene potenzialmente 2^n indirizzi che distano dal nodo dato per gli n bit meno significativi e così via. Nella pratica esiste un limite k al numero di indirizzi che sono memorizzati in ogni k-bucket (in genere $k = 20$) determinato in modo empirico sulla base di considerazioni relative alla permanenza media online dei nodi. I k-bucket vengono popolati e aggiornati in modo continuo per mezzo delle informazioni contenute in tutti i messaggi che il nodo instrada nel corso del suo normale funzionamento.

3.1.2 Lookup

L'operazione di *lookup* di un nodo avviene in modo ricorsivo: come prima cosa l'esecutore ricerca nel proprio k-bucket più opportuno gli indirizzi di α nodi i cui *node ID* sono quanto più possibile vicini (secondo la metrca XOR) al nodo cercato e invia a questi una richiesta FIND_NODE.

Ognuno dei nodi interpellati risponde prelevando dalla propria

tabella di routing i k indirizzi più vicini. Il lookup avviene in modo parallelo e asincrono, se uno dei nodi interpellati non risponde il nodo che ha iniziato il lookup può decidere di interpellarne un altro e/o procedere con il passo ricorsivo in base alle altre risposte eventualmente ottenute. Le risposte ricevute sono usate per ripopolare la tabella di routing del nodo e proseguire nell'operazione di lookup che, gradualmente converge verso il nodo cercato. Il numero di passi necessario per completare una operazione di lookup è in media *O(log N)* per una rete costituita da N nodi (la prova di questo risultato è data in [17])

3.2 Pastry

Pastry [27] è un'architettura peer to peer completamente decentralizzata, scalabile, affidabile e robusta; ogni nodo nella rete è identificato da un codice numerico unico (nodeID). L'algoritmo i routing alla base di Pastry è in grado, dati un messaggio, e un idetificatore numerico, di instradarlo verso il nodo il cui nodeID è numericamente più *vicino* all'identificatore dato, i nodeID sono identificatori di 128 bit, e rappreentano la posizione di ogni

nodo all'interno di uno spazio di indirizzamento circolare che va da 0 a 2^{128}.

ll numero di salti necessario per inoltrare un mesaggio al corretto destinatario è log(N), in cui N è il numero di nodi nella rete Pastry.

Inoltre l'assegnazione dei nodeID aviene in modo casuale, e pertanto una famiglia di nodi con nodeID simile, sarà generalmente distribuita in modo casuale, rispetto alla collocazione geografica, alla proprietà, ecc...

per questa ragione l'algoritmo di routing di Pastry seleziona, ad ogni salto, fra i potenziali candidati, quello più vicino al nodo presso il quale il messaggio ha avuto origine, secondo una metrica di prossimità. Quando un nodo riceve un messaggio cerca di instradarlo verso il nodo il cui nodeID ha in comune con la chiave del messaggio un prefisso la cui lunghezza sia maggiore rispetto a quella del prefisso che il nodo corrente condivide con la stessa chiave. Se tale nodo non esiste o non è noto, il messaggio viene inoltrato verso il nodo identificato da un nodeID che condivide con la chiave de messaggio un prefisso di lunghezza uguale al nodo corrente, ma è numericamente più vicino alla

chiave del messaggio Chiaramente ogni nodo deve conservare una tabella di routing, in cui sono memorizzati i riferimenti ai nodi conosciuti. Come si vede l'algoritmo di routing è nella sostanza molto simile all'algoritmo di Kademlia, e se ne differenzia fondamentalmente per la metrica di prossimità usata.

4 Stato dell'Arte

Come evidenziato nei capitoli precedenti, le reti peer to peer offrono un design efficiente e affidabile per lo sviluppo di applicazioni distribuite. Nel corso delgi ultimi anni,le architetture peer to peer si sono evolute e hanno superato molte delle limitazioni che le affliggevano in origine. Le Hash Table Distribuite si stanno oggi affermando come standard per le provate doti di efficienza, affidabilità e scalabilità che le caratterizzano, e, nonstante offrano una interfaccia minimale, basata in genere sulle sole primitive *put* e *get*, rispettivamente per la memorizzazione e il recupero di un dato, costituiscono l'infrastruttura portante di un numero sempre crescente di applicazioni.

Le caratteristiche desiderabili di una infrastruttura peer to peer

variano in base alla classe di applicazioni che su di essa si desidera creare, tuttavia in generale si dà notevole importanza alle seguenti:

sicurezza: la garanzia che la rete preservi l'integrità dei dati in essa memorizzati, anche, o in particolare, contro interventi malevoli;

privacy: la rete deve proteggere i dati in essa memorizzati contro accessi non autorizzati;

persistenza: garantire la disponibilità dei dati quando questi vengono richiesti. Questo comporta la capacità della rete di resistere a malfunzionamenti locali, e di adattarsi agli avvicendanti dei nodi di calcolo che la compongono;

scalabilità: requisito essenziale è che le prestazioni della rete siano (quasi) indipendenti dalla sua dimensione;

load balancing: o *fairness*, garantisce che tutti gli utenti abbiano uguale accesso alla rete e che il contributo ad essi richiesto per partecipare alla rete stessa sia equamente di-

stribuito, evitando sovraccarichi su alcuni nodi, colli di bottiglia e single point of failure;

self maintenance : l'attività della rete non deve richiere un coordinamento centralizzato, appunto per evitare che il coordinatore possa essere oggetto di censura o costituisca un collo di bottiglia per il funionamento della rete o un punto centrale di rottura;

Se privacy e sicurezza possono essere affrontate ad un livello applicativo, ad esempio per mezzo di tecniche di crittografia e firma digitale, persistenza, load-balancing, scalabilità e self-maintenance richiedono un accurato design, già al livello dei protocolli e degli algoritmi di routing.
Come si è visto nel capitolo 3 le Hash Table Distribuite sono state progettate proprio per rispondere a questi requisiti. Tuttavia come è noto le DHT non supportano ricerche su intervalli, ma sono limitate a ricerche esatte (in cui cioè sia nota a priori la chiave di ricerca), escludendo di fatto una ampia gamma di applicazioni.
Il presente lavoro è mirato a superare tale limitazione.

Di seguito si descrivono i principali approcci per l'esecuzione di range queries su reti P2P, di ognuno si evidenziano i vantaggi e i punti deboli.

4.1 LP-DHT

Con il termine *Locality Preserving DHT* si indica una famiglia di approcci tesi a supportare l'esecuzione di range query in reti peer to peer strutturate modificando l'algoritmo di routing in modo tale che documenti identificati da chiavi *vicine* vengano memorizzate su nodi *vicini*. Esempi di tali architetture includono SkipNet [12], SkipGraphs [2], OP-Chord [29], PIER [14], Mercury [3]. L'idea di base che accomuna questi approcci parte dalla constatazione che l'algoritmo di routing delle tradizionali DHT distrugge la località dei dati memorizzati calcolando una funzione di hash crittografico della chiave primaria.
Le LP-DHT utilizzano invece la chiave stessa come identificativo della risorsa da memorizzare, ad esempio il nome del file in applicazioni di file sharing, ottenendo come risultato che risorse identificate da chiavi *simili* risultano memorizzate su nodi della

rete i cui identificattori sono numericamente vicini. In questa prospettiva, la gestione di range query consiste in:

1. trovare il nodo responsabile dell'inizio del range

2. seguendo i salti successivi arrivare al nodo responsabile della fine del range

i dati del range sono quelli relativi ai contenuti restituiti dai nodi attraversati.
In un overlay formato da q nodi, le complessità è pari a $O(logN+q)$: $O(logN)$ salti per la fase (1), più altri q salti per la fase (2). Occorre tuttavia tenere presente che l'uso di una funzione di hash crittografico nell'assegnazione dell'identificativo delle risorse ha il preciso scopo di garantire una equidistribuzione delle risorse stesse sui nodi della rete. Per contro, adottando uno schema di identificatori non casuali per le risorse si perde la caratterstica di load-balancing, che contraddistingue le DHT e ne ha favorito la diffusione.

4.2 Prefix Hash Trees

Un Prefix Hash Tree (PHT) è una struttura dati distribuita che può essere agevolmente memorizzata su una Distributed Hash Table ed essere usata come indice di ricerca per gli elementi contenuti nella DHT stessa [4, 25]. Nella forma più semplice supporta l'esecuzione di ricerce in intervalli unidimensionali, ma può essere agevolmente applicato al caso di ricerche bidimensionali (ad esempio per sistemi informativi geografici). È anche possibile eseguire heap queries (ricerca del massimo/minimo) e proximity queries (trovare l'elemento più vicino ad un elemento dato).

Un PHT è un albero binario in cui ogni nodo corrisponde a un determinato prefisso del dominio di dati che si vuole indicizzare. Le chiavi che identificano i dati devono essere espresse come stringhe binarie di lunghezza fissa e sono memorizzate nei nodi foglia del PHT. La struttura dati risultante viene memorizzata in una DHT, garantendo così accesso *diretto* al nodo dell'PHT che memorizza un certo prefisso. L'estensione al caso di domini bidimensionali avviene mediante un processo di linearizzazione.

Ad esempio per memorizzare coordinate geografiche, i valori float di ogni coordinata sono dapprima normalizzati in un intero di N bit e in un secondo momento sono combinati in un unica chiave di 2N bit. Esistono differenti tecniche di linearizzazione, una panoramica di tali tecniche è presentata in appendice A.
Ogni livello del PHT divide il dominio di ricerca in due, alternativamente rispetto a latitudine e longitudine. Le primitive disponibili su un PHT sono lookup, range_query, insert e delete.

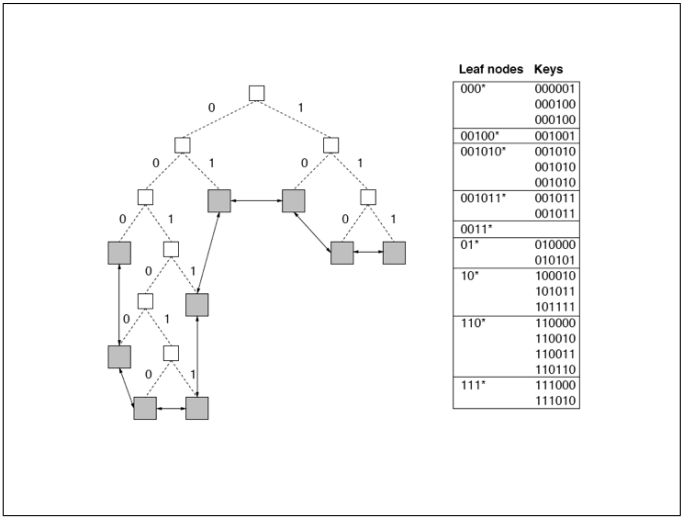

Figura 4.1: Prefix Hash Tree.

Data una chiave K l'operazione di *lookup* trova l'unico nodo foglia la cui chiave è un prefisso di K. Dato un prefisso P l'operazione di *lookup* trova il nodo contenente il più lungo prefisso comume a P. L'operazione di lookup si implementa efficientemente con una ricerca binaria dei D-1 possibili prefissi di una chiave di lunghezza D. iniziando con un prefisso di lunghezza D/2 e tentando via via prefissi più lunghi/più corti se il lookup trova un nodo interno/non trova alcun nodo.

Range_query: per un PHT unidimensionale e date due chiavi $K1 \leq K2$ una query sull'intervallo [*K1*,*K2*] viene risolta individuando il nodo che contiene il più lungo prefisso comune tra le chiavi *K1* e *K2*. Una query bidimensionale viene eseguita calcolando il prefisso linearizzato che racchiude la regione oggetto della query. l'inserimento e la rimozione di una chiave K nel PHT (primitive *insert* e *delete*) richiedono un operazione di lookup *leaf(K)*. l'operazioni di inserimento può richiedere il partizionamento di un nodo eccessivamente carico che verrà trasformato in un nodo interno e la creazione di due nodi foglia sui quali ripartire i dati.

In modo simile la rimozione di una chiave può risultare nel mer-

ging di due nodi foglia i cui dati saranno assorbiti dal nodo *parent* (che memorizza il prefisso comune a tutti i dati presenti nei figli). Le prestazioni di un PHT dipendono dalla sua capacità di offrire un accesso diretto ai nodi intermedi dell'albero (senza passare dal nodo radice, che dunque non rappresenta un collo di bottiglia), ciò avviene grazie al fatto che la struttura è memorizzata all'interno di un DHT, la cui topologia è del tutto indipendente da quella della struttura di ricerca.

Sebbene estremamente efficiente, l'algoritmo di lookup su un Prefix Hash Tree, mostra alcuni punti deboli che lo rendono inadatto a una applicazione su vasta scala: le chiavi, di lunghezza fissa sono organizzate in un albero binario e il lookup di una chiave inizia appunto ricercando il più lungo prefisso noto memorizzato nell'albero. Per far si che la radice dell'albero non venga inondata di richieste di lookup l'algoritmo inizia la sua ricerca da una posizione intermedia (ciò è reso possibile dall'indirizzamento diretto della sottostante DHT). In teoria questo dovrebbe ripartire le query su un numero sufficiente di nodi evitando di sovraccaricare il nodo radice. L'assunto implicito è che le chiavi siano approssimativamente equidistribuite nello spazio

di N bit ad esse dedicato, ma tale assunto non è sempre vero. Ad esempio nel caso di una applicazione geografica, in cui ogni risorsa viene indicizzata rispetto alle proprie coordinte (longitudine e latitudine) è lecito aspettarsi una concentrazione di entry in determinate aree (in corrispondenza delle città). Il nodo della rete che si trovasse a memorizare il prefisso di lunghezza N/2 relativo a un'area particolarmente densa di informazione dovrebbe partecipare a tutte le query relative a quell'area.
Un secondo punto debole è rappresentato dalle operazioni di split e merge di un nodo, operate rispettivamente in caso in inserimento e rimozione di dati. Il mancato completamento di tali operazioni di manutenzione (ad esempio per una momentanea indisponibilità della connessione, porterebbe la struttara dati in uno stato di incosistenza, complesso da recuperare.

4.3 P-Trees

Un P-tree [6] è una struttura di ricerca distribuita su rete P2P, ottenuta frazionando e distribuendo sui peer i rami di un albero binario di ricerca, in modo che ogni nodo conosca il cammi-

no che lo connette alla radice dell'albero. La struttura di dati mantenuta da ogni nodo ha la forma di una array *p.node[i][j]* dove:

- $i < maxdepth$ (la massima profondità dell'albero per il nodo p)
- $j < numentries$ (un valore prefissato, di solito = 2)

Ogni elemento dell'array è una coppia $< value, peer >$ che punta al *peer* che conserva il valore *value*. Se consideriamo i peer organizzati in un anello ordinato secondo le chiavi *value* possiamo definire *succ(p)* e *pred(p)* come i nodi che stanno rispettivamente a destra e sinistra del nodo p in un attraversamento orario dell'anello. Per i nodi valgono le seguenti proprietà:

1. (numero di entries per nodo) ogni nodo interno ha un numero di entries compreso tra depth e 2depth. Il nodo radice ha un numero di entries compreso tra 2 e 2depth;
2. la prima entry di ogni nodo in un peer punta al peer stesso: *p.node[i][0] =(p.value,p) i* in [0,p.maxdepth];
3. coverage: ogni chiave di ricerca è indicizzata nel P-tree;

4. separation: due entries adiacenti al livello i hanno almeno d entries non sovrapposte al livello *i-1*

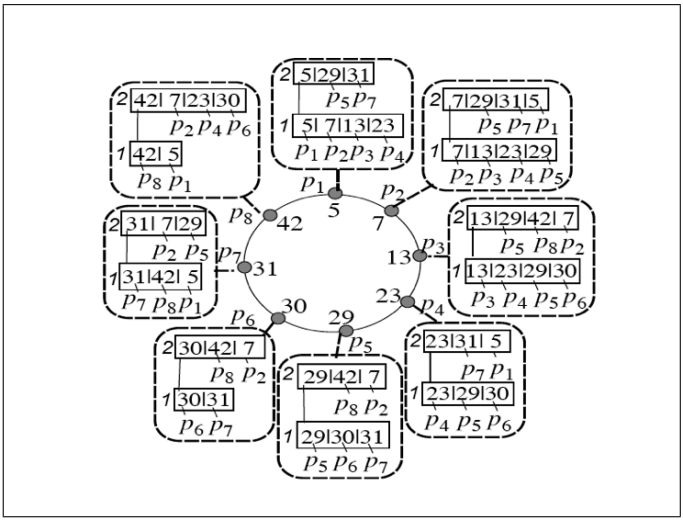

Figura 4.2: P-Tree.

Mantenere la consistenza di un P-tree richiede l'esecuzione periodica di un processo diagnostico consistente di due attività:

ping controlla se un peer è stato cancellato o è in failure e in caso affermativo rimuove la relativa entry dal P-tree locale. Controlla anche se le entries memorizate sono consistenti rispetto alle proprietà di coverage e separation. Le inco-

sistenze eventualmente rilevate sono corrette dal processo di *stabilization*;

stabilization viene eseguito periodicamente su ogni nodo e ripara le incosistenze rilevate dal processo *ping*.

Le operazioni possibili in un P-Tree sono inserimento, rimozione e ricerca. Una query ha origine in un nodo della rete P2P e ha come chiavi un range *[lb, ub]*. L'algoritmo di ricerca di ogni nodo seleziona il più lontano peer conosciuto che converva un valore non superiore a *lb* e inoltra ad esso la query. Quando la query raggiunge un nodo foglia, l'anello viene attraversato fino al raggiungimento di una valore superiore a *ub*; un messaggio *SearchDoneMessage* viene inviato al peer che ha iniziato la query quando ciò si verifica. In una rete stabile di N peers con un P-Tree consistente di ordine d, una query che restituisce m risultati richiede $O(m + log_d(N))$ messaggi.
Le operazioni di *ping* e *stabilization* sono il principale problema legato ai P-Trees. Infatti tali operazioni, oltre a rappresentare un costante sovraccarico sulla rete la cui entità cresce linearmente con la dimensione della rete stessa, possono di per se portare

a incosistenze nella struttura dati. Occorre infatti tenere presente che tali operazioni devono essere eseguite in un ambiente inerentemente dinamico, in cui in ogni momento uno o più nodi possono incorrere in problemi tecnici che li rendono momentanemante non disponibili. Inoltre la struttura sostanzialmente ad albero tende a sovraccaricare i nodi della rete più prossimi alla radice dell'albero.

4.4 Range Guard

Range Guard è un'architettura che permette di eseguire query su intervally estendendo le architetture peer to peer già esistenti senza introdurre alcun nuovo overlay, ma partendo piuttosto dalla considerazione che, se nelle tradizionali architetture DHT non viene preservata la caratteristica di *vicinanza fisica* (infatti due documenti molto simili hanno chiavi completamente diverse), a causa della casualità data dalla funzione di hash, quando si ha a che fare con un accesso sequenziale, sarebbe invece auspicabile preservare la vicinanza. Negli overlay per le range query si utilizza il contenuto per localizzare le risorse, piuttosto che la

chiave data dalla funzione di hash.

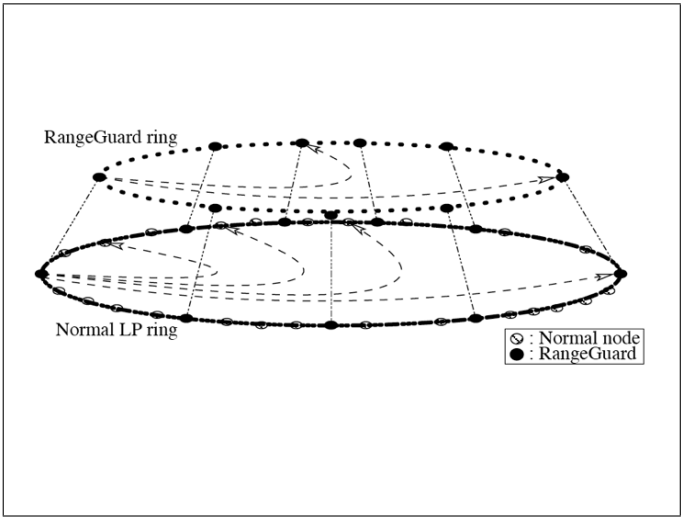

Figura 4.3: Range Guard.

Range Guard supporta le query su intervalli per mezzo di una architettura a più livelli, a sua volta basata su LP-DHT. Lo scopo di Range Guard consiste nel realizzare un indice distribuito delle risorse, la cui gestione è affidata ai nodi più performanti della rete. Tuttavia valgono le considerazioni già fatte a proposito delle LP-DHT.

4.5 Range Addressable Networks

Con Range Addressable Network [16] (RAN) si intende indicare una struttura dati distribuita che permette di effettuare range queries su un overlay P2P esistente. L'idea di base è che le range queries eseguite sulla rete seguano degli schemi tipici e prevedibili, e possano pertanto essere agevolate da un sistema di caching. Un peer inoltra in rete una query relativa a un certo range di valori. Il sistema tenta dapprima di localizzare un peer che memorizza tutti i valori che ricadono nei vincoli della query. Se ciò non è possibile la query è risolta in modo diretto. Il risultato viene succesivamente indicizzato rispetto al range in cui ricade e memorizzato complessivamente da un peer (caching) che da quel momento sarà responsabile per quel particolare intervallo.

Ogni nodo conserva anche informazioni sui propri vicini, in particolare un peer mantiene un collegamento con i nodi che memorizzano intervalli che contengono o sono contenuti nel proprio intervallo di pertinenza. In pratica quindi un RAN è un *interval tree* distribuito in cui ogni peer è responsabile di un dato nodo

dell'albero. Una query può avere origine ad ogni nodo e risale o discende l'albero, muovendosi lungo la rete, fino a raggiungere il peer che indicizza l'intervallo più prossimo all'intervallo cercato. Gli intervalli indicizzati ad ogni nodo non sono completamente disgiunti ma sono previste delle sovrapposizioni che creano una certa ridondanza allo scopo di garantire la robustezza del sistema. Anche in questo caso però non si hanno garanzie sulla equidistribuzione del carico fra i nodi della rete. inoltre il principio secono cui i risultati delle query possano essere memorizzati e riutilizzati in query successive dipende fortemente dal tipo di applicazione e dalla natura dei dati memorizzati. Non è peraltro chiaro in che modo il peer responsabile di un particolare range venga aggiornato sull'inserimento di nuovi valori che ricadono all'interno dell'intervallo di pertinenza.

5 Skip Lists

Una skip list [24] è una struttura dati che permette ricerche efficienti in collezioni ordinate di dati. Si distingue dagli alberi bilanciati per la semplicità di implementazione e la relativa facilità nell'ottimizzazione degli algoritmi. Una skip list è formata da una lista concatenata, alla quale, per accelerare le ricerche, vengono aggiunti dei puntatori ridondanti che permettono di attraversare la lista saltandone ampie porzioni, raggiungendo l'elemento cercato (se esiste) molto rapidamente.

La figura 5 illustra la struttura di una skip list: alla lista concatenata del livello più basso in cui i puntatori legano ogni elemento al successivo, si aggiungono altri livelli che consentono una ricerca veloce nella lista. Un elemento ogni due è dotato di

un puntatore due posizioni in avanti, uno ogni quattro è dotato di puntatore quattro posizioni in avanti e così via, un elemento ogni N è dotato di un puntatore che consente di saltare i 2^{N-1} elementi successivi.

La ricerca di un elemento nella skip list inizia dalla testa della lista e procede iterativamente effettuando ad ogni passo il salto più lungo possibile, senza oltrepassare l'elemento cercato. Come è facile constatare la complessità dell'algoritmo è O(log(N). Tuttavia l'inserimento e la rimozione di un elemento richiede complesse operazioni di manutenzione sui puntatori, allo scopo di mantenere coerente la struttura a livelli e non compromettere l'efficienza delle operazioni i ricerca. La strategia che si adotta per semplificare le operazioni di manutenzione mantendo inalterate le caratteristiche di efficienza e semplicità di implementazione coniste nell'affidare il bilanciamento della struttura a un algoritmo di inserimento *randomized*. Si rinuncia all'obiettivo di costruire un skip list perfettamente simmetrica, in cui cioè esattamente un elemento ogni N appartengono alla lista di livello N, in favore di una scelta probabilistica: ogni nuovo elemento viene aggiunto alla lista di livello N con probabilita $1/2^{N-1}$. Dunque

tutti gli elementi sono aggiunti alla lista concatenata di livello 1, solo la metà alla lista di livello 2, e così via. Le skip list così modificate conservano inalterata la loro efficienza complessiva, sebbene localmente possano presentarsi a volte sbilanciate, con i conseguenti rallentamenti. Come già accennato, il principale punto di forza delle skip list rispetto agli alberi binari, consiste nella semplicità di implementazione, dato che gli algoritmi di inserimento e ricerca si prestano naturalmente ad essere implementati con costrutti iterativi, rispetto agli alberi che sono strutture inerentemente ricorsive, essi si prestano meglio ad un lavoro di ottimizzazione, che invece sugli alberi è in genere meno immediato. Tuttavia, tradizionalmente, anche per il fatto che gli alberi si prestano perfettamente a rappresentare diversi pattern fondamentali dell'informatica, essi si sono affermati sia nella teoria che nella pratica, come struttura dati preferenziale per la rappresentazione di collezioni ordinate.

Le skip list, d'altra parte, presentano numerose caratteristiche che le rendono preferibili agli alberi per la realizzazione di una struttura dati distribuita:

- La skip list non ha una *radice* o un entry preferenziale

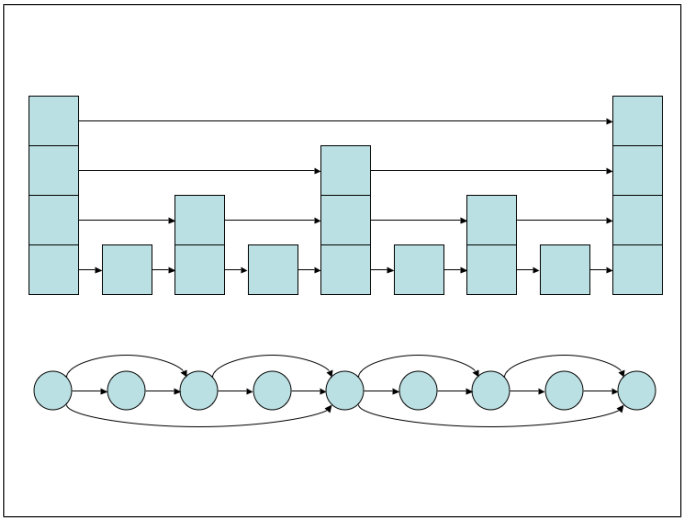

Figura 5.1: Struttura di una skip list.

dalla quale iniziare un'operazione di ricerca. Sebbene sia consuetudine descrivere le operazioni di ricerca a partire da un entry di testa della lista, niente vieta di iniziare da un nodo qualsiasi, procedendo in avanti o a ritroso, secondo necessità, sempre che le liste siano doppiamente concatenate. Per contro tutte le operazioni di ricerca in un albero binario coinvolgono la radice dell'albero, che in un ambiente distribuito e concorrente si trova così a sostenere

tutto il carico delle operazioni compiute sulla struttura dati, e costituisce un punto di rottura in caso di guasto.

- ogni entry della skip list rappresenta una entità indipendente dalle altre e non ha un ruolo privilegiato o determinante nella gesione delle operazioni di ricerca o inserimento. Al contrario i nodi interni di un albero hanno funzione di guida nella ricerca e la loro importanza è tanto maggiore quanto più vicino essi si trovano alla radice. Un aggiornamento inconsistente della struttura di un albero rende indisponibili ampie porzioni dell'albero stesso, mentre la stessa incosistenza nell'aggiornamento di una skip list viene assorbita in modo più naturale, come sarà evidenziato nel seguito.

- L'efficienza delle operazioni di ricerca in un albero dipende dal bilanciamento della struttura, ma l'operazione di bilanciamento non è facilmente implementabile in un ambiente distribuito (anche quando possibile) e pone seri problemi di consistenza in caso di accesso concorrente; le skip list non richiedono operazioni di bilanciamento in quan-

to l'equilibrio della struttura dati è garantito dalla natura probabilistica dell'assegnamento dei puntatori.

- Le skip list si dimostrano particolarmente robuste, anche se ospitate in un ambiente distribuito estremamente inaffidabile. In particolare la rottura o il malfunzionamnto di un nodo di memorizzazione, sebbene possa causare l'indisponibilità di alcuni elementi della lista, non compromette la coerenza della struttura nel suo complesso.

Queste ragioni suggeriscono di utilizzare le skip list come punto di partenza per la progettazione di una struttura dati distribuita capace di memorizzare collezioni ordinate di dati e permettere l'esecuzione di query su intervalli. Nel seguito si descrive dettagliatamente come le skip list sono state adattate al caso specifico, gli algoritmi e le strutture interessate e si espongono le principali considerazioni sull'efficienza dell'approccio proposto e misurazioni sperimentali effettuate su una implementazione dimostrativa. Occorre qui precisare che esistono in letteratura vari adattamenti delle Skip List al problema del calcolo distribuito, Gli skip graphs [2] sono una struttura da

distribuita ispirata alle skip list. La localizzazione delle risorse e l'inserimento/cancellazione dinamico dei nodi richiede tempo logaritmico. Ogni nodo richiede spazio logaritmico per conservare le informazioni sui vicini. Non si usano algoritmi di hashing per determinare le chiavi delle risorse. Ogni nodo appartiene a più liste concatenate. Il livello 0 contiene tutti i nodi in sequenza. Un membership vector *m(x)* tiene il controllo delle liste ed è in grado di determinare a quale lista appartiene il nodo x. Si può immaginare *m(x)* come una parola casuale infinita su un alfabeto fissato. Ogni lista concatenata nello skip graph viene etichettata da una qualche parola finita w, e un nodo x è contenuto nella lista etichettata da w se e solo se w è un prefisso di *m(x)*. L'implementazione parallela di skip lists, su macchine multiprocessore a memoria condivisa è discussa in [23].

6 Skip List Distribuite

L'implementazione distribuita della skip list sfrutta le proprietà caratteristiche delle skip list tradizionali e delle hash table distribuite per raggiungere gli obiettivi gia citati nel capitolo 4:

scalabilità: Le caratteristiche del sistema si mantengono indipendenti dal numero di nodi o di documenti memorizzati. Un aumento, anche consistente, delle dimensioni della rete (cioè del numero di nodi di calcolo su cui la struttura viene distribuita) e della quantità di dati memorizzati devono avere un impatto minimo sulle prestazioni degli algoritmi.

fault tolerance: La struttura dati conserva la propria integrità anche in caso di malfunzionamenti di uno o più nodi. In particolare la skip list distribuita eredita le caratteristiche

di fault tolerance della sottostante DHT, e si mantiene coerente anche in caso di frequenti avvicendamenti nella rete: l'instabilità della rete (in ogni momento un nodo può disconnettersi dalla rete e nuovi nodi possono sostituirlo) è accettata come una sua caratteristica intrinseca.

self maintenance: In un ambiente distribuito e fortemente dinamico, come quello descritto, le operazioni di manutenzione della struttura dati rappresentano un rischio inaccettabile. Qualora una operazione di manutenzione dovesse fallire avremmo un degrado delle prestazioni nel caso più favorevole, una struttura dati incosistente nel caso peggiore. Pertanto l'algoritmo di inserimento deve essere progettato in modo da non sbilanciare la struttura dati, e da non poter mai lasciare la struttura stessa in uno stato incosistente nel caso in cui fosse impossibile completare l'operazione.

load balancing: Nessun nodo deve avere un ruolo privilegiato nell'esecuzione degli algoritmi o nella memorizzazione dei dati. Tanto le richieste di elaborazione, quanto l'impegno

di memorizzazione devono essere equidistribuiti su tutti i nodi della rete. Nel determinare un criterio di equità per la suddivsione del carico di lavoro si possono prendere in considerazione numerosi fattori, ad esempio la capacità di calcolo e memorizzazione di ogni nodo, il tempo di latenza della connessione, il fatto che il processore che esegue il nodo sia dedicato esclusivamente a tale operazione o meno, ecc. Nel seguito si considerano i seguenti criteri: (i) per ogni operazione di inserimento ogni nodo è candidato a gestire il dato memorizzato nella struttura e ha uguale probabilità di essere selezionato per la memorizzazione del dato e/o dei relativi puntatori; (ii) per ogni operazione di ricerca ogni nodo ha uguale probailità di essere coinvolto nell'esecuzione fornendo i dati e/o i puntatori che esso ha memorizzato.

Descriviamo ora brevemente il funzionamento della skip list distribuita; gli algoritmi e le strutture dati coinvolte saranno descritti nel dettaglio nel paragrafo 6.1.

La skip list viene memorizzata sotto forma di nodi con puntatori. I puntatori non legano necessariamente ogni elemento

al successivo, ma hanno piuttosto il significato di una relazione d'ordine lineare: l'esistenza di un puntatore dall'elemento A all'elemento B va interpretata come $A < B$. Ad esempio, data la lista in figura 6:

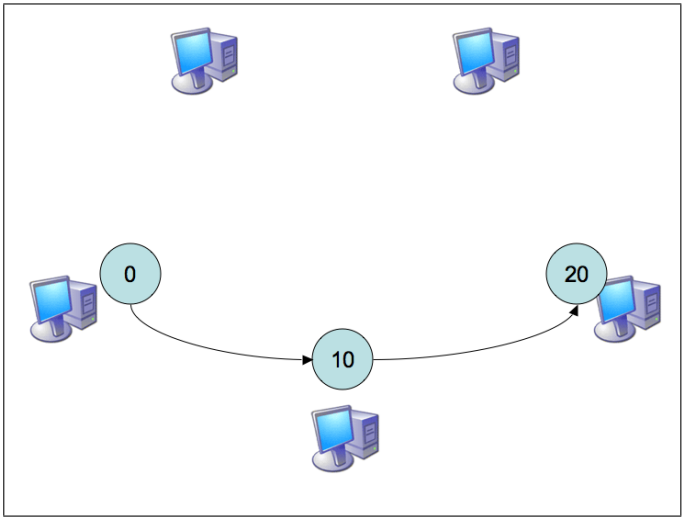

Figura 6.1: Distributed skip list.

L'algoritmo di inserimento memorizza nella DSL gli elementi e i puntatori relativi. L'inserimento di un nuovo elemento nella lista richiede una operazione di ricerca nella DSL per individuare le entry più vicine: se l'elemento è già presente nella DSL, non

viene eseguita alcuna ulteriore operazione. In caso contrario il nuovo elemento viene inserito nella posizione che gli compete memorizzando nella DSL i relativi puntatori. Si consideri la figura 6:

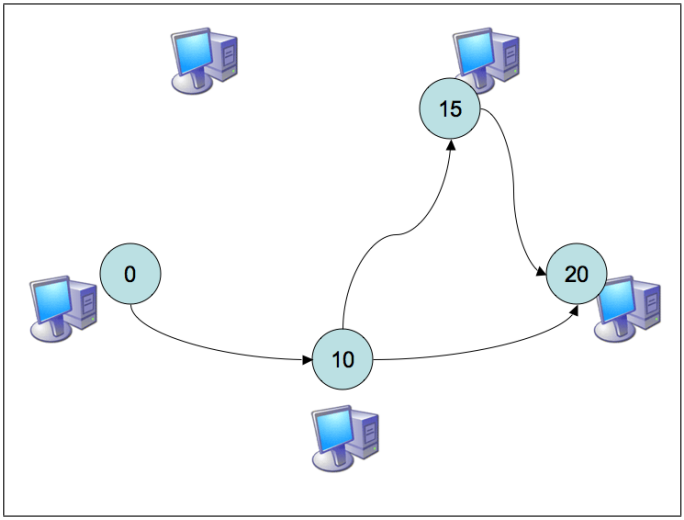

Figura 6.2: Inserimento nella DSL.

L'elemento etichettato 15 viene inserito tra 10 e 20 e due nuovi puntatori sono memorizzati nella DSL: $10 \mapsto 15$ e $15 \mapsto 20$. Il puntatore $10 \mapsto 20$ rimane tuttavia valido, coerentemente con il fatto che un nuovo inserimento non invalida la relazione $10 < 20$.

In analogia con le tradizionali skip list questo puntatore *allungato* permetterà l'esecuzione veloce di operazioni di ricerca, dato che consente di saltare ampie porzioni della lista (in rapporto alla popolazione).

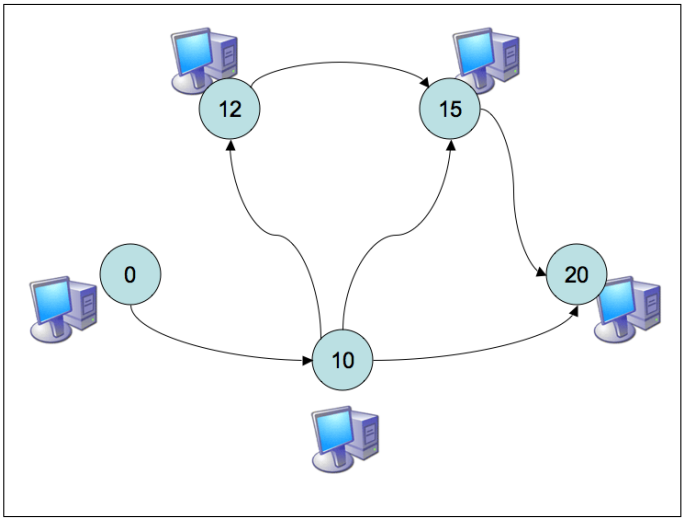

Figura 6.3: Inserimento nella DSL.

Infine si consideri l'operazione di inserimento rappresentata in figura 6: vengono memorizzati i nuovi puntatori $10 \mapsto 12$ e $12 \mapsto 15$ e i puntatori $10 \mapsto 20$ e $10 \mapsto 15$ vengono implicitamente *promossi* a puntatori di livello più alto. È importante osserva-

re che le operazioni di inserimento non comportano passaggi di manutenzione o ribilanciamento della struttura. In effetti non è nemmeno necessario cancellare o alterare i puntatori esistenti, il che rende la skip list distribuita particolarmente robusta quando eseguita in un ambiente con alte probabilità di errori, come la rete Internet. Una operazioni di inserimento può fallire o rimanere incompleta, ma tale eventualità non affligge la struttura nel suo complesso, rimanendo localizzata al valore che ha generato l'errore. Naturalmente se si desidera ottenere le caratteristiche di scalabilità, fault tolerance, self balancing, e self maintenance è necessario ricorrere a strategie più evolute della semplice idea delineata sopra: nei seguenti paragrafi sono descritte le operazioni primitive eseguibili sulla skip list distribuita e i relativi algoritmi.

6.1 Algoritmi e Strutture Dati

6.1.1 Strutture Dati

Idealmente ad ogni entry della DSL dovremmo associare una struttura di riferimento da usare nelle successive operazioni di

ricerca. Tale struttura deve memorizzare i puntatori che legano l'elemento ad altri elementi nella DSL, oltre a un riferimento al dato che si desidera memorizzare. Per ogni struttura viene calcolata una chiave (quasi) unica che la identifica univocamente all'interno della sottostante DHT, tale chiave deve essere ricavata in funzione del dato da memorizzare o di una combinazione dei suoi attributi, in modo da rispettare l'ordinamento relativo dei dati. Un esempio di tale struttura di riferimento è il seguente:

```
struct item_struct {
     byte[] key
     item[] forward_pointers
     item[] backward_pointers
     byte[] value
}
```

Come si può notare, due liste, rispetivamente di puntatori in avanti e di puntatori all'indietro, legano gli elementi della DSL in una lista doppiamente concatenata e *stratificata*. L'immediato successore di ogni elemento è il minore degli elementi puntati dall'array `forward_pointers`. Specularmente l'immediato predecessore di un elemento è il maggiore degli elementi puntati

dall'array `backward_pointers`.

Nella pratica, l'implementazione di tale struttura si avvale della capacità della DHT di memorizzare più valori sotto la stessa chiave, e pertanto essa viene memorizzate come collezione di coppie:

```
key = forward_pointer[1]
key = forward_pointer[2]
...
key = forward_pointer[n]

key = backward_pointer[1]
key = backward_pointer[2]
...
key = backward_pointer[n]

key=value
```

Ciò permette anche di aggiungere nuovi puntatori senza dover modificare una struttura già memorizzata.

Facendo riferimento alla DSL rappresentata in figura 6 i dati memorizzati sono:

```
struct item_struct {
     key = "item_0"
     forward_pointers = ["item_10"]
     value = "0"
} item_0
```

```
struct item_struct {
     key = "item_10"
     forward_pointers = ["item_20"]
     backward_pointers = ["item_0"]
     value = "10"
} item_10

struct item_struct {
     key = "item_20"
     backward_pointers = ["item_10"]
     value = "20"
} item_20
```

In seguito all'inserimento dell'elemento con etichetta '15' le strutture vengono modificate come segue:

```
struct item_struct {
     key = "item_0"
     forward_pointers = ["item_10"]
     value = "0"
} item_0

struct item_struct {
     key = "item_10"
     forward_pointers = ["item_20" ; "item_15"]
     backward_pointers = ["item_0"]
     value = "10"
} item_10

struct item_struct {
     key = "item_15"
```

```
     forward_pointers = ["item_20"]
     backward_pointers = ["item_10"]
     value = "15"
} item_15

struct item_struct {
     key = "item_20"
     backward_pointers = ["item_10" ; "item_15"]
     value = "20"
} item_20
```

Per ogni nuovo inserimento vengono memorizzati un numero costante di puntatori, dunque lo spazio di memoria richiesto per memorizzare la DSL cresce linearmente con la lunghezza della DSL stessa.

6.1.2 Primitiva *lookup*

Una operazione di *lookup* nella DSL permette di individuare gli elementi più vicini all'elemento cercato (o l'elemento stesso se esso è presente nella DSL). L'operazione usa come base un qualunque elemento noto, del quale si ha la certezza che esso sia presente nella DSL (ad esempio uno dei risultati parziali di un lookup precedente) e, a partire da questo, segue la più breve catena di puntatori (forward o backward) che converge sull'e-

lemento cercato. Per individuare efficientemente tale catena si segue ad ogni passo il salto più lungo possibile che non superi l'elemento cercato, in modo del tutto analogo a quanto già illustrato nel caso delle skip list tradizionali.

6.1.3 Primitiva *insert*

L'operazione di inserimento, o *insert*, è senza dubbio la più delicata, in quanto da essa dipende l'efficienza delle successive operazioni di *lookup*. Per inserire un nuovo elemento nella DSL occorre per prima cosa individuare la posizione corretta, ovvero i due elementi che, dopo l'inserimento, saranno rispettivamente l'immediato antecedente e successore dell'elemento dato. Tali elementi si individuano per mezzo di una normale operazione *lookup* e i relativi puntatori sono memorizzati nella DSL come illustrato nel paragrafo 6.1.1. Completata tale operazione il nuovo elemento è inserito nella DSL, e risulta raggiungibile da una operazione *lookup*. Tuttavia l'efficienza dell'algoritmo di ricerca dipende dalla disponibilità di puntatori *lunghi*, che permetto di saltare ampie porzioni della DSL ad ogni iterazione. Sebbene l'inserimento di un nuovo elemento abbia come effetto collate-

rale l'implicita promozione di almeno una coppia di puntatori (un puntatore `forward` e un puntatore `backward`) a un livello superiore, ciò comporterebbe una concentrazione dei puntatori più *lunghi* sui valori più *anziani*, e un coneguente sbilanciamento della distribuzione delle operazioni di gestione della DSL, a svantaggio di alcuni nodi della rete.

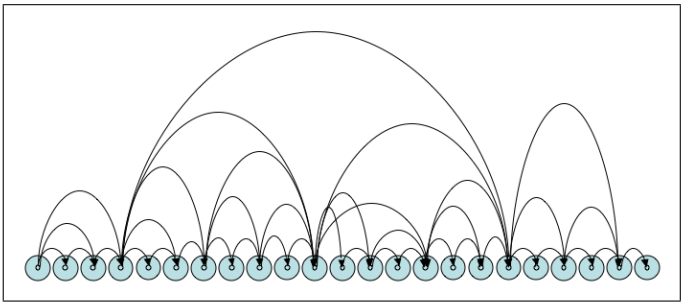

Figura 6.4: Distribuzione dei puntatori in una DSL.

Si consideri ad esempio la figura 6.4 che rappresenta una DSL dopo l'inserimento di un certo numero di entry generate casualmente; le entry sono disposte in ordine crescente da sinistra a destra. Come è ovvio alcuni nodi sono dotati di un numero maggiore di puntatori, e tali nodi sono anche in genere

quelli che consentono i salti più lunghi nelle operazioni di lookup. Per evidenziare meglio il problema, in figura 6.5 i nodi sono stati ordinati dal basso verso l'alto per numero crescente di puntatori. Non sorprende che la DSL, in mancanza di opportune contromisure, assuma la forma di un'albero. Una ulteriore nota a sfavore coniste nel fatto che, sebbene occasionalmente si notino dei tagli trasversali, in generale un percorso di lookup tra due entry non consecutive coinvolgerà i nodi più alti dell'albero, cioè le entry più ricche di puntatori. Evidentemente in un ambiente distribuito, caratterizzato da numerosi accessi concorrenti, non è accettabile il fatto che la grande maggioranza delle query vengano risolte da un sottoinsieme ristretto della rete.

Occorre pertanto elaborare una strategia per far si che, in media, ogni entry abbia un egual numero di puntatori, e che tali puntatori siano approssimativamente omogenei rispetto all'ampiezza del salto che essi definiscono. Intuitivamente tale obiettivo si può raggiungere inserendo dei puntatori ad hoc, che taglino trasversalmente la zona più bassa dell'albero, generando un certo numero di percorsi alternativi tra le entry più *giovani*, che non coinvolgano le entry più *anziane*.

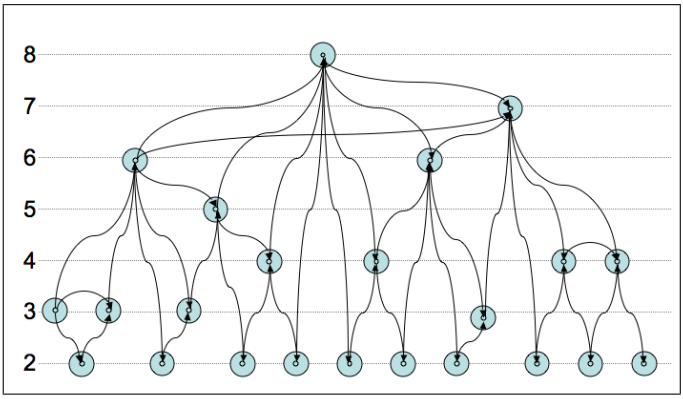

Figura 6.5: Distribuzione dei puntatori in una DSL.

Questa strategia è evidenziata in figura 6.6, in cui l'inserimento di una nuova entry, indicata con un quadrato, è accompagnato dalla creazione di due puntatori addizionali, verso altrettante entry, scelte casualmente tra le meno anziane. Come conseguenza la DSL assume una forma più simile ad un grafo, con prevedibili benefici sulla funzionalità della struttura, come rappresentato in figura 6.7.

Poiché in generale non è possibile avere una panoramica completa della popolazione della DSL la scelta della strategia per l'assegnazione dei puntatori è guidata da considerazioni

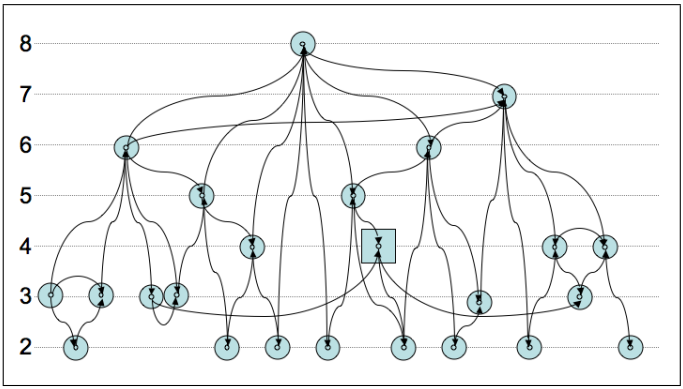

Figura 6.6: Inserimento di puntatori aggiuntivi nella DSL.

empiriche:

- i nodi che sono stati usati nell'operazione di lookup, eseguita per individare il punto d'inserimento, sono stati scelti proprio perché già forniscono la più breve catena di puntatori che converge nel punto d'inserimento: non è vantaggioso aggiungere puntatori a questi nodi

- tra i nodi che sono stati trovati nell'operazione di lookup, alcuni, pur non essendo stati usati perché non offrivano il salto più lungo possibile, compaiono con più frequenza di altri (essendo più ricchi di puntatori)

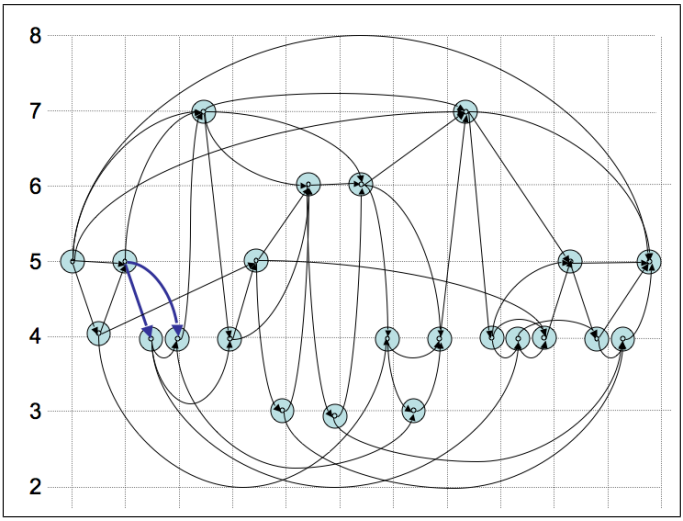

Figura 6.7: La DSL con bilanciamento dei puntatori.

- poiché il nostro obiettivo è quello di distribuire in modo omogeneo i puntatori, scegliamo tra i nodi che ne hanno meno: assegnamo i puntatori *lunghi* al nodo corrente, scegliendo, tra i nodi conosciuti, quelli usati meno recentemente in una operazione di lookup.

Questa strategia presume che ogni processore conservi una cache di elementi noti, e che tale cache rappresenti un campione abbastanza casuale dell'insieme della DSL. Questa ipotesi è giu-

stificata dal fatto che ogni operazione di lookup usa come base un elemento scelto a caso dalla cache (anche allo scopo di equidistribuire il carico di lavoro su tutti i nodi noti). Nella pratica si osserva che la cache non rappresenta un campione casuale della popolazione della DSL. Infatti le entry che hanno accumulato un maggior numero di puntatori hanno maggiore probabilità di trovarsi nella cache, e, in mancanza di un strategia di contenimento, tedono a sopraffare le altre entry. Questo fenomeno è sistematico e inevitabile: infatti le entry più anziane hanno statisticamente un patrimonio di puntatori più cospicuo rispetto a quelle che sono state inserite più di recente. Per controbilanciare questa tendenza è necessario penalizzare nell'assegnazione dei puntari quelle entry che, per anzianità o casualmente, sono sovraccariche di puntatori. A tale scopo la cache viene riorganizzata ad ogni operazione di lookup nel modo seguente:

1. le entry che compaiono con maggiore frequenza vengono portate in coda (o rimosse se la cache eccede una dimensione massima prefissata)

2. le entry che vengono utilizzate in una operazione di *lookup*

sono portate in coda alla cache

Conseguentemente, le entry usate meno di recente emergono in testa alla cache, e da qui vengono prelevate per essere usate come base in una operazione di *lookup* o per l'assegnazione dei puntatori in una operazione di *insert*. Questa ipotesi è confortata dal'osservazione sperimentale. La figura 7.2 evidenzia il numero medio di puntatori in relazione alla posizone nella cache, e la reativa varianza. Come si osserva le entry di testa presentano il più basso numero di puntatori e la minima varianza, e sono pertato i candidati ideali per l'allocazione di nuvi puntatori.

6.1.4 Primitiva *range*

La primitiva *range* permette di recuperare dalla DSL un insieme di valori che ricadono all'interno di un determinato intervallo. L'esecuzione dell'operazione di ricerca di intervalli, richiede una operazione di *lookup* sul lower bound o sull'upper bound dell'intervallo. Individuato questo si segue la catena di puntatori più lunga che convergente verso l'estremo opposto dell'intervallo selezionato: cioè si passa da ogni elemento al successivo seguento il più piccolo puntatore disponibile, fino a raggiungere l'estremo

dell'intervallo. L'algoritmo è *output senesitive* in quanto per intervalli ragionevolmente ampi, il numero di messaggi scambiati per recuperare il risultato complessivo dipende in modo predominante e secondo una legge lineare, dall'ampiezza del result set.

7 Prestazioni e Test

L'efficienza degli algoritmi proposti è stata controllata sperimentalmente. l'implementazione dimostrativa è realizzata in linguaggio Python [1] e fa uso della implementazione del protocollo Kademlia nota come Khashmir [7].

7.1 Scalabilità

Come discusso in precedenza, le prestazioni degli algoritmi di ricerca e inserimento devono essere, per quanto possibili, indipendenti dalla dimensione della rete. Due sono i fattori che influiscono sul numero di messaggi scambiati per ogni operazione:

- il numero di nodi di calcolo che compongono la sottostante rete DHT

- il numero di entry presenti nella Skip List Distribuita.

Con riferimento al primo, è noto che le hash table distribuite in generale, e Kademlia in particolare, permettono l'esecuzione di una operazioni di lookup o inserimento scambiando un numero di messaggi proporzionale a log(N), dove N è il numero di nodi di calcolo. Per quanto riguarda invece le prestazioni della Skip List Distribuita, come si è visto, essa tende a disporre valori inseriti in modo casuale secondo una gerarchia ad *albero*, che succesivamente viene riorganizzata memorizzando puntatori tra le *foglie*. Tuttavia non sorprende il fatto che le prestazioni siano proporzionali, anche in questo caso, a log(M), in cui M è il numero di entry nella DSL. Tale osservazione è riportata in figura 7.1.

Pertanto il numero complessivo di messaggi scambiati nella rete per una tipica operazione di inserimento è O(log(N) + log(M)), mentre la complessità di una range query è O(log(M) + k log(N)), se k è l'ampiezza del result set. Questi risultati

sono in linea con altri approcci presenti in letteratura.

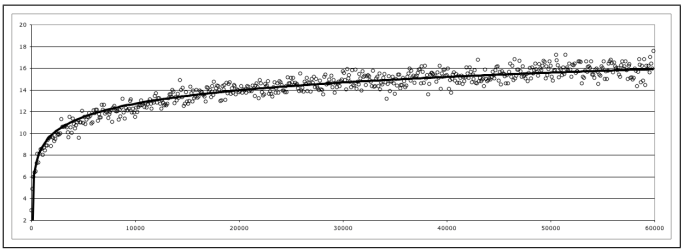

Figura 7.1: Numero di iterazioni per una operazione *lookup*.

Figura 7.1 il numero di iterazioni effettuate in media da una operazione di *lookup* in rapporto al numero di elementi presenti nella DSL. Le misurazioni sono state effettuate effettuate su un campione di 2^{16} inserimenti. ed evidenziano il fatto che l'operazione di lookup ha una complessità O(log N), e pertanto soddisfa i requisiti di scalabilità descritti nel capitolo 4.

Figure 7.2 e 7.3 numero medio i puntatori e varianza in relazione alla posizione nella cache: gli elementi più vicini alla testa sono i candidati ideali come base per le operazioni di lookup e per la creazione di nuovi puntatori. L'algoritmo *insert*, una volta individuato il punto di inserimento, seleziona alcune entry dalla testa della cache e definisce dei nuovi puntatori tra

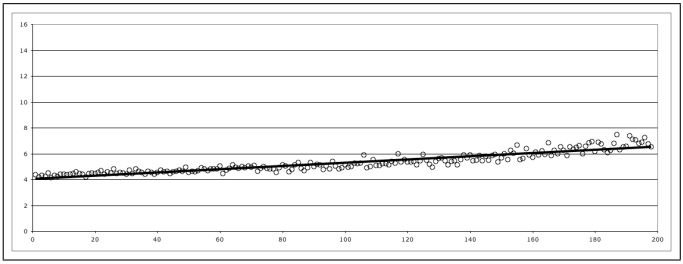

Figura 7.2: Numero medio di puntatori rispetto alla posizione nella cache.

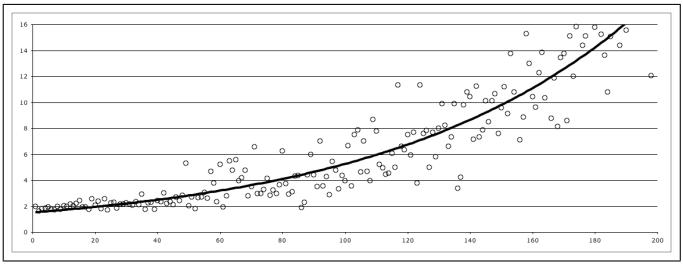

Figura 7.3: Varianza sulla media di puntatori rispetto alla posizione nella cache.

questi e l'elemento appena inserito.

7.2 Load balancing

L'equilibrio nell'impegno di ogni singolo nodo è garantito dall'adozione dell hash Table Distribuita come infrastruttura di comu-

nicazione. Infatti l'adozione di un algoritmo di hash crittografico nell'assegnazione degli identificatori alle entry della Skip List Distribuita assicura che le varie risorse siano ripartite uniformemente sulla rete. La strategia di gestione della cache privilegia l'uso dei nodi più giovani, che sono in maggioranza, rispetto ai più anziani e controbilancia la tendenza della skip list a formare *comunità* di nodi fortemente connessi, che diversamente risulterebbero sovraccarichi.

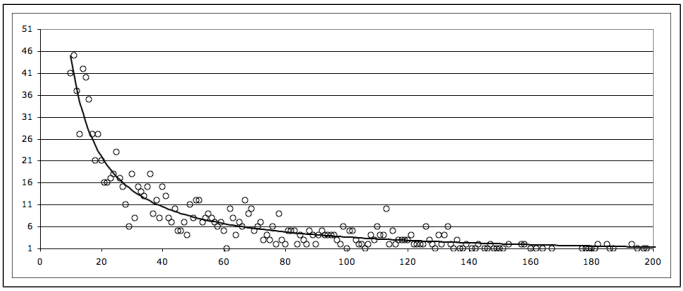

Figura 7.4: Distribuzione delle query sulla rete.

Figura 7.4 la ripartizione delle query sui nodi della rete per intervalli di frequenza monitorata su 2^{16} operazioni di lookup su una popolazione di 2000 entry. La grande maggioranza dei nodi viene interrogata meno di 100 volte e in nessun caso un nodo viene interrogato più di 1000 volte.

Tale misurazione mostra come l'algoritmo di inserimento crei una configurazione ben bilanciata nella Skip List Distribuita, che offre un buon compromesso tra occupazione di memoria e efficienza nel lookup, consentendo al tempo stesso una equa ripartizione dell'impegno computazionale necessario alla risoluzione di una query, soddisfacendo i requisiti di self-balancing, descritti nel capitolo 4.

8 Conclusioni

Abbiamo illustrato gli algoritmi e le strutture dati necessarie per l'implementazione di una struttura dati ditribuita per la memorizzazione di collezioni di dati che supporti l'esecuzione efficiente di query su intervalli. La Skip List Distribuita usa come overlay di appoggio la Distributed Hash Table per le sue provate doti di efficienza, affidabilità e scalabilità.
Gli obiettivi centrali nello sviluppo di tale struttura sono:

scalabilità: Le caratteristiche del sistema si mantengono indipendenti dal numero di nodi o di documenti memorizzati. Un aumento, anche consistente, delle dimensioni della rete (cioè del numero di nodi di calcolo su cui la struttura viene distribuita) e della quantità di dati memorizzati ha

un impatto minimo sulle prestazioni degli algoritmi.

fault tolerance: La struttura dati conserva la propria integrità anche in caso di malfunzionamenti di uno o più nodi. In particolare la skip list distribuita eredita le caratteristiche di fault tolerance della sottostante DHT, e si mantiene coerente anche in caso di frequenti avvicendamenti nella rete: l'instabilità della rete (in ogni momento un nodo può disconnettersi dalla rete e nuovi nodi possono sostituirlo) è accettata come una sua caratteristica intrinseca.

self maintenance: In un ambiente distribuito e fortemente dinamico, come quello descritto, le operazioni di manutenzione della struttura dati rappresentano un rischio inaccettabile. Qualora una operazione di manutenzione dovesse fallire avremmo un degrado delle prestazioni nel caso più favorevole, una struttura dati incosistente nel caso peggiore. A questo scopo l'algoritmo di inserimento è progettato in modo da non sbilanciare la struttura dati, e da non poter mai lasciare la struttura stessa in uno stato incosistente nel caso in cui fosse impossibile completare l'operazione.

load balancing: Nessun nodo ha un ruolo privilegiato nell'esecuzione degli algoritmi o nella memorizzazione dei dati. Tanto le richieste di elaborazione, quanto l'impegno di memorizzazione sono equidistribuiti su tutti i nodi della rete. Nel determinare un criterio di equità per la suddivsione del carico di lavoro si possono prendere in considerazione numerosi fattori, ad esempio la capacità di calcolo e memorizzazione di ogni nodo, il tempo di latenza della connessione, il fatto che il processore che esegue il nodo sia dedicato esclusivamente a tale operazione o meno, ecc.

Sebbene ognuno dei requisiti di cui sopra sia considerato fondamentale nell'implementazione di sistemi distribuiti, nessuna delle soluzioni per l'esecuzione di query su intervalli presenti in letteratura li soddisfa tutti contemporaneamente.

É stato invece dimostrato, anche con approfondite misurazioni spermentali, che la Skip List Distribuita soddisfa pienamente i requisiti esposti:

1. Gli algoritmi di inserimento e ricerca, come pure l'esecuzione di query su intervalli, richiedono ll'esecuzione di un

numero di lookup sulla sottostante DHT proporzionale a $log(N)$ in una lista di N entry; Questo risultato é pienamente soddisfacente, anche per reti di grandi dimensioni, in considerazione del fatto che le DSL permettono di archiviare grandissime quantità di dati sfruttando spazio di memorizzazione, spesso inutilizzato, disponibile ai margini della rete Internet;

2. L'affidabilità della sottostante DHT e la semplicità degli algoritmi di inserimento garantiscono una elevata tolleranza ai gasti; infatti la replicazione dei dati effettuata dalla DHT e le strategie di ripubblicazione in caso di guasto o defezione di un nodo di memorizzazione, sono sufficienti per assicurare la disponibilità dei dai, anche in caso di reti caratterizzate da elevatissimo turnover;

3. La particolarità della DSL, che permette di effettuare inserimenti e ricerche, anche concorrenti, senza richiedere operazioni di manutenzione o ribilanciamento, la rendono un candidato ideale per l'implementazione in ambienti molto dinamici, in cui non è auspicabile la presenza di

una autority di coordinamento o non sarebbero facilmente implementabili stategie più complesse di gestione;

4. Infine, si è ampiamente provato con valutazioni sperimentali che la DSL distribuisce in modo uniforme il carico di lavoro, tanto per quanto rigurda lo spazio di memoria, quanto per l'impegno di CPU, su tutti i nodi che formano la rete.

È stata inoltre presentata un implementazione dimostrativa degli algoritmi descritti, basata sulla rete Khashmir, implementazione open source del protocollo DHT Kademlia.
L'implementazione proposta consente la memorizzazione e la ricerca all'interno di una Distributed Skip List a valori interi. Sebbene in generale questo costuituisca una limitazione per la realizzazione di applicazioni non banali, abbiamo mostrato come tale limite possa essere superato con la tecnica della linearizzazione (cfr. appendice A).
Questo lavoro è stato parzialmente realizzato nell'ambito del progetto MIUR Distributed Agent-Based Retrieval Toolkit (DART), che ha come obiettivo il design di un motore di ricerca seman-

tico e distribuito, sviluppato congiuntamente da CRS4, Tiscali e Università di Cagliari.

A Linearizzazione

La linearizzazione è una tecnica di mapping basata sulla definizione di una corrispondenza biunivoca tra un generico spazio multidimensionale, cui appartengono gli elementi che si desidera rappresentare, e uno spazio unidimensionale, che meglio si adatta alla memorizzazione in un computer. Le tecniche di linearizzazione trovano applicazione nei campi più svariati: dal comune problema di rappresentare in memoria centrale (caratterizzata da un indirzzamento sequenziale) matrici a valori reali in analisi numerica, ai sistemi informativi territoriali, alla gestione della memoria nei sistemi operativi.

Nel caso specifico dei database, l'uso di tecniche di linearizzazione ha essenzialmente due obiettivi:

1. Gli elementi di uno spazio multidimensionale vengono indicizzati rispetto a uno spazio di indirizzamento lineare (ad esempio in memoria centrale o su memoria di massa). Dato che una selezione viene specificata per mezzo di intervalli di valori ammissibili di uno o più attributi, è generalmente desiderabile che elementi vicini nello spazio multidimensionale vengano mappati in posizioni vicine nello spazio unidimensionale. Si noti che il normale ordinamento lessicografico, spesso usato in geometria computazionale, non gode di questa proprietà.

2. La tecnica di linearizzazione deve essere tale che, dato un oggetto di uno spazio multidimensionale, specificato per mezzo delle coordinate dei suoi estremi, e i punti di uno spazio unidimensionale corrispondenti a tali estremi, l'intervallo che essi delimitano includa pochi (o idealmente nessuno) punti non appartenenti all'oggetto originale. Ad esempio dato lo spazio $\Re^2$ e un rettangolo definito in esso tramite le coordinate di due vertici opposti, è desiderabile che l'intervallo in $\Re$ delimitato dalle posizioni dei vertici linearizzati contenga solo pochi punti non appartnenti al

rettangolo stesso.

Esistono numerose tecniche di linearizzane, al di la del semplice ordinamento lessicografico, usato comunemente per la rappresentazione di dati tabulari o matrici numeriche. Una soluzione alernativa, consistente nell'interlacciamento, bit per bit, delle coordinate originali in un unica coordinata è descritta in [10,20]. Un altro metodo, descritto in [8] è basato sulle space-filling curve, discusse fin dal diciannovesimo secolo da Hilbert [13] e Peano [22].

Nel seguito si decrivono i principali schemi di linearizzazione. Per semplicità di rappresentazione si fa riferimento al caso di linearizzazione $\Re^2 \rightarrow \Re$ di punti dal piano reale alla retta reale.

A.1 Funzioni di Mapping.

Come già accennato, il più semplice schema di linearizzazione è costituito dalla tecnica di ordinamento lessicografico. in questo schema i punti vengono ordinati *per righe* o *per colonne*, scegliendo quindi una coordinata dominante. Nello sche-

ma denomiato *snake scan*, l'ordinamento rispetto alla seconda coordinata è alternativamente crescente e decrescente.

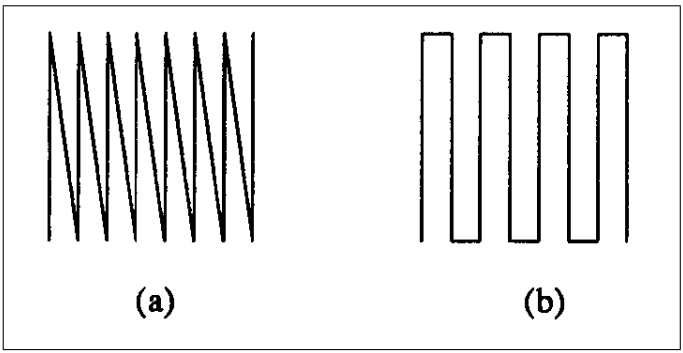

Figura A.1: Ordinamento lessicografico per colonne (a) e snake scan (b).

Nello schema lessicografico per colonne, rappresentato in figura A.1(a), al punto di coordinate (x,y) viene assegnata una coordinata linearizzata $x*ydim+y$. Ciò comporta due problemi

- in generale l'ampiezza $ydim$ non è nota, sebbene questo metodo si presti alla rappresentazione di matrici di dimensioni fisse, altre applicazioni risultano meno dirette.

- punti vicini nello spazio di origine possono essere mappati in punti lontani nello spazio unidimensionale. Questo pro-

blema è meglio evidenziato in figura A.2: sebbene i punti *p1* e *p3* siano tra loro molto vicini, la linearizzazione li rappresenta con coordinate relativamente lontane, in aggiunta il punto *p2*, molto distante tanto da *p1* quanto da *p2*, viene mappato in una posizione intermedia tra i due e a una distanza minore da entrambi.

Quest'ultimo requisito, sebbene desiderabile non può essere raggiunto con arbitraria precisione. Si consideri un generico mapping da uno spazio m-dimensionale a uno spazio n-dimensionale, con n ¡ m. Ogni punto dello spazio m-dimensionale ha in generale 2m punti contigui. Per contro un punto nello spazio n-dimensionale ha solo 2n punti contigui, dunque (2m - 2n) fra i *vicini* dell'elemento originale devono essere spostati, violando la località dello spazio originale. Le migliori tecniche di linearizzazione sono tese a minimizzare gli effetti di tale anomalia.

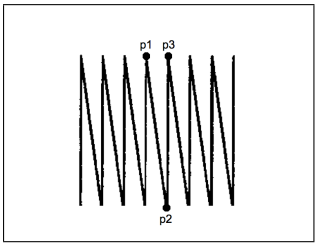

Figura A.2: Anomalie dell'ordinamento lessicografico.

A.2 z-curve

La tecnica delle z-curve è stata proposta in [21] e [19]. Il mapping fra lo spazio multidimensionale e lo spazio undimensionale avviene interlacciando bit per bit le rappresentazioni binarie delle coordinate dell'elemento. Il risultato è mostrato in figura A.3, il tipico andamento a zig zag è all'origine del nome z-curve.
La linearizzazione mediante z-curve può essere pensata come un processo incrementale, in cui lo spazio da linearizzare viene suddivizo in quadranti, i cui baricentri sono linearizzati. In un passo successivo si procede alla linearizzazione di ogni singolo quadrante sudivvidendolo ulteriormente e ripetendo la procedura. Tale procedimento ricorsivo è illustrato in figura **??** (a), (b) e (c).

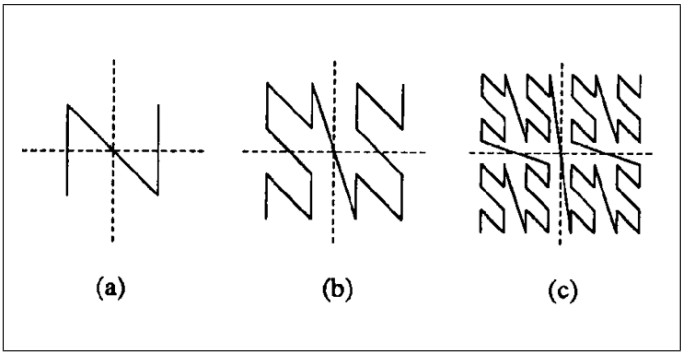

Figura A.3: Linearizzazione con z-curve.

A.3 Gray coding

La tecnica di Gray Coding è stata proposta in [10] e [9]. Consiste nel codificare un numero in rappresentazione binaria in modo tale che numeri successivi differiscano esattamente per un bit. Anche in questo caso è possibile considerare un approccio incrementale, dividendo via via il piano in quadranti più piccoli e applicando la tecnica a tali suddivisioni.

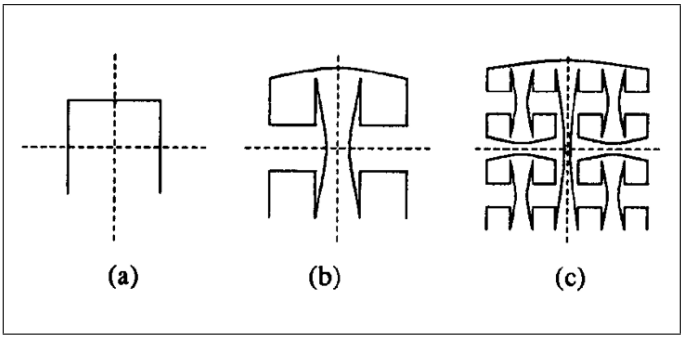

Figura A.4: Linearizzazione con Gray Coding.

A.4 Curva di Hilbert

Come osservato in precedenza, la linearizzazione altera la località dei punti dello spazio di partenza: nel caso del mapping $\Re^2 \rightarrow \Re$ un punto in $\Re^2$ ha quattro *vicini*, mentre un punto di $\Re$ ne ha solo 2. I punti rimanenti devono necessariamente essere allontanati; l'obiettivo è fare in modo che (i) punti vicini nell'insieme lineare siano vicini anche nell'insieme di partenza e, allo stesso tempo, che (ii) punti vicini nello spazio di partenza siano *non troppo lontani* nello spazio lineare. Come si può facilmente osservare l'ordinamento lessicografico viola grossolanamente la condizione (ii) mentre le tecniche z-curve e gray coding violano

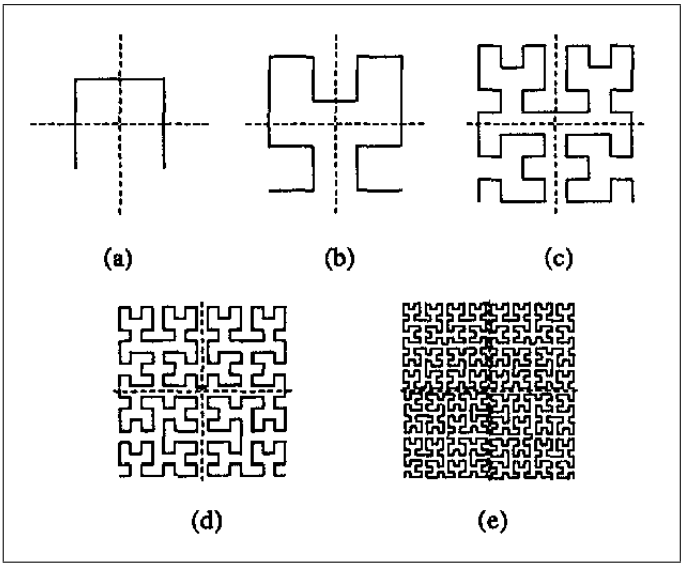

Figura A.5: Linearizzazione con curva di Hilbert.

entrambe la condizione (i).

La linearizzazione mediante curve di Hilbert invece si dimostra maggiormente efficace, rispettando entrambe le condizioni. La costruzione delle curve di Hilbert, corredata da una dettagliata esposizione delle tecniche di query e range selection è descritta in [15].

B Source Code

```
1   class FastLookupSession:
2      def __init__(self, kashmir, target):
3         global global_cache
4
5         self.done = False
6         self.found = False
7         self.target = int(target)
8         self.kashmir = kashmir
9         self.base = int(global_cache.get_random
10          (kashmir))
11        self.path = [self.base]
12        self.forward = True
13
14     def fw_callback(self, param):
15        skip_chain = SkipChain(self.target, param)
16
17        if not self.done:
18           for i in skip_chain.skip_vector:
19           self.path.append(i)
20
```

```
21            if (skip_chain.contains_target()):
22               self.found = True
23               self.done = True
24
25            elif (skip_chain.forward_jump() >
26            self.base):
27                self.base =
28            skip_chain.forward_jump()
29                logfile = Log()
30                logfile.log_lookup(
31            self.target, self.base)
32                self.kashmir.valueForKey(
33                  sha("fw" + str(self.base)).digest(),
34            self.fw_callback)
35            else:
36               pass
37
38      def bw_callback(self, param):
39         skip_chain =
40           SkipChain(self.target, param)
41         if not self.done:
42            for i in skip_chain.skip_vector:
43               self.path.append(i)
44
45           if (skip_chain.contains_target()):
46              self.found = True
47              self.done = True
48           elif (skip_chain.backward_jump(self.base)
49           < self.base):
50              self.base = skip_chain.
51           backward_jump(self.base)
52              logfile = Log()
53              logfile.log_lookup(
```

```
54            self.target, self.base)
55               self.kashmir.valueForKey(
56                 sha("bw" + str(self.base)).digest(),
57            self.bw_callback)
58         else:
59             pass
60
61      def lookup(self):
62         logfile = Log()
63         logfile.log_lookup(
64           self.target, self.base)
65         logfile.log_cache()
66         if self.base != self.target:
67            if self.base < self.target:
68               self._forward_lookup()
69            else :
70               self._backward_lookup()
71            while not self.done:
72               reactor.iterate()
73
74      def _forward_lookup(self):
75         reactor.callLater(1, self.stop_session)
76         self.kashmir.valueForKey(
77            sha("fw" + str(self.base)).digest(),
78           self.fw_callback)
79
80      def _backward_lookup(self):
81         reactor.callLater(1, self.stop_session)
82         self.kashmir.valueForKey(
83            sha("bw" + str(self.base)).digest(),
84           self.bw_callback)
85
86      def stop_session(self):
```

```
87          self.done = True
```

```
1   class InsertSession(FastLookupSession) :
2        def __init__(self, khashmir, target):
3             FastLookupSession.__init__(self,
4           khashmir, target)
5             self.first_base = self.base
6             self.khashmir = khashmir
7
8        def insert(self):
9             self.lookup()
10
11            if not self.found:
12                 global global_count
13                 global_count += 1
14                 skip_chain = SkipChain(self.target,
15          self.path)
16                 self.store_value(self.target, 0)
17                 self.store_path(skip_chain.pred,
18          self.target)
19                 self.store_path(self.target,
20          skip_chain.next)
21                 self.store_random_paths(1)
22
23                 logfile = Log()
24                 logfile.log_insert(self.target)
25
26       def store_random_paths(self, num):
27            for i in range(num):
28                 global global_cache
29                 random_node =
30          global_cache.get_random(
31          self.khashmir)
32                 if (int(random_node) is not 0):
33                      self.store_path(
```

```
34              random_node, self.target)
35
36          def path_stored(self, value):
37              self._path_stored = True
38
39          def store_value(self, key, value):
40              global global_cache
41              random_node =
42             global_cache.add_nodes([key])
43              self._path_stored = False
44              self.kashmir.storeValueForKey(
45             sha(str(key)).digest(), str(value),
46             self.path_stored)
47              while not self._path_stored:
48                  reactor.iterate()
49
50          def store_path(self, base, target):
51              a = min(int(base), int(target))
52              b = max(int(base), int(target))
53              self.store_fw_path(a, b)
54              self.store_bw_path(b, a)
55              global global_stats
56              global_stats.add_pointer(a, b)
57
58          def store_fw_path(self, base, target):
59              self._path_stored = False
60              self.kashmir.storeValueForKey(
61             sha("fw" + str(base)).digest(),
62             str(target), self.path_stored)
63              while not self._path_stored:
64                  reactor.iterate()
65
66          def store_bw_path(self, base, target):
```

```
67              self._path_stored = False
68              self.kashmir.storeValueForKey(
69            sha("bw" + str(base)).digest(),
70            str(target), self.path_stored)
71              while not self._path_stored:
72                   reactor.iterate()
73
74        def stop_session(self):
75              self.done = True
76
```

```
1   class Cache:
2       def __init__(self):
3           self.cache_vector = []
4           self.max_length = 200
5
6       def add_nodes(self, nodes):
7           for node in nodes:
8               if int(node) != 0:
9                   if int(node) not in
10              self.cache_vector:
11                      self.cache_vector.insert(
12                  self.max_length,
13                  int(node))
14                   else:
15                      self._move_to_tail(
16                  int(node))
17
18          if  (len(self.cache_vector) -
19          self.max_length > 0):
20                self.cache_vector =
21          self.cache_vector
22             [0:self.max_length]
23
24      def _move_to_tail(self, node):
25          self.cache_vector.remove(int(node))
26          self.cache_vector.append(int(node))
27
28      def get_random(self, khashmir):
29          if len(self.cache_vector) >
30          (0.5 * self.max_length):
31
32              head = self.cache_vector[0]
33              self._move_to_tail(head)
```

```
34                  if not self.expired(head, khashmir):
35                       return str(head)
36
37              return str(0)
38
39
40
41          def expired(self, key, khashmir):
42               if key == 0:
43                    return False
44
45               get_session = Session()
46               reactor.callLater(2,
47              get_session.stop_session)
48               khashmir.valueForKey(
49              sha(str(key)).digest(),
50              get_session.callback)
51               while not get_session.done:
52                    reactor.iterate()
53
54               if  len(get_session.result) > 0:
55                    return False
56               else:
57                    return True
```

```
1   class SkipChain:
2        def __init__(self, target, skip_vector):
3             self.skip_vector = []
4             self.target = target
5             self.pred = 0
6             self.next = 0
7
8             global global_cache
9             global_cache.add_nodes(skip_vector)
10
11
12            for i in skip_vector:
13                 self.skip_vector.append(int(i))
14
15                 if int(i) > self.pred and int(i)
16           < self.target:
17
18                      self.pred = int(i)
19                 if (int(i) < self.next and int(i)
20           > self.target)
21           or (int(i) > self.target
22           and self.next == 0):
23
24                      self.next = int(i)
25
26        def contains_target(self):
27             for value in self.skip_vector:
28                  jump = int(value)
29                  if (jump == self.target):
30                       return True
31             return False
32
33        def forward_jump(self):
```

```
34              longest_jump = 0
35              for value in self.skip_vector:
36                   jump = int(value)
37                   if (jump < self.target and jump >
38           longest_jump):
39
40                        longest_jump = jump
41
42              return longest_jump
43
44         def backward_jump(self, base):
45              longest_jump = int(base)
46              for value in self.skip_vector:
47                   jump = int(value)
48                   if (jump > self.target and jump
49           < longest_jump):
50
51                        longest_jump = jump
52
53              return longest_jump
54
```

Bibliografia

[1] *The Python Language Reference Manual*. Network Theory Ltd, 2003.

[2] J. Aspnes and J. Shah. Skip graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 384–393, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[3] A. Bharambe, Agrawal M., and S. Seshan. Mercury: Supporting scalable multi-attribute rangequeries. In *SIGCOMM. (2004)*.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered dht applications. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 97–108, New York, NY, USA, 2005. ACM Press.

[5] I. Clarke, O. Sandberg, and B. Wiley. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings oftheWorkshop on Design Issues in Anonymity andUnobservability (2000)*.

[6] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In

WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases, pages 25–30, New York, NY, USA, 2004. ACM Press.

[7] Burris T. Ewell. Khashmir - http://sourceforge.net/projects/khashmir/.

[8] C. Faloutsos. Fractals for secondary key retrieval. In *Proc. ACM Conference on the Pronciples of Database Systems, 1989.*

[9] C. Faloutsos. Gray codes for partial matches and range queries. In *IEEE transactions on Software Engineering, 1987.*

[10] C. Faloutsos. Multiattribute hashing using gray codes. In *Proc. ACM-SIGMOD International Conference on the Management of Data. 1985.*

[11] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semanticweb applications. In *Proceedings of the12th International Conference on World WideWeb.*

[12] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlaynetwork with practical locality properties. In *USITS. (2003).*

[13] D. Hilbert. Uber die steiuge abbildung einer linie auf ein flachenstuck. In *Math. Ann., 38, 1891.*

[14] R. et al. Huebsch. Querying the internet with pier. In *VLDB. (2003).*

[15] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *SIGMOD '90: Proceedings of the 1990 ACM*

SIGMOD international conference on Management of data, pages 332–342, New York, NY, USA, 1990. ACM Press.

[16] A. Kothari, D. Agrawal, A. Gupta, and S. Suri. Range addressable network: A p2p cache architecture for data ranges. In *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 14, Washington, DC, USA, 2003. IEEE Computer Society.

[17] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[18] W. et. al. Nejdl. Edutella: A p2p networking infrastructurebased on rdf. In *Proceedings of the 12th InternationalConference on World Wide Web (2003)*.

[19] J. A. Orenstein. Redundancy in spacial databases. In *Proc. ACM-SIGMOD International Conference on the Management of Data. 1989*.

[20] J. A. Orenstein. Spatial query processing in a object-oriented database system. In *Proc. ACM-SIGMOD International Conference on the Management of Data. 1986*.

[21] J. A. Orenstein and T. H. Merett. A class of data structures for associative searching. In *Proc. SIGACT SIGMOD Symposium on the Principles of Database Systems. 1984*.

[22] G. Peano. Sur une courbe qui remplit toute une aire plaine. In *Math. Ann., 36, 1890*.

[23] W. Pugh. Concurrent maintenance of skip lists. Technical report, College Park, MD, USA, 1990.

[24] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[25] S. Ramabhadran, J. Hellerstein, S. Ratnasamy, and S. Shenker. Prefix hash tree - an indexing data structure over distributed hash tables.

[26] S. Ratnasamy, P. Francis, M Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACMMiddleware*.

[28] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[29] P. Triantafillou and T. Pitoura. Towards a unifying framework for complex query processingover structured peer-to-peer data networks. In *DBISP2P. (2003)*.

[30] M. et. al. Waldman. Publius:a robust, tamper-evident, censorship-resistantweb publishing system. In *Proceedings of the 9thUSENIX Security Symposium (2000)*.

[31] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.