UNIVERSITÀ DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea Specialistica

XCONE: RANGE QUERY IN SISTEMI P2P

Relatore:
Prof.ssa LAURA RICCI

Candidato:
DAVIDE CARFÌ

Prof. MASSIMO COPPOLA

ANNO ACCADEMICO 2007-2008

Indice

Capitolo 1

Introduzione

Le applicazioni distribuite basate sul modello P2P [1, 2, 3, 4, 5] (Peer-to-Peer) sono state negli ultimi anni oggetto di una grande diffusione nel settore informatico. Tali sistemi si caratterizzano per la presenza di un insieme di nodi equivalenti, *peer*, capaci di auto-organizzarsi e che risultino in grado di condividere un insieme di risorse distribuite all'interno della rete. L'evoluzione delle applicazioni ha visto l'adozione, da parte degli sviluppatori, di stili architetturali che si distaccano sempre più dal paradigma centralizzato. Le reti P2P infatti basano il proprio paradigma computazionale sulla decentralizzazione del controllo, sulla condivisione delle risorse e risultano caratterizzate da un ambiente di esecuzione estremamente dinamico che può assumere dimensioni di larga scala. Rispetto al consolidato modello di riferimento client/server, le reti P2P non pretendono di sostituirsi ma piuttosto di fornire una valida alternativa in tutti quei contesti in cui la presenza di un punto di fallimento del sistema o la scalabilità dell'applicazione possa diventare un aspetto critico.

La prima generazione di applicazioni P2P come Napster[2], Gnutella[1] e SETI[6] ha contribuito alla diffusione del paradigma P2P evidenziandone le enormi potenzialità, in particolare le capacità di aggregazione di una grande mole di dati ma anche la possibilità di condividere potenze di calcolo in maniera

estremamente poco costosa. Con l'evoluzione delle potenzialità della rete, le applicazioni P2P di prima generazione hanno iniziato a mostrare i primi limiti, in particolare la *topologia non strutturata* della rete. Un sistema P2P non strutturato è caratterizzato dal fatto che ogni risorsa è localizzata sul peer che la mette a disposizione, non esistono informazioni relative alla loro locazione e la ricerca risulta essere difficoltosa a causa della mancanza di *"organizzazione"* all'interno del sistema. D'altra parte, l'inserimento o rimozione di nodi dalla rete risulta essere un'operazione poco costosa e semplice.

Nella seconda generazione le applicazioni P2P iniziano ad essere basate su *reti di tipo strutturato*. Le reti P2P strutturate utilizzano per la costruzione della topologia di rete tra i propri peer una struttura di base con specifiche proprietà. In questo modo è stato possibile ottimizzare le operazioni di ricerca. La principale struttura di base utilizzata per la costruzione della topologia di rete è la Distributed Hash Table, DHT [7, 8, 9, 10]. Le DHT mantengono sia i peer che le risorse distribuite in rete attraverso l'impiego di una funzione hash che ne garantisce la distribuzione casuale, con elevata probabilità. Con le DHT si è arrivati ad avere *costi di ricerca logaritmici* organizzando la rete in base ai contenuti attraverso la costruzione di un indice distribuito. Per contro, l'espressività delle query rimane limitata a ricerche di tipo esatto e su singolo attributo. Infatti avendo un indice distribuito monodimensionale non risulta ancora praticabile eseguire, ad esempio, *range query* o ricerche su un *numero arbitrario di attributi* a meno di costi di ricerca esponenziali.

Negli ultimi anni si è assistito all'utilizzo di applicazioni P2P in nuovi contesti come il Gaming online[11] o il Grid Computing[12]. Queste tipologie di applicazioni presentano un insieme di caratteristiche comuni quali ad esempio l'elevata *dinamicità* delle informazioni, la rappresentazione di una risorsa mediante *più attributi* o la ricerca di risorse mediante criteri di *selezione strut-*

turati. Se consideriamo per esempio le applicazioni P2P in un contesto come il *Grid Informazion Service*[13], appare chiaro come tali applicazioni debbano effettuare un discovery delle risorse condivise in maniera immediata, efficiente, scalabile ma, il più importante, in maniera espressiva. L'espressività di ricerca risulta il punto cruciale delle applicazioni P2P. Nel caso del Grid Informazion Service, contesto in cui la presente tesi si è focalizzata, risulta fondamentale fornire un supporto all'esecuzione di query del tipo: *"Find N resource with: CPUSpeed* $\geq$ *2.0GHz* ***and*** *0.8* $\leq$ *WorkLoad* $\leq$ *0.4* ***and*** *RAM Free* $\geq$ *20%"*. A questo punto emerge chiaramente che gli attuali modelli P2P non siano in grado di soddisfare tali requisiti e anzi risultino fortemente limitati sotto l'aspetto dell'espressività di ricerca delle risorse.

In letteratura sono state presentate diverse proposte [14, 15, 16, 17, 18] che si occupano di potenziare i modelli P2P incrementandone di conseguenza l'espressività di ricerca delle risorse. Tali proposte sono classificabili all'interno di due aree di ricerca: *(i)* proposte che modificano le DHT esistenti, *(ii)* proposte che sostituiscono la struttura di base DHT attraverso strutture distribuite di tipo gerarchico.

I lavori basati sulla modifica delle DHT esistenti si focalizzano nel tentativo di introdurre all'interno della DHT il concetto di *località dei dati*. La rete P2P è costruita attraverso l'utilizzo della DHT e il supporto a ricerche più espressive, come ad esempio range query, viene fornito attraverso la manipolazione della funzione hash della DHT. La funzione hash dovrà preservare la località dei dati. Nel lavoro presentato da [16], MAAN *Multi-Attribute Addressable Network*, risulta chiaramente visibile come il fulcro del modello prestazionale in tali sistemi sia la scelta della funzione hash. Infatti in presenza di distribuzioni non uniformi i modelli iniziano a mostrare problematiche di *load-balancing*.

Nella stessa tipologia di approccio troviamo i lavori basati sull'utilizzo delle

space filling curves [15, 19, 20, 21] che consentono di effettuare *range query multi-attributo*. Le space filling curves si occupano di effettuare la linearizzazione di uno spazio multi-attributo in uno spazio ad una dimensione, preservando al meglio la proprietà di località dei dati originari. In questa direzione si colloca il lavoro di ricerca proposto da [15] con il sistema Squid. Squid è una rete P2P strutturata che utilizza una DHT e che memorizza le risorse applicando la funzione space filling di Hilbert. Questo approccio delega il compito di distribuzione delle risorse nella rete alla funzione di space filling. Come nel caso precedente le funzioni di space filling presentano delle problematiche sia per quanto riguarda il bilanciamento del carico in caso di distribuzioni non uniformi, sia per la difficoltà di gestione delle strutture dati al crescere del numero di attributi gestiti. Quest'ultima problematica viene definita *"curse of dimensionality"*.

A seguito delle problematiche illustrate la ricerca si è indirizzata verso un'altra direzione, basandosi sull'utilizzo di strutture di base alternative alle DHT, strutture di tipo *gerarchico* e *distribuite*. Ad esempio nel caso della proposta presentata in BATON[17] la struttura utilizzata è simile ad un B-albero. Si tenta quindi di sfruttare le proprietà possedute dalle strutture gerarchiche, già note ed utilizzate in ambienti non distribuiti, per incrementare l'espressività delle operazioni di ricerca nei modelli P2P. Nel caso di BATON l'approccio presenta degli evidenti overhead di gestione soprattutto in presenza di attributi dinamici per effetto del continuo bilanciamento della struttura distribuita. Risulta pertanto fondamentale che le proposte siano in grado di gestire efficacemente il dinamismo del sistema senza introdurre elevati overhead di gestione e colli di bottiglia verso particolari peer della rete. Un approccio che si spinge in questa direzione è quello presentato da [22] in Cone. Cone può essere considerato una soluzione ibrida, infatti, la strutturazione della rete avviene

attraverso una DHT ma al contempo prevede la costruzione di una struttura dati distribuita per la ricerca delle risorse. La struttura dati è basata su un albero distribuito ed a differenza delle altre soluzioni risulta essere molto promettente sia per quanto riguarda il bilanciamento del carico che per la gestione della struttura in caso di aggiornamenti. Per contro, in Cone è possibile effettuare una ristretta tipologia di query del tipo: *"Trova k risorse di dimensione maggiore di S"* e tale requisito riduce l'espressività di ricerca nell'approccio.

Dopo aver analizzato attentamente gli approcci presenti in letteratura, la presente tesi si è focalizzata nel realizzare una soluzione in grado di non alterare il comportamento della DHT ma bensì di integrarlo attraverso una struttura dati distribuita di tipo gerarchico. Proponiamo una soluzione in grado di gestire il dinamismo delle risorse con la possibilità di effettuare ricerche più espressive attraverso range query. Il risultato di questa tesi è stato il sistema XCone. Tale sistema si ispira al lavoro effettuato da Cone ma introduce significativamente degli elementi di innovatività. In XCone è stato possibile effettuare attraverso un approccio originale la risoluzione di range query. Come vedremo si è generalizzato il concetto di struttura gerarchica attraverso l'introduzione di due nuove funzioni: (i) *funzione di mapping* e (ii) *funzione di digest*. L'introduzione di queste due funzioni ha reso XCone una struttura estremamente generica.

In XCone un ulteriore aspetto innovativo è rappresentato dall'integrazione tra la propria struttura gerarchica e una Distributed Hash Table. A differenza di altri approcci XCone non altera la struttura della DHT ereditandone in tale modo le proprietà di bilanciamento del carico tra i peer della rete. La struttura gerarchica di XCone corrisponde ad un *trie* costruito sugli identificativi della DHT ma mantenuto in uno spazio distinto. Quando un peer si unisce ad XCone ottiene un identificatore (ID) dalla DHT e di conseguenza viene associato al

rispettivo nodo foglia del trie. L'associazione *foglia/peer* rimane fissata per tutta la permanenza del peer in rete e, come già detto, eredita la proprietà di assegnamento casuale garantito dalla *funzione hash* utilizzata dalla DHT. In XCone le risorse sono memorizzate localmente ai peer che rappresentano le foglie dell'albero, successivamente, ogni nodo logico interno dell'albero viene assegnato ad un peer attraverso la *funzione di mapping* e di conseguenza le proprie risorse aggregate secondo una *funzione di aggregazione*.

La funzione di mapping determina quindi il numero di nodi logici non foglia assegnati ad ogni peer della rete. In linea generale ogni nodo fisico può gestire un qualsiasi nodo logico ma per ottenere buone prestazioni nelle operazioni occorrerà definire dei vincoli nelle proprietà della funzione di mapping. XCone come vedremo utilizza una specifica funzione di mapping che rispetta le proprietà introdotte.

Per quanto riguarda la funzione di digest, con tale funzione viene associato ad ogni nodo interno dell'albero una struttura che sintetizza le chiavi contenute nel sottoalbero radicato nel nodo interno. Come vedremo è possibile definire diverse funzioni di digest, in particolare nella tesi saranno presentati due proposte. Il primo approccio prende spunto dalla proposta presentata da [23], introducendo i *"BitVector Indexes"*. L'altra proposta trae fondamento dal campo delle reti di sensori, presentata da [24], la funzione di digest prende il nome di *"q-digest"*.

Nel caso della ricerca è stato realizzato un algoritmo di ricerca delle risorse in distribuito che utilizza le informazioni di digest possedute da ogni peer. Come vedremo in tale modo si è minimizzata la visita dell'albero XCone. Da precisare che l'operazione di ricerca, diversamente dalle classiche operazioni di ricerca all'interno di strutture gerarchiche, è eseguita partendo da un qualunque foglia dell'albero. Infatti l'esplorazione dell'albero è stata divisa in due

fasi. Nella prima fase denominata di *"trickling"* si è gestita la propagazione dell'operazione di ricerca verso i nodi logici del cammino che porta dalla foglia alla radice. Nella seconda fase denominata di *"esplorazione"* si è invece gestita la ricerca in maniera da indirizzare l'operazione verso i sotto-alberi promettenti massimizzando le operazioni di potatura.

Infine l'architettura del sistema è stata sviluppata seguendo i dettami dello sviluppo ingegneristico del software. Si è realizzato un sistema modulare ed estremamente espandibile, il tutto finalizzato a rendere XCone la soluzione P2P in grado di rispondere ad un insieme delle problematiche illustrate. Infatti XCone supporta sia contesti applicativi come il Grid Informazion Service, in cui le risorse da localizzare sono rappresentate ad un insieme di caratteristiche degli hosts della rete, che contesti applicativi come il Gaming-Online. In quest'ultimo caso il punto di forza di XCone rimane l'astrazione introdotta dalla funzione di aggregazione consentendo la descrizione di un insieme di oggetti passivi di un mondo virtuale, per esempio, attraverso una opportuna informazione di digest come risultato del processo di aggregazione.

La tesi è così organizzata. Nel capitolo 2 sono presentati i principali approcci presenti in letteratura per la gestione di query complesse in reti P2P. In particolare verranno classificate le varie proposte evidenziandone vantaggi e limiti. Nel capito 3 verrà presentata l'architettura di XCone. Saranno illustrate le principali operazioni ed analizzate le relative proprietà. Nel capitolo 4 verrà illustrata l'implentazione del sistema XCone. Saranno mostrati il framework utilizzato, le scelte architetturali, l'organizzazione dei packages e il diagramma delle classi. Infine verranno illustrate le principali interazioni tra le istanze del sistema. Nel capitolo 5 saranno descritti gli esperimenti per la valutazione di XCone e saranno illustrati i risultati sperimentali ottenuti. Infine nel capitolo 6 saranno riassunte alcune considerazioni conclusive e i possibili sviluppi futuri

al sistema.

Capitolo 2

Stato dell'Arte

In questo capitolo verranno illustrati gli attuali approcci presenti in letteratura. In particolare verranno classificate le diverse proposte evidenziandone vantaggi e limiti.

2.1 Introduzione

Le reti P2P rappresentano attualmente una promettente soluzione per la condivisione di risorse in ambienti distribuiti. Possiamo infatti notare come le *Ditributed Hash Table* (DHT)[7] sono alla base di numerosi servizi quali il file-sharing, storage network e il content delivery network. Negli ultimi anni si è inoltre assistito all'applicazione di tali sistemi verso nuovi contesti come il Gaming-online, il Semantic Web[25] o il Grid Computing. Il risultato è stato la focalizzazione della ricerca scientifica in una ben nota problematica delle reti P2P, cioè la mancanza di espressività dei criteri di selezione delle risorse[26].

Riuscire a recuperare in maniera efficiente un insieme di risorse che soddisfano specifici criteri è un requisito indispensabile sia nei sistemi Grid quanto nelle applicazioni di Gaming-online. Consideriamo il caso dei sistemi Grid. Un sistema Grid è una infrastruttura di base necessaria per la condivisione di un insieme di risorse[13] (desktops, clusters computazionali, supercomputers, sen-

sori o strumenti scientifici). Una delle funzionalità di base dell'infrastruttura consiste nel selezionare su larga scala un insieme di risorse secondo specifici criteri (*Resource Discovery/Selection*).

In questi nuovi ambiti le reti P2P stanno lentamente convergendo ponendo alla ricerca scientifica nuovi interrogativi[12].

2.2 Query complesse in sistemi P2P: problemi aperti

Attraverso le DHT le reti Peer-to-Peer sono in grado di fornire una infrastruttura scalabile e con funzionalità di base nella ricerca delle risorse. Il classico tipo di query supportate da questi sistemi sono del tipo *"Trova i file il cui nome contenga una determinata stringa"*. Per poter applicare le reti P2P in nuovi contesti occorre aggiungere maggiore espressività al linguaggio d'interrogazione riuscendo ad esprimere *correlazioni o combinazioni tra le proprietà delle risorse*.

Una prima classificazione basata sul numero di attributi specificati in una query porta alla definizione di:

1. Query a *singola dimensione*, nel caso in cui il criterio di ricerca coinvolga un solo attributo.

2. Query *multidimensionali*, nel caso in cui il criterio di ricerca riguardi più attributi.

Come è facile intuire le query muldimensionali risultano significativamente più complesse. Focalizzandoci adesso sul criterio di selezione possiamo ulteriormente classificare le query in:

- **Exact match query**, in cui vengono specificati i valori esatti per tutti gli attributi. Esempio *"Trova delle risorse che abbiano OS=Linux e CPU=1Ghz e RAM=512MB "*.

- **Partial match query**, rappresentano exact query in cui vengono specificati valori esatti solo per un sottoinsieme degli attributi. Ad esempio supponendo che una risorsa sia caratterizzata dai sequenti attributi: CPU, RAM, Memoria disponibile, Workload, la query risulta del tipo *"Trova delle risorse che abbiano OS=Linux e RAM=512MB"*.

- **Range query**, in cui vengono specificati per tutti o alcuni attributi dei range di valori ammissibili. Esempio: *"Trova delle risorse che abbiano* $1Ghz \leq CPU \leq 3Ghz$ *e* $512MB \leq RAM \leq 1GB$*"*.

- **Boolean query**, in cui si specificano le condizioni sui tutti o alcuni attributi attraverso operatori booleani. Esempio *"Trova delle risorse che (not* $RAM \leq 512MB$*) and (not OS=Linux)"*.

- **Nearest-Neighbor Query**, in cui vengono specificati per un insieme di attributi sia un insieme di condizioni che una funzione di selezione definita *metrica*. La metrica consente quindi di differenziare le risorse rispetto alla distanza dalla soluzione ottimale di ricerca. Esempio *"Trova delle risorse che abbiano* $CPU \geq 1Ghz$ *e* $512MB \leq RAM \leq 1GB$ *considerando come metrica la distanza dello spazio euclideo d-dimensionale"*.

Quanto descritto può essere sintetizzato mediante la Figura 2.1.

Nel Resource Discovery un ulteriore aspetto è legato alla *dinamicità* delle risorse. Ritornando all'esempio del resource discovery, una classica interrogazione protrebbe essere la seguente: *"Trova almeno k risorse con OS='Linux'*

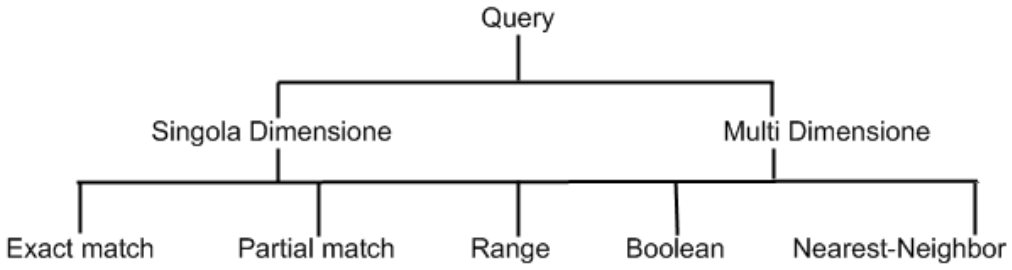

Figura 2.1: Classificazione Query

$and\ 256MB \leq FreeMem \leq 512MB\ and\ CPUWork \geq 1Ghz$". Possiamo notare la presenza di attributi quale la memoria disponibile o il carico di lavoro della CPU il cui valore è fortemente variabile nel tempo. Per ogni attributo vale la seguente classificazione:

1. **Attributi statici** la cui frequenza di aggiornanto rimane tendenzialmente costante nel tempo. Per esempio il sistema operativo, il numero di processori o la capacità di immagazinamento dati.

2. **Attributi dinamici** i cui valori variano significativamente istante per istante. A questa categoria appartengono lo stato corrente di utilizzazione del processore, la quantità di memoria fisica disponibile o la percentuale di utilizzo della rete.

La ricerca si è quindi concentrata sul potenziamento delle DHT per incrementare l'espressività delle query e gli attributi gestiti.

2.3 Classificazione sistemi P2P

Lo stile architetturale di una rete P2P risulta molto differente dal classico paradigma client/server. In particolare nelle reti P2P si vuole sfruttare al massimo le potenzialità della rete di interconnessione (Internet) basandosi su un'archi-

tettura quanto più possibile *scalabile*, *tollerante ai fallimenti* e *completamente decentralizzata*.

Possiamo analizzare le reti P2P secondo due aspetti:

1. La topologia della rete
2. L'organizzazione delle strutture dati

2.3.1 Topologia della rete

La topologia della rete indica la struttura mediante cui i peer sono logicamente interconessi. In particolare si hanno topologie di reti *strutturate* e *non-strutturate*[27].

Una rete P2P non strutturata è tipicamente descritta da un grafo in cui le connessioni dei peer sono basate sulla loro popolarità[28]. Tra le reti non strutturate, come Gnutella[1], Napster[2], BitTorrent[3], JXTA[4] e Kazaa[5], possiamo distinguere vari livelli di decentralizzazione. Il grado di decentralizzazione identifica la possibilità di effettuare in maniera distribuita una ricerca e/o recupero delle risorse.

All'interno delle reti non strutturate possiamo ulteriormente suddividere i sistemi che effettuano la ricerca in modo deterministico da quella non deterministica[29]. Nel primo caso, ogni risorsa individuata da una query viene localizzata in tempi certi. A questa categoria appartengono sia la rete Napster che quella BitTorrent. Nel secondo caso, reti come Gnutella e Kazaa mantengono un sistema di indici distribuito e per effettuare le ricerche utilizzano un modello basato sul *"flooding"* dei messaggi ai vicini della rete, in Kazaa tra super-peers. In Kazaa i super-peers raccolgono i dati ricevuti dai peer normali, effettuano una compressione delle informazioni ed eseguono il "flooding" dei messaggi di ricerca esclusivamente verso altri super-peers. Queste reti pur limitando la quantità di messaggi scambiati in rete risultano essere

ancora poco scalabili e presentare il fenomeno dei *"falsi positivi"* portando ad una perdita di accuratezza nelle operazioni di ricerca.

Analizziamo adesso le proposte di rete strutturate. Le reti CHORD[9], CAN[8], Pastry[10], Tapestry[30] e Kademlia[31] appartengono a reti strutturate in quanto utilizzano una struttura di base per la costruzione della rete. Nella maggior parte dei casi questa struttura è la Distribute Hash Table(DHT). La DHT garantisce un modello di ricerca deterministico con una complessità di ricerca definita e nella maggior parte dei casi logaritmica nel numero di nodi presenti in rete. Le differenze tra le soluzioni che adottano reti DHT consistono negli algoritmi utilizzati per la costruzione, mantenimento e ricerca nella DHT.

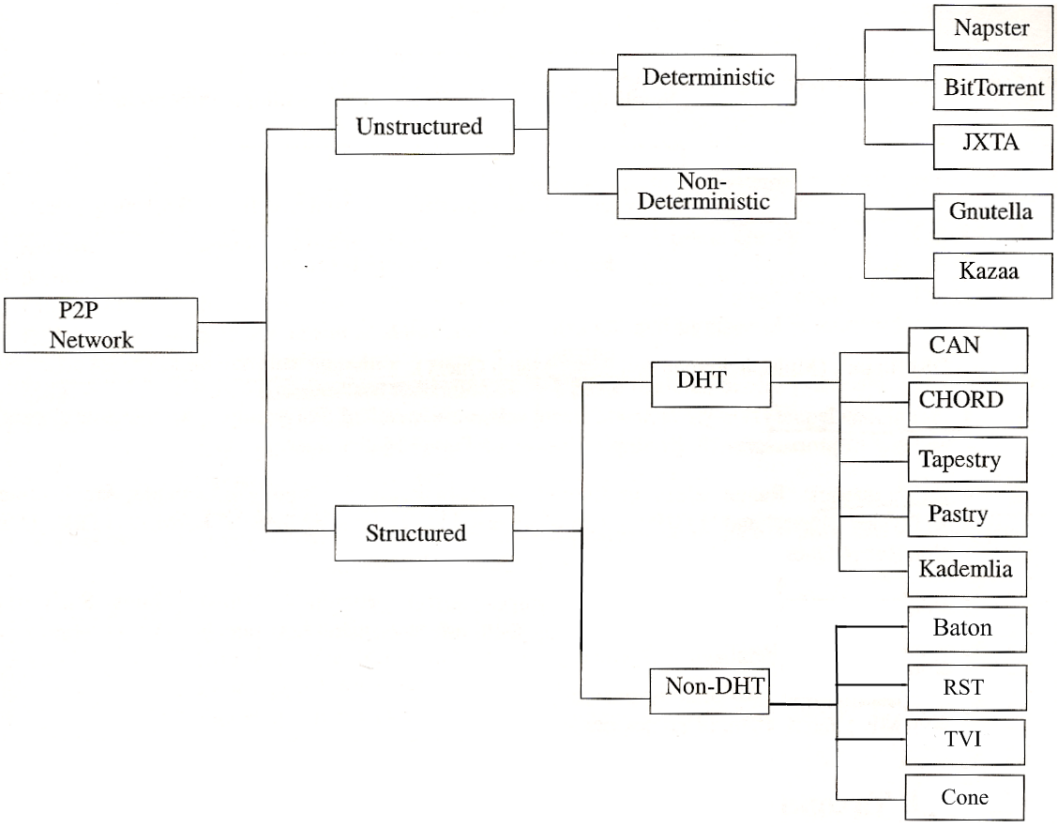

Figura 2.2: Topologia reti p2p

2.3.2 Organizzazione delle strutture dati

Le DHT costituiscono la struttura dati più utilizzata nelle reti P2P in quanto garantiscono anche un perfetto supporto a query di tipo "Exact match". Pensare di estendere gli attuali sistemi comporta necessariamente una ristrutturazione o potenziamento delle attuali strutture dati. In letteratura sono state tracciate le seguenti direzioni di ricerca:

- Utilizzo di tecniche di hashing con relative varianti
- Utilizzo di space filling curves
- Utilizzo di strutture dati basate su alberi di ricerca

Gli approcci basati sulla manipolazione della funzione hash consistono nel cercare di *preservare proprietà come l'ordine o la località dei dati*. In questo modo la risoluzione delle query può avvantaggiarsi mediante l'utilizzo di differenti euristiche. Da notare come l'utilizzo di tali tecniche possa essere critico se non opportunamente gestito in situazioni di distribuzioni dei dati non uniformi.

Le space filling curves SFC[32]-[33] sono strutture dati che effettuano una linearizzazione di uno spazio d-dimensionale in uno spazio ad 1-dimensione ($f : N^d \rightarrow N$). Una risorsa, definita da un insieme di attributi in uno spazio multidimensionale, viene posizionata in uno spazio lineare attraverso una specifica funzione di *"mapping"*. Il valore generato da tale funzione è denominato *chiave surrogata* della risorsa. Esistono varie funzioni in letteratura (Hilbert[19], di Gray o Z-Order) ma in tutti i casi devono rispettare le seguenti proprietà:

1. Copertura totale dello spazio n-dimensionale senza nessuna sovrapposizione.

2. Visita di ogni punto esclusivamente una sola volta.

3. Preservare la località dei punti: chiavi surrogate vicine provengono da punti vicini dello spazio n-dimensionale.

Quest'ultima proprietà non risulta applicabile in senso inverso, cioè non tutti i punti adiacenti nello spazio n-dimensionale risultano essere adiacenti nella curva SFC. Le SCF come mostrato in [34] presentano alcuni limiti sia nel caso di distribuzioni di dati non uniformi che nel caso di numero di dimensioni elevato (*"curse of dimensionality"*).

L'ultima categoria di strutture dati è quella basata su *alberi di ricerca distribuiti*. Questa categoria può essere ulteriormente suddivisa a seconda che si trattino dati ad una dimensione o dati multidimensionali. Nel primo caso le strutture dati distribuite ([17]-[18]) tendono a basarsi sull'utilizzo di *Prefix Hash Tree* o *B-alberi*. Nel caso multidimensionale si hanno strutture dati ispirate a alberi *"Space Driven"* come i *KD-Tree* o a strutture *"Data Driven"* come gli *R-Tree*. In tutti i casi le operazioni di ricerca non riportano falsi negativi.

Concludendo le proprietà auspicabili che tutte le strutture dati dovrebbero avere sono:

- *Bilanciamento del carico*: ogni nodo della rete in media inoltra e riceve lo stesso numero di messaggi e gestisce una quantità di dati paragonabile.

- *Minimizzazione delle informazioni di stato*: mantenere il minor numero di informazioni per la gestione delle strutture dati distribuite.

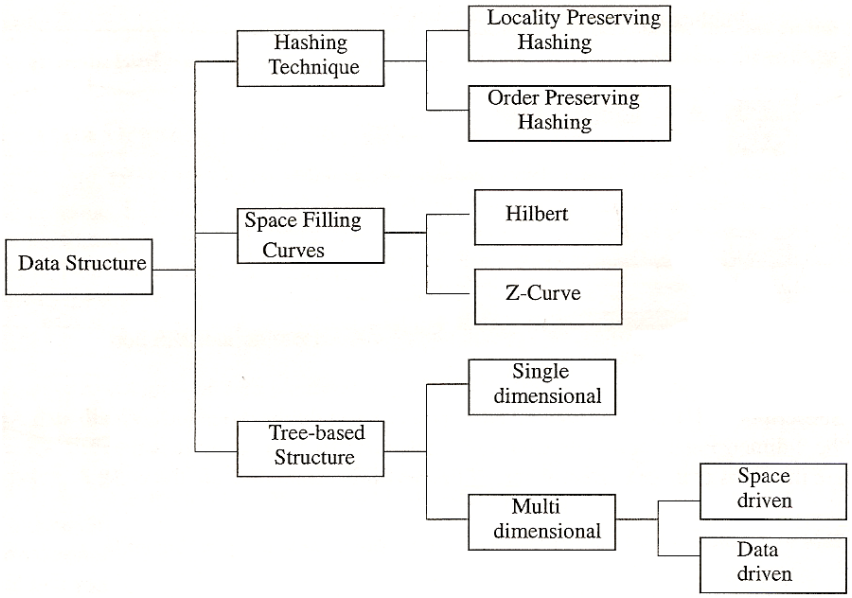

Figura 2.3: Organizzazione strutture dati p2p

2.4 Proposte in letteratura

In questa sezione verranno illustrati gli approcci principali per ognuna delle direzioni di ricerca individuate nei paragrafi precedenti.

2.4.1 MAAN

MAAN, Multi-Attribute Addressable Network, è una rete P2P strutturata in grado di fornire il supporto a range query multidimensionali[16]. MAAN utilizza Chord estendendo la DHT con una funzione hash in grado di *preservare la località dei dati.*

In MAAN le risorse sono identificate da un insieme di coppie attributo-valore $a_1 = v_1, a_2 = v_2, \ldots, a_n = v_n$, memorizzate secondo la funzione hash nell'anello Chord $[0, 2^{m-1}]$. Il processo di memorizzazione consiste quindi nell'applicare per ogni coppia la relativa funzione hash e nel memorizzare nel corrispondente nodo della rete l'intera risorsa. Da notare come una risorsa venga

registrata un numero di volte pari al numero di attributi che la compongono.

L'esecuzione di una range query multidimensionale in MAAN comporta la risoluzione di un insieme di *query unidimensionali* una per ogni coppia attributo-valore della risorsa. Una volta ottenuti i risultati, la risoluzione della range query multidimensionale consiste in una semplice intersezione. Gli autori ottimizzano il processo appena descritto proponendo il concetto di *attributo dominante*. Un attributo è definito dominante se la percentuale di nodi filtrati risulta maggiore rispetto agli altri attributi. Il nuovo processo di risoluzione di una interrogazione consisterà nel risolvere esclusivamente la query unidimensionale dell'attributo dominante. Ricordando nuovamente che ogni nodo ha memorizzato l'intera risorsa la ricerca per attributo dominante potrà risolvere localmente l'intera query multidimensionale evitando di effettuare nuovi invii di messaggi.

2.4.2 Squid

Proposto da [15], Squid è una rete P2P che consente di effettuare range query multimensionali. Squid utilizza le Space filling curves per memorizzare le risorse all'interno di un anello Chord. La funzione SFC utilizzata per preservare la località dei dati è la funzione di Hilbert[20]-[21].

Per ogni risorsa si calcola la relativa *chiave surrogata* e si procede con la memorizzazione al primo nodo dell'anello il cui ID risulta essere maggiore o uguale della chiave surrogata. La funzione di Hilbert attraverso un procedimento ricorsivo identifica ad ogni iterazione un insieme di celle. I punti centrali di tali celle sono uniti dalla curva di Hilbert (Figura 2.4). In aggiunta si definiscono *"cluster"* un insieme di celle generate ad uno specifico passo di iterazione.

In Squid l'elaborazione di una query consiste nell'effettuare due operazioni

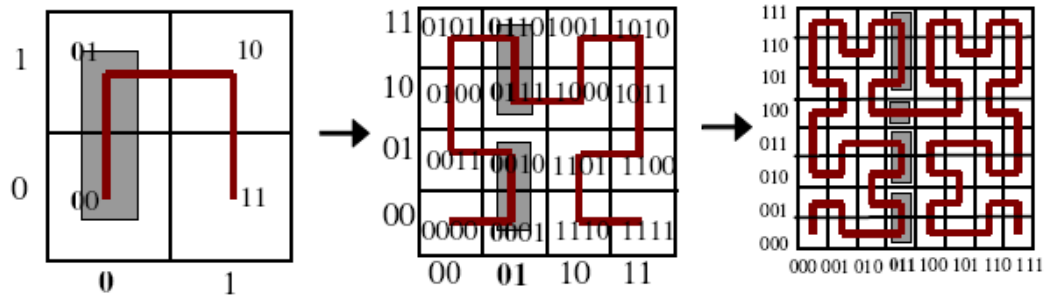

Figura 2.4: Squid Funzione di Hilbert

distinte. La prima nel tradurre la query in un insieme di *"clusters rilevanti"*, la seconda nell'interrogare i corrispondenti nodi (Figura 2.5). Un cluster è definito rilevante se al proprio interno vi sono dati.

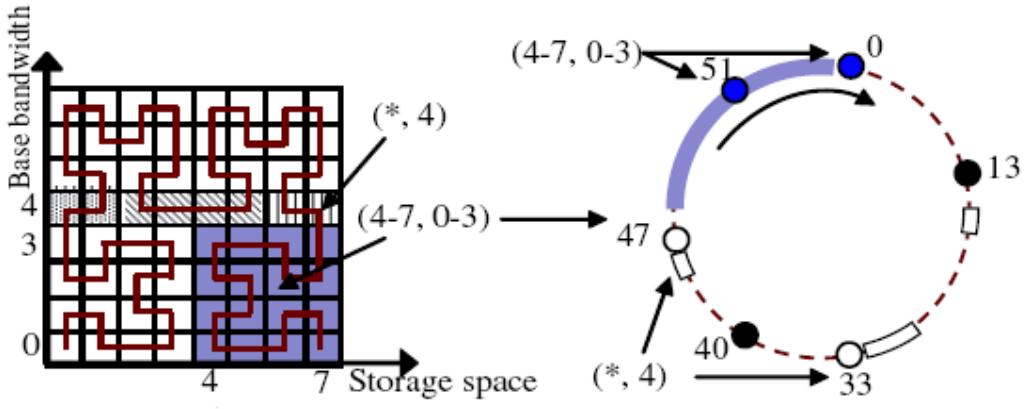

Figura 2.5: Squid Clusters

In Squid viene proposto un ulteriore passo di ottimizzazione per evitare che un numero elevato di clusters possa rendere la soluzione non scalabile. Si effettua quindi la generazione dei clusters rilevanti in maniera incrementale e guidata dai dati. Ogni cluster è identificato da un prefisso dell'albero, per poter eseguire la query i clusters vengono raffinati incrementalmente dai soli nodi che ricadono all'interno dei clusters via via raffinati (Figura 2.6).

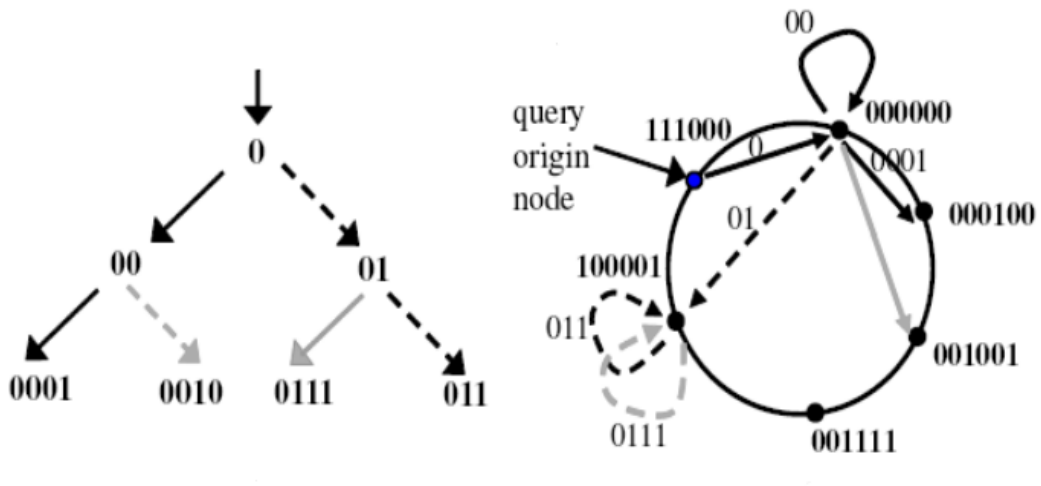

Figura 2.6: Squid Raffinazione Clusters

2.4.3 Baton

Baton, *BAlanced Tree Overlay Network*, è una rete P2P strutturata in grado di fornire supporto a range query ad una sola dimensione attraverso l'impiego di una struttura dati distribuita simile ad un B-albero[17]. Sfruttando infatti le proprietà della struttura ad albero, Baton, consente di effettuare sia *"exact match"* che *"range query"*.

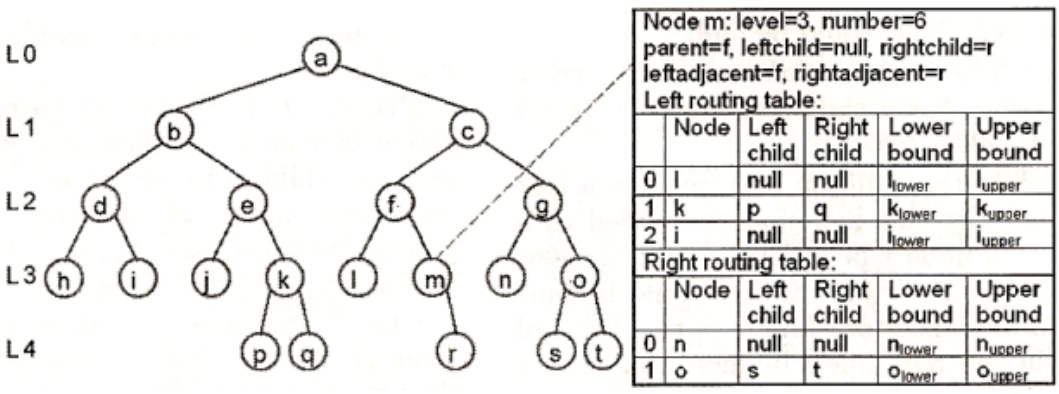

Figura 2.7: Routing Table Baton

Ogni nodo dell'albero rappresenta esattamente un peer della rete e mantie-

ne una parte della struttura dati. Gli identificativi dei diversi nodi consistono in indirizzi IP di rete. Un nodo Baton memorizza come stato interno una routing table con le seguenti informazioni: un riferimento al nodo padre, i riferimenti ai nodi figli, un insieme di nodi adiacenti seconda una visita simmetrica o *"in-order"* dell'albero, ed m nodi vicini dello stesso livello (Figura 2.7). In particolare gli m nodi vicini, a destra e a sinistra del nodo, sono selezionati secondo la seguente successione $m-2^2, m-2^1, m-2^0, m+2^0, m+2^1, m+2^2$ supponendo una numerazione dei nodi per livelli successivi a partire da sinistra. Come si può notare il procedimento risulta molto simile a quanto effettuato da Chord a meno della richiusura ad anello.

Baton è un albero binario bilanciato ma per poter effettuare delle ricerche necessita di una ulteriore struttura ad indici distribuita. Ogni nodo dell'albero possiede un intervallo di valori da gestire definito dai limiti *"up"* e *"lower"*. L'assegnazione degli intervalli viene effettuata attribuendo ad ogni nodo come valore di *"lower"* il valore massimo gestito dal sotto albero sinistro e come valore di *"up"* il valore minimo presente del sotto albero destro (Esempio in figura 2.8). Da notare come ai nodi foglia l'assegnazione dei range venga effettuata in maniera ordinata e che, a differenza dei $B^+ - aberi$, anche i nodi interni gestisco range di valori.

Illustriamo adesso attraverso l'esempio riportato in figura 2.8 come viene effettuata una operazione di ricerca. Supponiamo per ipotesi che il nodo h voglia cercare un valore gestito dal nodo c. L'algoritmo di ricerca illustrato in [17] effettua i seguenti passaggi. Il nodo h confronta il valore cercato con il proprio range determinando, attraverso la routing table, che il valore di *"up"* risulta inferiore al valore cercato. h inoltra la query al vicino destro più estremo tale per cui il relativo limite di *"lower"* risulti inferiore al valore cercato. Ricordiamo che i vicini sono memorizzati esponenzialemente sempre

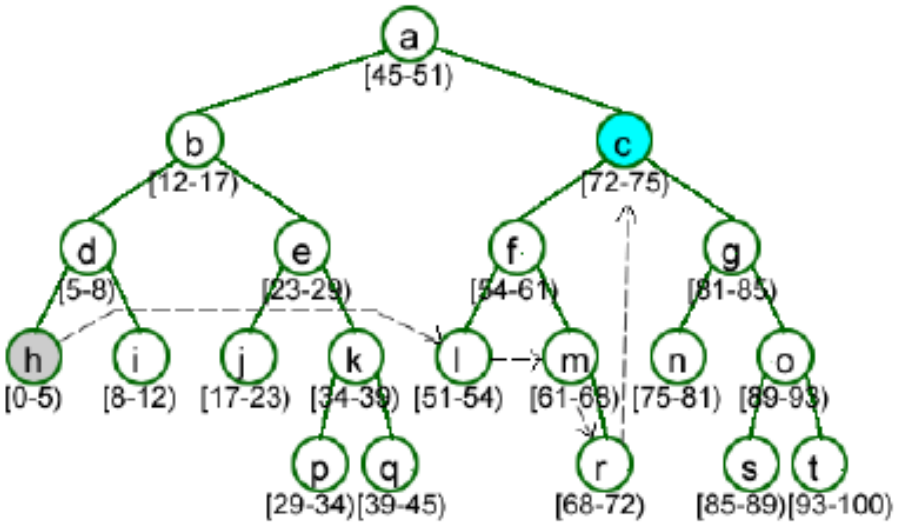

Figura 2.8: Indici distribuiti in Baton

più distanti. Il nodo l effettuando un procedimento analogo inoltra la query al vicino m. A questo punto il nodo m non avendo nella propria routing table nessun vicino destro in grado di soddisfare la condizione di limite inferiore, invia la query al figlio destro r. Il nodo r non trovando nessun vicino destro e non possedendo nessun figlio invia la query ad un nodo fisicamente adiacente, c, concludendo in questo modo l'operazione di ricerca. Da osservare come nel caso di range query l'algoritmo si comporti in maniera del tutto analoga intersecando i range esaminati.

Analizzando l'esempio possiamo notare come l'operazione di ricerca sia limitata nel numero di nodi da visitare. Infatti come mostrato dagli autori il costo di ricerca al caso pessimo risulta logaritmico e pari alla profondità dell'albero. Inoltre non vengono generati falsi negativi.

Per quanto riguarda l'operazione di join, Baton effettua una separazione della procedura in due fasi. Nella prima fase viene determinato la collocazione all'interno dell'albero del nuovo nodo. Successivamente nella seconda fase vengono effettuate le operazioni di inclusione del nuovo nodo nell'albero Baton.

Il costo dell'operazione di join è determinato dalla prima fase di ricerca. Gli autori mostrano in [17] come tale ricerca non coinvolga più di $O(\log N)$ nodi, con N il numero di nodi o peer nella rete.

In ultima analisi occorre esaminare Baton nella gestione degli aggiornamenti dei valori ed in particolare di come possano innescare il bilanciamento dell'albero. Ipotizziamo che il valore gestito da un nodo abbia una variazione oltre i limiti del range gestito localmente. Consideriamo l'esempio in figura 2.9 e supponiamo che l'aggiornamento comporti lo spostamento di un valore dal generico nodo e al nodo g (Figura 2.9). Questo spostamento può essere determinato attraverso una semplice procedura di ricerca. Supponiamo che il nodo g adesso debba gestire un elevato numero di valori e quindi risulti particolarmente carico. In questo caso occorre procedere con una suddivisione del range gestito da g ed eventualmente ad un bilanciamento dei nodi. Il caso riportato in figura 2.9 *(a)* mostra esattamente la situazione in cui è presente uno sbilanciamento del carico nel nodo g.

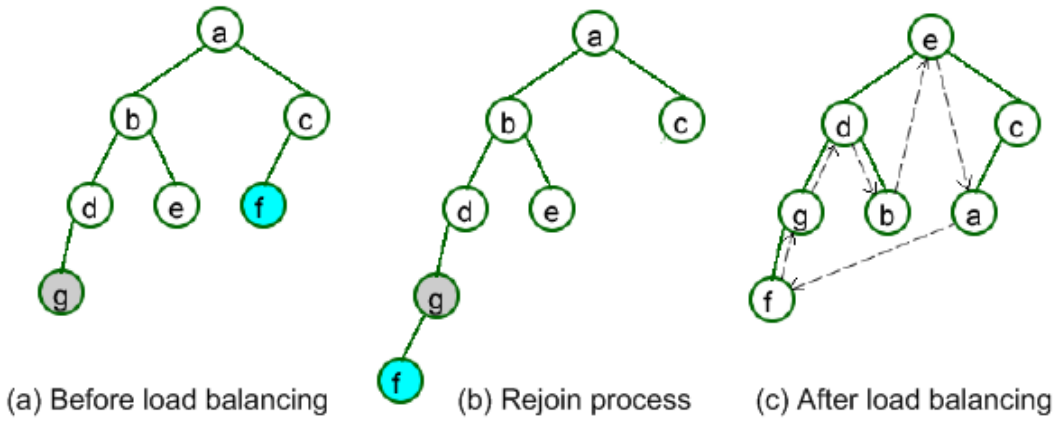

Figura 2.9: Bilanciamento del Carico

L'algoritmo di bilanciamento del carico procede nel seguente modo. Il nodo g identifica un nodo scarico presente in rete, per esempio il nodo f. Il nodo f passa il proprio range di valori al nodo padre c ed effettua un nuovo inserimen-

to come nodo figlio di *g*, a questo punto *g* divide una parte dei propri valori con *f*. (Figura 2.9 *(b)*). A seguito del bilanciato del carico l'esempio in figura illustra come sia necessario procedere anche ad un bilanciamento dell'albero. La procedura è simile a quella utilizzata negli alberi AVL. I movimenti effettuati sono illustrati in figura 2.9 *(c)* attraverso la linea tratteggiata. Il nodo *f* rimpiazza il nodo *g*, *g* a sua volta rimpiazza il nodo *d*, *d* rimpiazza il nodo *b*, *b* rimpiazza il nodo *e*, *e* il nodo *a* ed infine *a* assume la posizione originale avuta da *f*.

Il problema del bilanciamento del carico è il punto critico della proposta di Baton. Infatti anche se il range di valori assegnati ad un nodo non varia, il numero di query ricevute da un nodo potrebbe discostarsi significativamente da quelle ricevute da altri nodi. Pertanto anche in questo caso è necessario il ribilanciamento dinamico della struttura. Si noti inoltre che tale bilanciamento del carico può comportare, nel caso peggiore, ad un trasferimento totale dei valori del nodo carico al nodo scarico e coinvolgere potenzialmente tutti i nodi dell'albero. Questa situazione potrebbe ricrearsi frequentemente in presenza di attributi altamente dinamici.

2.4.4 Range Search Tree

La struttura dati *Range Search Tree* (RST) è la soluzione proposta nella tesi di dottorato da [14]. RST è anch'essa una struttura basata sul mantenimento di un albero distribuito tra i peer della rete ed in grado di fornire un supporto a range query multidimensionali.

Per illustrare l'albero RST occorre descrivere la tecnica di base utilizzata per effettuare il bilanciamento del carico tra i nodi. Una risorsa R è definita da una serie di coppie attributo/valore avente la seguente forma:

$$R = \{(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)\}$$

Ad ogni coppia (a_i, v_i) è possibile associare una matrice di bilanciamento del carico denominata *"Load Balancing Matrix"* (LBM). La LBM tiene traccia del carico relativo alle registrazioni e richieste effettuate per la coppia (a_i, v_i). In particolare organizza i peer già presenti in rete al fine di condividerne il carico.

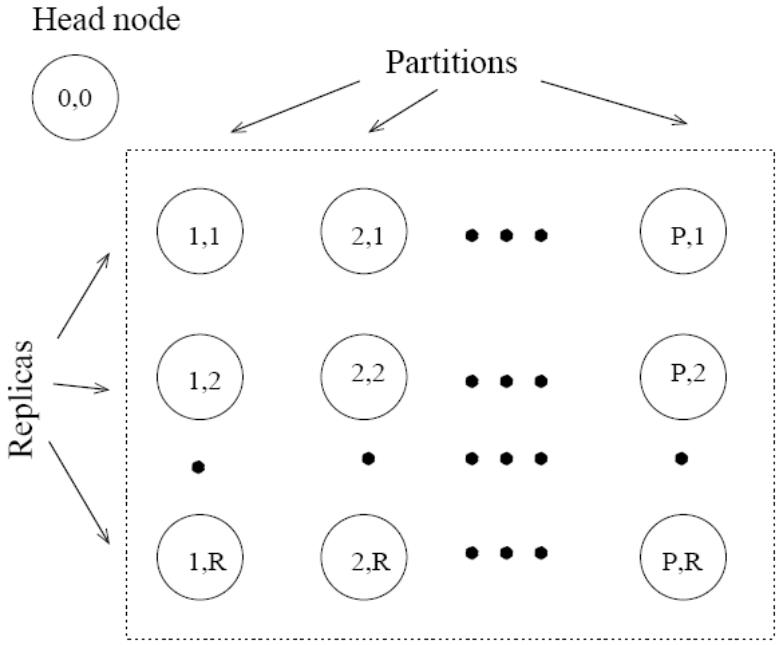

Figura 2.10: Load Balancing Matrix per (a_i, v_i)

La figura 2.10 illustra la matrice di bilancimento per l'attributo (a_i, v_i). I nodi nelle colonne memorizzano una partizione della risorsa R, ovvero un sotto-insieme delle coppie attributo-valore di cui è composta la risorsa R con il vincolo di essere sempre inclusa la coppia (a_i, v_i). I nodi di una stessa colonna consistono invece in delle repliche della stessa partizione.

La matrice LBM si espande e contrae dinamicamente nel tempo a seconda del carico di registrazioni o interrogazioni effettuate nella rete. Per tenere traccia della dimensione della LBM si ricorre ad un *"head node"* che memorizza il numero di partizioni e repliche. Descriviamo adesso come il generico nodo

possa memorizzare una risorsa R attraverso l'uso della LBM. Per ogni coppia di valori della risorsa, (a_i, v_i), vengono recuperate attraverso il *"head node"* il numero di partizioni e repliche. Successivamente con l'ausilio di soglie del carico mantenute localmente viene decisa la partizione p in cui memorizzare la risorsa R. Per determinare tutti i nodi replica r della stessa partizione viene applicata la seguente funzione:

$$N \leftarrow H((a_i, v_i), p, r)$$

La matrice LBM descritta risulta alla base della struttura Range Search Tree. Un albero RST è costruito sul dominio dei valori di un singolo attributo. Tra i nodi dello stesso livello viene *partizionato* il dominio in *intervalli di differente granuralità*. Nello specifico tutti i nodi di livello l possiedono degli intervalli di ampiezza minore rispetto ai nodi di livello *l+1* e l'unione degli intervalli di un livello ricopre sempre l'intero dominio. Viene riportato un esempio di partizionamento dell'albero in figura 2.11 *(a)* con dominio di valori definito nel range [0,7].

Illustriamo adesso le operazioni di registrazione ed interrogazione nell'albero RST. Per memorizzare una risorsa si estende l'algoritmo descritto precedentemente con la matrice LBM. Consideriamo la generica coppia (a_i, v_i) il valore v_i identifica un percorso specifico all'interno dell'albero RST denominato $Path(v_i)$. La memorizzazione della risorsa R viene effettuata esclusivamente tra i nodi del Path. Recuperando i valori della LBM l'algoritmo viene poi esteso determinato i nodo nel seguente modo:

$$N \leftarrow H((a_i, v_i), [s, e], p, r)$$

i parametri [s, e] rappresentano il range del nodo del path in cui memorizzare, i valori p ed r gli indici della relativa LBM (Figura 2.11 *(b)*).

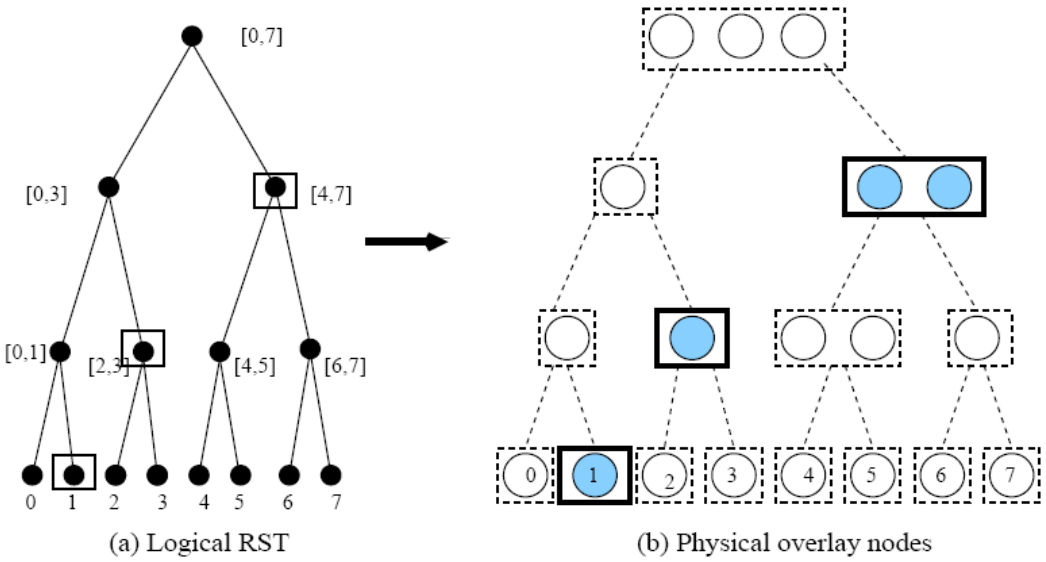

Figura 2.11: Esempio struttura RST

Questo procedimento di memorizzazione come notato degli autori risulta non ottimizzato in quanto l'attributo potrebbe essere memorizzato in un sottoinsieme dei nodi del Path. Idealmente i nodi in cui memorizzare dovrebbero essere i nodi RST i cui range siano non eccessivamente granulari ma neanche tali da concentrare le registrazioni verso pochi nodi producendo uno sbilanciamento del carico. Per ottenere questo comportamento viene introdotto il concetto di *"banda"* dell'albero RST. La banda rappresenta un sottoinsieme dei nodi di un Path, determinati dinamicamente dalla rete, in cui è conveniente memorizzare le risorse. Le informazioni sulla banda vengono mantenute in maniera simile a quanto fatto con la matrice LBM attraverso l'*head node* e il *Path Mainenance Protocol* (PMP) (Figura 2.12).

Con questa ottimizzazione la procedura di ricerca varia leggermente dovendo in via preliminare determinare la banda e successivamente memorizzare esclusivamente nei nodi del Path all'interno della banda.

Passiamo adesso ad illustrare il meccanismo di interrogazione. Nella ri-

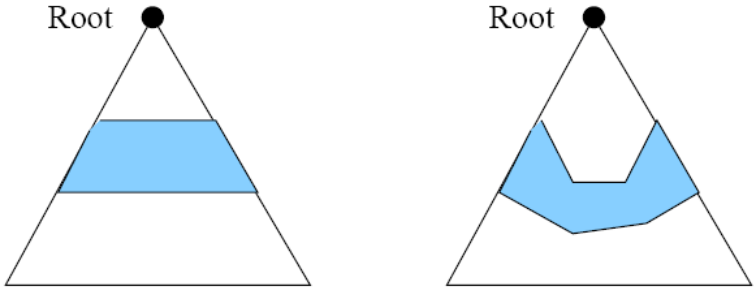

Figura 2.12: Determinazione della banda

chiesta di una range query occorre notare come esistano diversi modi per decomporre il range richiesto utilizzando l'albero RST. Pertanto l'efficienza dell'algoritmo di query è in parte legato alla determinazione di una decomposizione ottimale della query e quindi dei nodi RST da contattare. A guidare il processo di decomposizione viene introdotta una metrica chiamata *"rilevanza"*. Formalmente supponendo di avere una range query Q con intervallo [s,e], l'algoritmo di query potrà decomporre l'intervallo originario in k sub-queries corrispondenti a k nodi dell'albero RST, $N_1, N_2, \ldots, N_k$. La rilevanza r è definita come $r = \frac{R_q}{\sum R_i{}_{i=1}^{k}}$ dove con R_i si rappresenta l'ampiezza del range gestito dal nodo N_i e con R_q l'ampiezza della query originaria. Intuitivamente la rilevanza indica quanto la decomposizione effettuata corrisponda bene alla query originaria. Per ulteriori dettagli sulla determinazione ottimale dei nodi RST si rimanda all'articolo [14].

A questo punto l'esecuzione di una range query consiste nel recuperare preliminarmente le informazioni sulla banda, decomporre il range richiesto in una lista di nodi RST ed interrogare i nodi in accordo con la banda individuata e le informazioni LBM.

2.4.5 Tree Vector Indexes

Il *Tree Vector Indexes* (TVI) si presenta come una struttura in grado di effettuare range query in reti P2P altamente dinamiche. Presentata da [23] i TVI costruiscono una rete strutturata tra un insieme di peer associando le informazioni di routing direttamente tra i links di connessione dei nodi. L'esecuzione di una range query unidimensionale consisterà nell'instradare la richiesta verso i nodi target in base alle informazioni possedute in ogni collegamento.

La rete TVI come già detto è una rete P2P strutturata ed in particolare viene mantenuto un albero di copertura non orientato tra i nodi. Per descrivere la struttura occorre iniziare a definire alcuni concetti di base.

Ogni elemento è descritto da un insieme di coppie attributo valore e identifichiamo con $T = (N, E)$ l'albero di copertura non orientato definito sull'insieme di N peer. Per definizione le possibili connessioni in tale sistema sono al più N-1. Definiamo quindi con $E \subseteq \{P_i, P_j\} \mid 1 \leq i, j, \leq N$ l'insieme delle connessioni tra due nodi e con $Nb(P_i)$ l'insieme dei nodi vicini direttamente connessi $P_i \in P$.

Ogni peer P_i memorizza le informazioni riguardo ai valori posseduti attraverso un indice binario chiamato *BitIdx*. Il *BitIdx* è costruito nel seguente modo. Consideriamo un attributo A il cui dominio è definito dal range [a,b] e selezioniamo k+1 punti di divisione in modo tale da ottenere un partizionamento disgiunto $a = a_0 \prec a_1 \prec \ldots \prec a_k = b$. Per ogni possibile valore del dominio D, si definisce con $BitIdx(D[A]) = [b_0, b_1, \ldots, b_{k-1}]$ il vettore binario di k bits tale per cui:

$$\forall i \; b_i = 1 \Leftrightarrow D[A] \in [a_i, a_{i+1}) \;\; con \; i = 0, 1, \ldots, k-1$$

Il generico nodo P_i memorizzerà per ogni vicino $P_j \in Nb(P_i)$ le informazioni relative ai possibili valori raggiunti attraverso il collegamento $\{P_i, P_j\}$.

A questo punto possiamo definire come $T(P_i \rightarrow P_j)$ il sotto albero T che contiene P_j e non contiene $P_i \in Nb(P_j)$. P_i definisce la seguente quantità: per ogni attributo A con valori D in $T(P_i \rightarrow P_j)$

$$LinkBitIdx(P_i \rightarrow P_j, A) \equiv \bigvee \rightarrow P_j BitIdx(D[A])$$

che corrisponde all'operazione di unione tra i bits di tutte i BitIdx(D[A]) associato al sotto albero T. Illustriamo meglio quanto descritto con un esempio in figura 2.13.

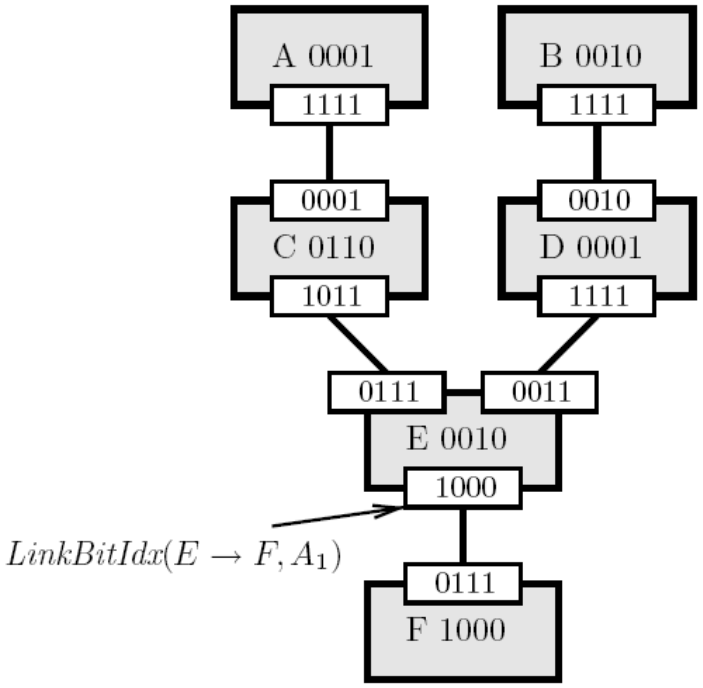

Figura 2.13: Tree Vector Indexes: un esempio

In figura si considera il singolo attributo A_1 il cui dominio è codificato in 4 bits. Il valore binario all'interno dei nodi in grigio rappresenta il *BitIdx* del valore gestito localmente. I valori binari presenti nei rettangoli bianchi rappresentano invece il $LinkBitIdx(P \rightarrow P', A_1)$. Dall'esempio si ha che il nodo E possiede un valore locale codificato con $BitIdx(D[A_1]) = 0010$ e

un valore di $LinkBitIdx(E \rightarrow F, A_1) = 1000$. Da notare come il valore di $LinkBitIdx(A \rightarrow C, A_1)$ sia l'unione bit a bit dei valori D[A_1] contenuti nei nodi B, C, D, E, F.

Descriviamo adesso l'algoritmo di query utilizzato. La procedura risulta estremamente semplice, infatti per effettuare una query viene eseguita una visita a ventaglio o *"Breadth First Search"* (BFS). Si propaga la query esclusivamente ai nodi P_i il cui relativo *LinkBitIdx* indichi la presenza di valori target. Da notare come venga evitato il fenomeno del *"flooding"* della rete. Come ultima osservazione per aggiornare il valore posseduto da ogni nodo basta ricalcolare il *BitIdx* locale ed aggiornare, solo se necessario, i *LinkBitIdx* dei nodi coinvolti.

2.4.6 Cone

Cone è una rete P2P strutturata che utilizza anch'essa una struttura dati ispirata ad un albero ed in particolare ad una struttura simile ad un Heap[22]. In Cone è possibile effettuare query del tipo *"Trova k risorse di dimensione maggiore di S"*.

Cone è costruito sopra un livello di routing che supporti la ricerca basata sui prefissi, come per esempio Chord. L'albero Cone è quindi un albero binario dei prefissi in cui le foglie vengono assegnate ai peer in maniera del tutto casuale attraverso la funzione di hashing della DHT. Ogni nodo interno N possiede il valore massimo contenuto nel sottoalbero radicaco in N, ottenuto mediante l'applicazione della *"heap-property"*(Figura 2.14). Un peer P mantiene quindi una Routing Table che memorizza una porzione consecutiva di nodi logici che corrisponde ad un percorso all'interno dell'albero. In particolare per ogni nodo logico, N, non foglia deve valere:

$$N = \begin{cases} left(N), & \text{left(N).key} \prec \text{right(N).key} \\ right(N), & \text{altrimenti} \end{cases}$$

Tale proprietà implica che il nodo N di livello $l \succ 0$ sia esso stesso anche uno dei nodi figli. Una volta collocato nell'albero Cone il nodo non varierà la sua

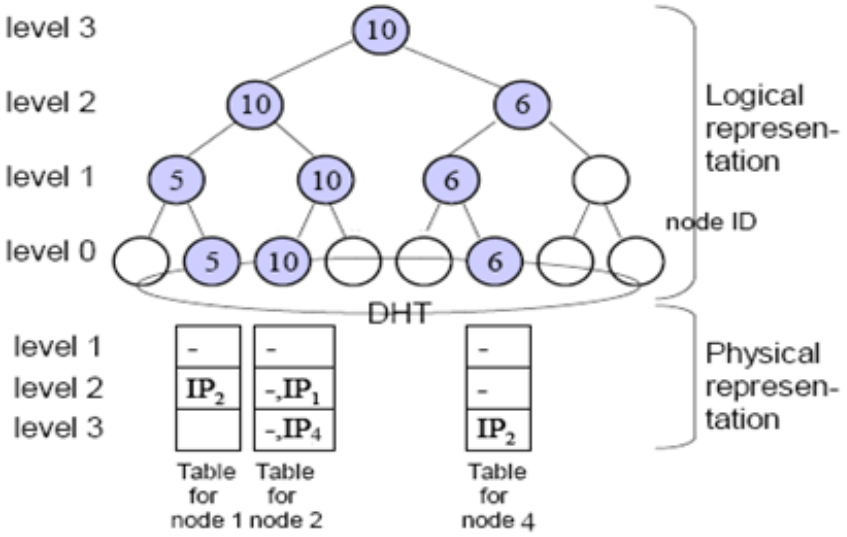

Figura 2.14: Architettura Cone

posizione durante tutta la permanenza in rete. Infatti la costruzione dell'albero Cone è determinata implicitamente dalle relazioni mantenute distribuita tra tutti i nodi nelle Routing Tables. Una variazione del valore gestito comporterà una riorganizzazione delle tabelle di routing dei soli nodi coinvolti.

La memorizzazione di un valore avviene localmente e come mostrato dagli autori in un numero di nodi logaritmico. Per effettuare delle ricerche in Cone viene sfruttata la *"heap-property"*. La ricerca infatti consiste in una risalita dell'albero attraverso l'utilizzo delle Routing Tables e sfruttando al massimo le operazioni di potatura dell'albero per ridurre l'esplorazione dell'albero. Anche in questo caso il costo dell'operazione è logaritmico.

Cone pur essendo una struttura ad albero presenta delle buone proprietà di bilanciamento del carico. Per poter analizzare la struttura occorre introdurre i concetti di *Data Traffic* e *Control Traffic*.

- **Data traffic**: rappresenta il carico di query che ogni nodo dell'albero gestisce. Dato un insieme di peer $P_1, P_2, \ldots, P_z$ aventi chiavi all'interno di un range. Si definisce $Prob_D\,(P_i)$ la probabilità che P_i soddisfi una query. Idealmente il carico risulta bilanciamento se $\forall i, j \leq z, i \neq jProb_D(P_i) = Prob_D(P_j)$.

- **Control traffic**: rappresenta il numero di messaggi di controllo scambiati dai nodi. Idealmente si ha bilanciamento del carico se il numero di messaggi scambiati dai nodi di tutto il sistema è identico. Formalmente definita $Prob_C(P_i)$ la probabilità che per qualche query un messaggio di controllo venga inviato al nodo N_i, si ha bilanciamento del carico se $\forall i, j \leq z, i \neq jProb_C(P_i) = Prob_C(P_j)$.

Analizziamo per entrambi i casi il fattore di massimo sbilanciamento tra i nodi fisici.

Analisi del Data Traffic

Iniziamo considerando due esempi in cui si verifica il maggiore sbilanciamento del carico per poi passare ad un analisi più generale.

Supponiamo di cercare una risorsa che soddisfa una determinata query. Ipotizziamo che la query possa essere risolta esclusivamente da due peer P_1, P_2 e che l'associazione tra i peer e i nodi dell'albero Cone sia quella in figura 2.15 *(a)*. Il peer P_2 è il padre al nodo logico di livello uno del peer P_1. In questo caso si ha lo sbilanciamento massimo del carico e tutte le query originate dai T-1 peer saranno dirette a P_2 mentre ad P_1 arriveranno le sole query originate da P_1 stesso. In questo caso lo sbilanciamento massimo è pari a $O(T)$, con T il numero di peer presenti nella rete.

Come caso degenere si ha la situazione illustrata in figura 2.15 *(b)*, in cui P_1 ed P_2 abbiano ottenuto lo stesso identificativo dalla DHT e quindi siano

stati assegnati alla stessa foglia. In questo caso P_1, per esempio, risponderà alle query originate da se stesso mentre lascerà ad P_2 la gestione di tutte le altre. Lo sbilanciamento anche in questo caso risulta $O(T)$.

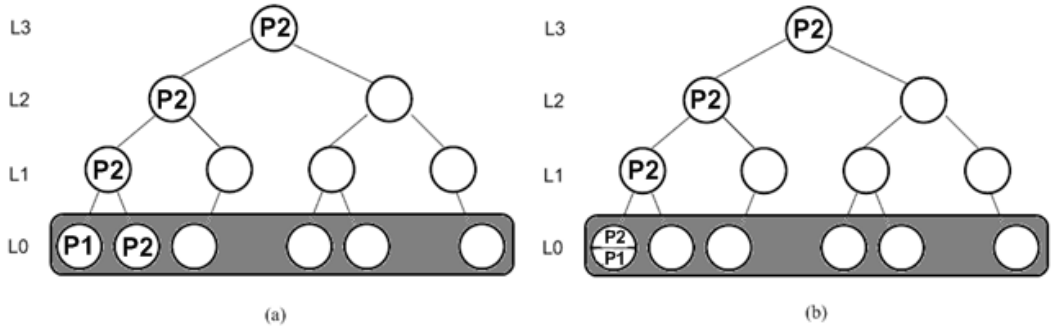

Figura 2.15: Casi di massimo sbilanciamento

Fortunatamente la probabilità che avvenga una tale configurazione per entrambi gli scenari risulta poco probabile. Infatti sia nel caso in cui i due nodi risultino adiacenti sia nel caso in cui abbiano generato lo stesso ID, la probabilità che questo accada è pari a $\frac{1}{2^h}$. Tale probabilità è calcolata assumendo che il numero di nodi fisici in rete sia esattamente uguale al numero di foglie nell'albero, per valori inferiori di nodi la dimostrazione può essere effettuata in maniera analoga.

A questo punto possiamo analizzare il caso medio. Consideriamo il caso in cui P_1 sia adiacente ad P_2 attraverso un sotto-albero di altezza k e che P_2 sia il padre dal livello k fino alla radice (Figura 2.16).

L'assegnamento del relativo ID ai nodi è effettuato dalla funzione hash della DHT e la probabilità che P_1 sia esattamente in un sotto-albero adiacente di altezza k è di $\frac{2^k}{2^h}$. Pertanto la probabilità che due nodi P_1 ed P_2 diventino adiacenti in un sotto-albero di altezza k è di $\frac{1}{2^{h-k}}$. Nello specifico P_2 diventerà il nodo padre di livello k. Tenendo conto di quanto detto possiamo definire le seguenti quantità:

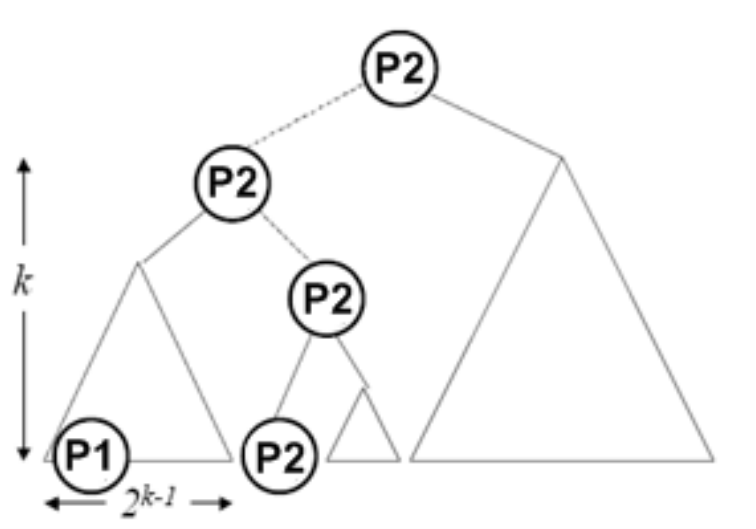

Figura 2.16: Caso generico del Data Traffic

- $Prob_D(P_1) = \frac{2^k}{2^h}$, la probabilità che P_1 soddisfi una determinata query.
- $Prob_D(P_2) = 1 - \frac{2^k}{2^h}$, la probabilità che P_2 soddisfi una determinata query.

Il rapporto di Data Traffic tra i due nodi risulta $r = \frac{Prob_D(P_1)}{Prob_D(P_1)} = \frac{2^h - 2^k}{2^k}$. Fissato k, lo sbilanciamento dell'albero risulta essere di $\left(\frac{1}{2^h - 2^k}\right)\left(\frac{2^h - 2^k}{2^k}\right) = \frac{\left(2^h - 2^k\right)}{2^h}$. Quindi per calcolare lo sbilanciamento al caso generico occorre sommare per tutti i possibili valori di k:

$\frac{2^h-1}{2^h} + \sum \frac{\left(2^h - 2^k\right)}{2^h}_{k=0}^{h-1} = \frac{2^h-1}{2^h} + \frac{1}{2^h}\left(\left(2^h - 1\right) + \left(2^h - 2\right) + \ldots + \left(2^h - 2^{h-1}\right)\right) = 1 - \frac{1}{2^h} + h + \sum_{k=1}^{n} \frac{1}{2^k} = h$

Ottenendo quindi che al caso medio, per la ricerca di una risorsa, il rapporto tra il peer selezionato il maggior numero di volte e quello selezionato il minor numero risulta non superiore a $h = \log(T)$.

Analisi del Control Traffic

Per poter effettuare l'analisi Control Traffic occorre fissare la seguente ipotesi: *"IP: La distribuzione delle chiavi possedute dai peer è identica alla distribuzio-*

ne delle query effettuate, ed entrambe le distribuzioni sono continue"

Questa ipotesi risulta necessaria per poter sfruttare le proprietà di simmetria e mettere in relazione la distribuzione delle chiavi con la distribuzione delle query. Consideriamo l'esempio in Figura 2.16 la probabilità che una query originata dal nodo P_1 arrivi al nodo P_2, situato a livello k, è uguale alla probabilità che il valore massimo cercato dalla query sia il più grande valore tra tutti i valori gestiti nei 2^{k-1} peer del sotto-albero.

Questo equivale a prelevare tra i $2^{k-1} + 1$ (l'uno in più rappresenta la query) campioni dalla distribuzione delle query e stimare la probabilità che il valore richiesto sia esattamente il più grande valore tra tutti quelli gestiti nel sotto-albero. Dato che ogni elemento del campione per ipotesi di simmetria, possa essere il valore massimo con uguale probabilità, allora la probabilità di selezionare il valore massimo risulta di $\frac{1}{2^{k-1}+1}$.

Ritornando all'esempio in figura 2.16 il numero di nodi dal quale la query può arrivare ad P_2 è di 2^{k-1}. L'indice k-1 è dato dal fatto che in Cone un peer che gestisce il nodo P_2 deve necessariamente gestire anche uno dei suoi figli, nell'esempio il figlio destro. Il carico complessivo di query che potrà raggiungere P_2 è quello proveniente dal solo sotto-albero di livello k-1. Quindi i messaggi di controllo scambiati a seguito di una interrogazione sono pari a $2^{k-1}\frac{1}{2^{k-1}+1}$.

Possiamo concludere che al caso pessimo il fattore di massimo sbilanciamento del Traffic Control è rappresentato dal caso in cui P_2 sia situato nella radice dell'albero Cone, quindi:

$$\sum \left(\frac{2^{k-1}}{2^{k-1}+1}\right)_{k=1}^{h} \prec h = \log(T)$$

2.5 Valutazioni

Le proposte illustrate presentano degli indubbi vantaggi in termini di aumento di espressività nel linguaggio d'interrogazione ma occorre prestare attenzione agli elementi caratterizzanti e agli aspetti critici.

Nel caso delle soluzioni basate sulla manipolazione delle funzioni hash come in MAAN, la scelta della funzione hash risulta il fulcro centrale del modello prestazionale che nel caso di distribuzioni dei dati non uniformi è estremamente critico. A questo aspetto si aggiuge l'ulteriore overhead che MAAN possiede nella presenza di risorse dinamiche dovendo memorizzare tutti gli attributi della risorsa.

Nell'approccio presentato da SQUID le risorse sono state preventivamente trasformate da una funzione di *"mapping"*. Come al caso precedente SQUID, memorizzando le risorse direttamente sull'anello Chord, risulta essere sensibile alle distribuzioni non uniformi. In aggiunta si anche la presenza della nota problematica *"curse of dimensionality"* che rende critica la gestione della struttura dati al crescere del numero di dimensioni dello spazio originario.

Infine negli ultimi approcci (Baton, RST, TVI, Cone) si è indagato su strutture basate sulla costruzione di una rete strutturata ispirata agli alberi. Queste soluzioni dal mio punto di vista presentano degli aspetti fondamentali da analizzare *(i)* il bilanciamento del carico, *(ii)* la gestione degli attributi dinamici e *(iii)* l'overhead di gestione. Nello specifico, l'approccio Baton mostra i suoi limiti sia con la presenza di attributi dinamici che con un elevato overhead di gestione in caso di bilanciamento della struttura. Infatti in Baton la variazione di un attributo può determinare uno sbilanciamento dei nodi e quindi innescare delle procedure di riorganizzazione della struttura. Nella struttura RST dal mio punto di vista i limiti maggiori si hanno sia nella scarsa scalabilità del sistema basato nel recupero di informazioni essenziali attraverso *"head node"*

che negli elevati costi di mantenimento dinamico delle LBM e banda. Infine la proposta Tree Vector Indexes risulta essere abbastanza promettente per quanto riguarda la gestione degli attributi dinamici ma tralascia il problema di sbilanciamento del carico definendo nella struttura della rete un generico albero di copertura tra nodi.

L'ultimo approccio, Cone, risulta molto promettente sia per il bilanciamento del carico sfruttando le proprietà di distribuzione dei dati attraverso (funzione hash) che nel mantenimento della struttura a seguito dei cambiamenti di valori. Per contro Cone presenta una scarsa espressività sulle query effettuabili.

Lo studio della letteratura ha quindi mostrato come ogni proposta presenti soluzioni sub-ottime al problema del Resource Discovery difficili da coniugare. Questa tesi mette proprio l'accento sulla gestione degli attributi dinamici e di come possano essere interrogati attraverso range query. In questo contesto la tesi si propone di generalizzare Cone, il quale offre una gestione efficace degli attributi dinamici, attraverso il supporto di un linguaggio d'interrogazione più espressivo quali le range query.

Capitolo 3

XCone: Architettura

In questo capitolo verrà presentata l'architettura di XCone. Dopo averne illustrato le principali caratteristiche si analizzeranno le operazioni di inserzione, cancellazione e di ricerca.

3.1 Introduzione

XCone si propone come estensione del lavoro presentato da [22] e discusso nel capitolo 2 al fine di supportare pienamente le range query in sistemi P2P. Attraverso una struttura ad albero distribuita, costruita al di sopra di una qualunque DHT, è possibile definire una procedura di aggregazione delle risorse tale per cui sia efficiente rispondere a query del tipo: *"Trova K risorse tale per cui il valore X sia* $V \leq X \leq S$*"*.

In XCone effettuare un aggiornamento di un valore risulta particolarmente efficiente e le operazioni di mantenimento e bilanciamento del carico risultano facilitate dal supporto fornito dalla DHT sottostante.

3.2 L'albero di mapping

La struttura dati alla base di XCone è un albero binario in grado aggregare un insieme di risorse secondo una specifica funzione di mapping. Tale albero

Notazione	**Descrizione**
T	Numero di peer nella rete
P, R, U	Peer generici della rete
S, V, X	Valori gestiti dai peer
Q	Query
K	Numero di peer richiesti
N, M, O	Nodi logici dell'albero XCone
h	Altezza dell'albero XCone
l	Livello generico dell'albero XCone
f	Funzione di mapping
d	Funzione di digest delle chiavi
m	Numero di bits della DHT

Tabella 3.1: Notazioni utilizzate

binario definisce un *trie* basato sullo spazio degli identificativi della DHT. I nodi dell'albero XCone sono definiti *nodi logici* e sono mantenuti in maniera distribuita tra i *nodi fisici* o *peer* della rete. I nodi logici foglia sono assegnati univocamente ai nodi fisici attraverso un procedimento di integrazione con la DHT che illustreremo, mentre, per tutti i nodi logici non foglia l'associazione con i nodi fisici è effettuata attraverso la *funzione di mapping* f.

La funzione di mapping determina il numero di nodi logici non foglia assegnati ad ogni peer della rete e nel caso pessimo il numero di nodi logici assegnati ad un peer è inferiore a h, pari al cammino dalla foglia alla radice dell'albero. Formalmente dati due nodi logici fratelli N ed M di livello *l-1*, assegnati rispettivamente a due peer differenti P ed R, la funzione di mapping $O \leftarrow f(P, R)$ assegna al nodo logico O di livello l il nodo fisico gestore (Figura 3.1). Appare chiaro come tale funzione costituisca un ruolo centrale nella costruzione dell'albero.

E' possibile definire diverse strategie per il mapping dei vari nodi logici del trie ai peer. In linea generale ogni nodo fisico può gestire un qualsiasi nodo logico ma per ottenere buone prestazioni nelle operazioni di join nella rete e

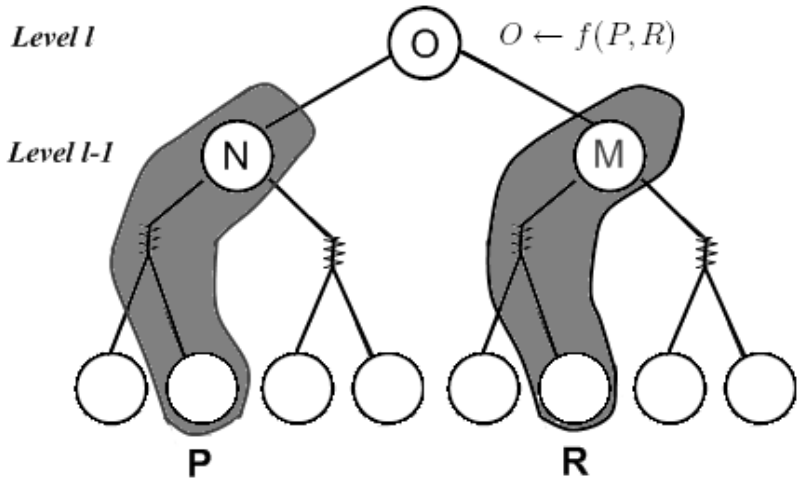

Figura 3.1: Funzione di mapping

di risoluzione delle query, occorre definire dei vincoli.

Il primo vincolo riguarda il fatto che ogni peer può gestire solo nodi logici che appartengono esclusivamente al cammino all'interno del trie determinato dal proprio identificativo. Questo consente di implementare in modo efficiente la fase di join in quanto il peer che si unisce nell'albero XCone per la ricerca dei nodi logici da gestire, deve risalire l'albero a partire dalla foglia assegnata evitando di visitare diversi sotto-alberi. Chiaramente l'assegnamento dei nodi foglia ai peer è univoco mentre per i nodi logici interni, che condividono il proprio prefisso con più foglie, possono essere assegnate ad uno qualunque dei peer che corrispondono a tali foglie.

Un ulteriore vincolo nella definizione della funzione di mapping è il seguente. Supponiamo che un peer gestisce la foglia identificata dall'identificatore ID. Allora P gestirà un insieme di nodi logici che corrispondono ad un suffisso del proprio ID. In altri termini il nodo P gestisce una sequenza continua di nodi logici a partire dalla foglia da lui gestita fino ad un certo livello l dell'albero. Questa scelta consente di ridurre durante le operazioni di ricerca il numero di hops tra peer diversi affettuati da una query nella risalita dell'albero XCone.

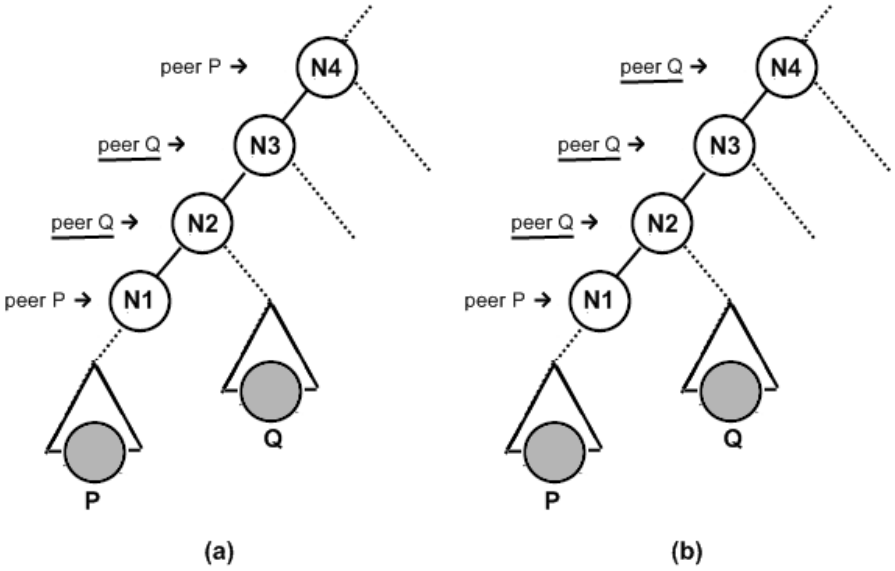

Figura 3.2: Proprietà funzione di mapping

La figura 3.2*(a)* mostra una configurazione dell'albero in cui una query che risale l'albero XCone deve essere inoltrata da P a R per poi ritornare nuovamente a P, poichè P non gestisce nodi logici consecutivi nel cammino verso la radice. In Figura 3.2*(b)*, caso in cui la funzione di mapping possiede le proprietà desiderate, P esplora inizialmente tutti i sotto-alberi radicati nei nodi logici gestiti da P stesso e successivamente solo dopo aver verificato che la query non è risolvibile in tali sotto-alberi, inoltra la query a R. Questa strategia di mapping riduce quindi il numero di hops tra peer diversi effettuati per la risoluzione della query.

XCone definisce un'altra funzione, la *funzione di digest d*, che consente di associare ad ogni nodo interno N dell'albero XCone una struttura che sintetizza le chiavi contenute nel sottoalbero radicato in N. Diverse possibili funzioni di digest verranno discusse nel paragrafo 3.4.

Analizzando attentamente quanto definito in Cone, descritto nel capitolo precedente, la funzione di mapping associa ad ogni nodo logico interno N il

peer tra quelli presenti nell'albero radicato in N che possiede la chiave di valore massimo. Inoltre in Cone, la funzione di mapping coincide con quella di digest, nel senso che quando due peer si *"combinano"* al livello *l-1* dell'albero vengono effettuate le seguenti operazioni:

1. la funzione di digest calcola il valore massimo tra le chiavi gestite da due peer. In questo modo si sintetizzano un insieme di chiavi attraverso il loro valore massimo.

2. la funzione di mapping assegna il nodo logico padre dei due nodi di livello *l-1*, al peer che possiede la chiave con valore maggiore.

Questa strategia di mapping risulta semplice e rispetta le caratteristiche descritte in precedenza. Infatti poichè i valori presenti sul cammino da un nodo foglia alla radice risultano crescenti, ogni peer P gestirà un suffisso di tale cammino contenente i nodi radice di sottoalberi con chiavi inferiori o uguali a quella posseduta da P.

E' possibile naturalmente definire ulteriori funzioni di mapping in grado di valutare aspetti come il carico dei nodi logici gestiti da ogni nodo o particolari caratteristiche come ad esempio la tipologia di banda. Una semplice strategia che consente di bilanciare il carico, definito come il numero di entrate significative presenti nella Routing Table di ogni nodo è la seguente. Quando un nuovo peer P si inserisce in XCone può individuare se sul cammino della foglia che gli è stata essegnata alla radice esistono nodi logici mappati a peer carichi. In quel caso P prenderà in gestione alcune entrate della tabella di Routing Table di tali nodi. Questa situazione comporta un nuovo aggiornamento del mapping di alcuni nodi logici ai peer presenti sul cammino di P dalla foglia alla radice. La proprietà di suffisso indicata in precedenza può essere mantenuta se ogni peer cede esclusivamente una sequenza di entrate della propria tabella di

routing corrispondente ai livelli più alti del cammino da lui gestito. Nel caso di XCone la funzione di mapping adottata coincide con quella di Cone basata sulla scelta del peer avente in gestione la chiave di valore massimo ma, come vedremo in seguito, XCone definisce funzioni di digest diverse rispetto a Cone per supportare più efficacemente le range query.

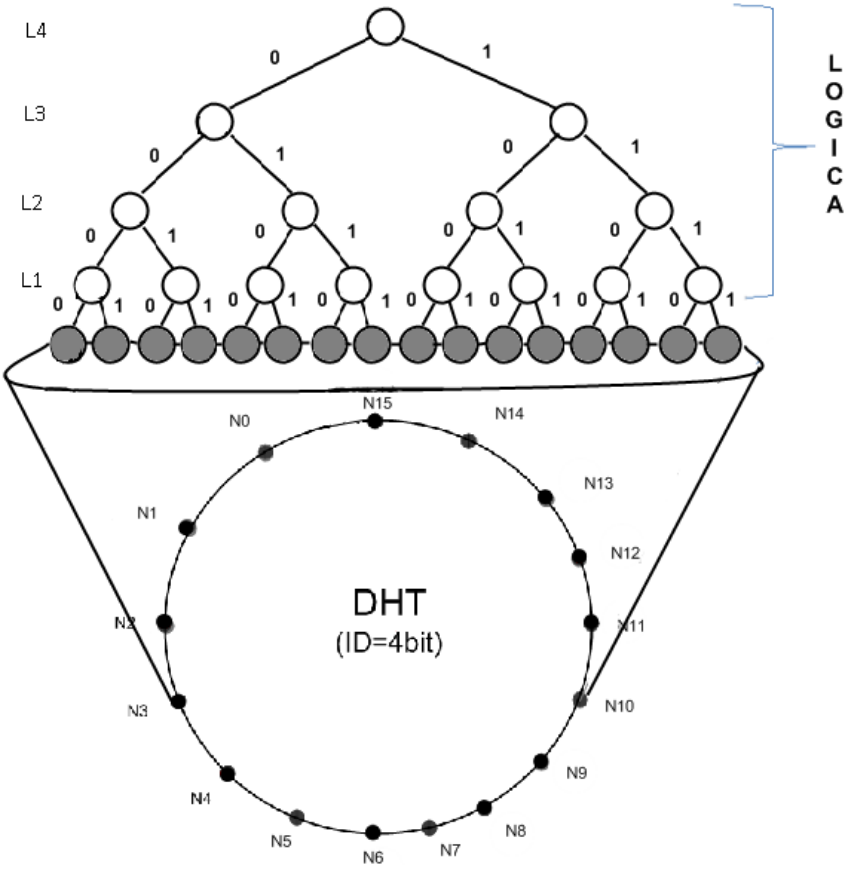

Figura 3.3: Overlay XCone-DHT

Passiamo adesso ad illustrare come viene effettuato l'assegnamento dei nodi logici foglia. L'assegnamento tra i peer della rete e nodi foglia XCone avviene attraverso l'integrazione con la rete DHT sottostante. Consideriamo una rete DHT con spazio degli identificativi (ID) pari ad m bit, i nodi logici foglia sono assegnati ai peer attraverso un *trie*. Ad ogni peer, al momento dell'ingresso

nella rete DHT, viene assegnato un identificativo ID di m bits. Poichè l'albero XCone definisce un trie basato sui prefissi degli identificatori, ad ogni peer può essere associato una determinata foglia all'interno del trie.

La figura 3.3 illustra attraverso un esempio il mapping tra nodi logici foglia dell'albero XCone e i peer di una rete DHT CHORD con identificativi di 4bits. Per chiarezza l'albero di mapping XCone riporta in ogni ramo il valore del bit associato al prefisso. L'associazione foglia/peer rimane fissata per tutta la permanenza del peer in rete. Tale associazione eredita la proprietà di assegnazione casuale garantita da una *funzione hash* impiegata dalla DHT per assegnare i peer nella rete. Infine una volta effettuato l'ingresso nella rete DHT e costruito l'intero albero di mapping XCone, ogni peer gestirà la propria chiave in maniera indipendente dalla DHT. Si ha pertanto che il dominio delle chiavi è distinto dal dominio degli identificativi della DHT.

3.3 Tabelle di Routing

Illustriamo adesso la struttura dati che consente di mantenere l'albero di mapping distribuito tra i peer della rete. Ogni nodo fisico XCone memorizza nodi logici da esso gestiti attraverso una struttura dati denominata Routing Table. La Routing Table è composta quindi al massimo da m entrate, una per ogni livello dell'albero, a parte il livello 0. La generica entrata E_l memorizza il risultato dell'applicazione della funzione di mapping f tra due nodi logici adiacenti di livello *l-1*. Tale risultato è memorizzato in una coppia (IP_1, IP_2) avente il seguente significato:

- IP_1: indirizzo del peer che gestisce il nodo logico di livello l.
- IP_2: indirizzo del peer che gestisce il nodo logico assegnato come figlio di livello *l-1*.

Da notare come le informazioni memorizzate siano dei riferimenti diretti ad indirizzi di rete dei peer. Questa è una scelta progettuale che consente di evitare un ulteriore livello di indirezione rappresentato dal passaggio per la rete DHT. In altri termini vengono velocizzate le operazioni di comunicazioni evitando inutili ricerche nella rete DHT.

Per illustrare meglio quanto detto consideriamo l'esempio di un peer P che gestisce il nodo logico N di livello *l-1* e il peer R che gestisce il nodo logico M, sempre dello stesso livello (Figura 3.4).

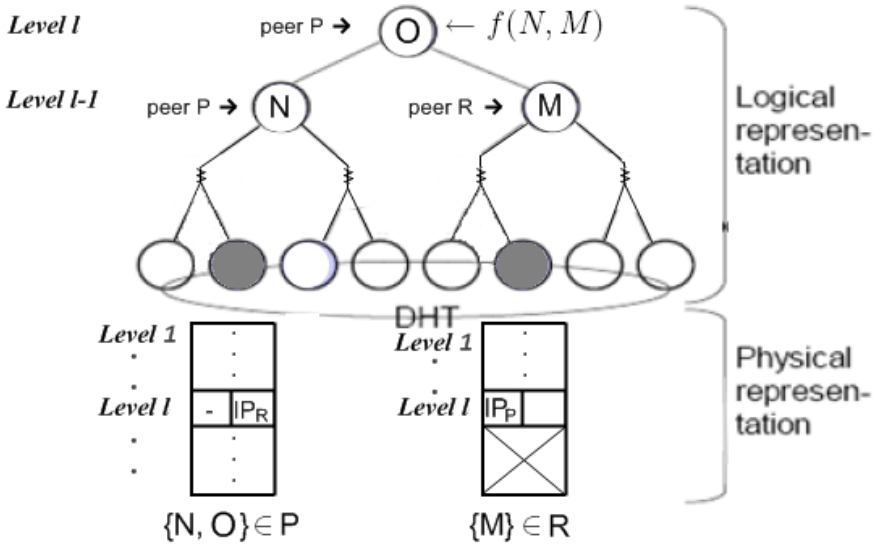

Figura 3.4: Costruzione Routing Table

Il peer P, a seguito dell'applicazione della funzione di mapping $f(P, R)$, risulta dover gestire anche il nodo logico O di livello l. Il peer P inserirà la coppia $(-, IP_R)$ nella l-esima entrata della tabella propria tabella di routing. Il simbolo "$-$" indica un riferimento al peer P stesso e il valore IP_R indica l'indirizzo IP del peer R che gestisce il nodo logico M, cioè il figlio al livello l-1. Mentre nel peer R l'entrata di livello l conterrà solo il primo dei due valori (IP_P) riferimento al peer gestore del nodo logico O. Le entrate successive

della Routing Table di R risulteranno non significative, in quanto R non può gestire nodi di livello superiore sul cammino da R alla radice mentre quelle relative a P dipenderanno dall'applicazione della funzione di mapping lungo il cammino nell'albero XCone che parte da P alla radice. A questo punto occorre definire il concetto di nodo padre di un nodo fisico R e il relativo concetto di livello del nodo padre. Se consideriamo l'esempio illustrato precedentemente il nodo padre di R risulta definito come il nodo fisico riferito nell'ultima entrata significativa della routing table di R. Nel caso del livello del nodo padre di R si determina come il livello del primo nodo logico non gestito da R.

La struttura fino a questo punto delineata risulta simile a quanto presentato in Cone[22], tuttavia tale struttura non consente di effettuare range query in maniera efficiente. Infatti l'assenza di informazioni sulla distribuzione delle chiavi contenute nei sotto-alberi porta ogni peer all'eventualità di non poter decidere con sufficiente certezza l'esplorazione o meno di un sotto-albero. Questo comporamento introduce inevitabilmente un degrado delle performance di ricerca. Occorre quindi colmare questa mancanza di informazioni adottando una funzione di digest in grado si fornire ad ogni peer una maggiore conoscenza sulla distribuzione delle chiavi presenti nei propri sotto-alberi.

Prima di passare alla definizione di tale estensione focalizziamoci meglio con un esempio sulla problematica osservata. La funzione di mapping fornisce le informazioni sulle relazioni tra nodi logici e nodi fisici e abbiamo visto come in XCone tale funzione coincida ad ogni livello di aggregazione con la funzione di digest. Le chiavi possedute dai peer collocati in un sotto-albero sono necessariamente inferiori o uguali al valore della chiave presente del nodo fisico radice logica di tale sotto-albero.

La figura 3.5 illustra attraverso un esempio un potenziale caso di deterioramente delle performance di ricerca. Assumiamo che nel sotto-albero marcato

in grigio siano presenti un insieme di chiavi comprese nel range [21-50], e che la query da risolvere sia Q = *"Find 3 values with* $10 \leq X \leq 20$ *"*. L'operazione di esplorazione del sotto albero radicato nel nodo M risulta non decidibile. Il peer P associato al nodo logico O di livello l è a conoscenza che tutti i valori presenti nel sotto-albero radicato nel nodo logico M sono necessariamente inferiori o uguali alla chiave 50.

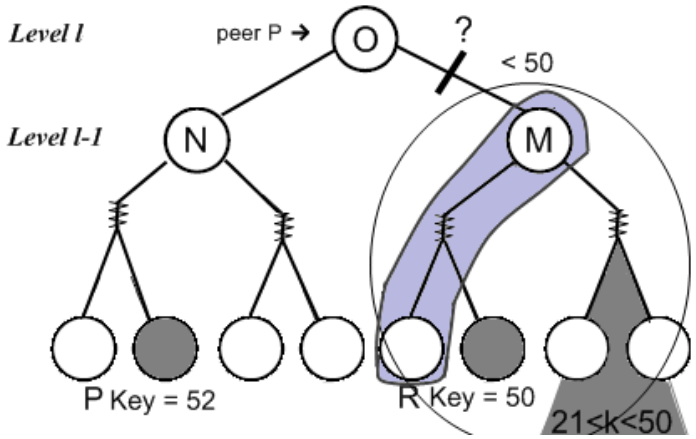

Figura 3.5: Problema esplorazione in Range Query

Le informazioni possedute da P non consentono di escludere a priori l'esplorazione del sotto-albero. Infatti la mancanza di un limite inferiore alla chiavi contenute nel sotto-albero o più in generale la mancanza della conoscenza della distribuzione delle chiavi, potrebbe portare ad una visita infruttuosa o ad una perdita di nodi target nel caso di esplorazione o meno del sotto-albero.

A questo punto appare chiaro come sia necessario ampliare le informazioni aggregando nel migliore dei modi le informazioni sulle chiavi presenti nei sotto-alberi. La generica entrata della Routing Table E_l e quindi così estesa:

- IP_1: indirizzo del peer che gestisce il nodo logico di livello l.
- IP_2: indirizzo del peer che gestisce il nodo logico assegnato come figlio di livello *l-1*.

- ST: informazione sulle chiavi presenti nei peer associati al sotto-albero radicato nel nodo figlio.

In XCone il calcolo dell'informazione di aggregazione è stato reso quanto più modulare possibile introducendo il concetto di funzione di digest d, $ST \leftarrow d(U)$ con U nodo fisico radice del sotto-albero logico dell'insieme di chiavi da aggregare.

3.4 Funzioni di digest

La funzione di digest ha il compito di aggregare un insieme di valori in una informazione di digest che richiede spazio di memoria limitato e che cattura nel migliore dei modi l'informazione sulla distribuzione delle chiavi.

La prima funzione analizzata in XCone prende spunto dalla proposta presentata da [23], i *"BitVector Indexes"*. Definiamo un vettore binario di k bits *BitIndex* in maniera tale da catturare la distribuzione dei valori di un attributo A, definito in un intervallo [a, b], nel seguente modo. Partizioniamo il dominio di A in *k-1* punti ottenendo k intervalli di separazione $a = a_0 \prec a_1 \prec \ldots \prec a_k = b$. Per ogni possibile istanza di $\{A = v\}$ definiamo il vettore binario *"BitIdx"* di k bits nel seguente modo:

$$BitIdx(A) = \begin{cases} 1, & se\ v \in [a_j, a_{j+1}]\ \text{con } 0 \leq j \leq k-1 \\ 0, & altrimenti \end{cases}$$

Il *BitVector Indexes* è ottenuto mediante l'unione (OR) di tutti i rispettivi *BitIdx* assegnati a delle risorse da aggregare. In XCone tale struttura viene definita attraverso la seguente funzione di digest:

$$d(U) = BitVector(U) = (BitIdx(U.key)\ \text{OR}\ E_1[ST] \ldots \text{OR}\ E_l[ST])$$

con U si riferisce il peer, U.key la chiave gestita dal peer U ed con E_l le rispettive entrate della routing table di U. Il nodo fisico U effettua come

operazione preliminare il calcolo del *BitIdx* relativo alla chiave da esso gestita successivamente, per ogni entrata significativa della propria Routing Table, effettua l'operazione di OR tra le informazioni contenute nel campo ST. Il risultato dell'operazione rappresenta l'informazione di digest che sintetizza le chiavi presenti nell'albero logico radicato nel nodo fisico U.

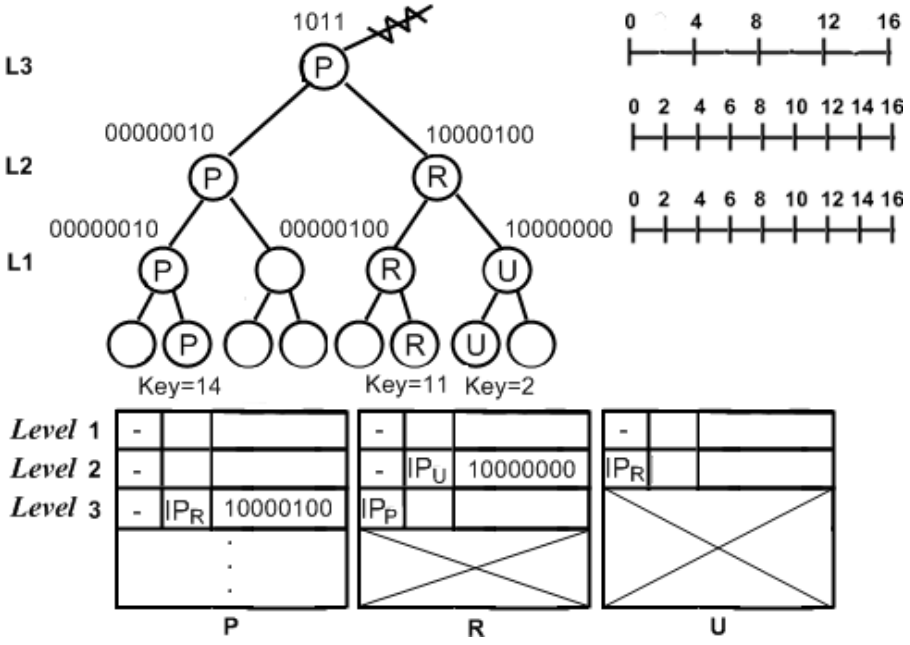

Figura 3.6: Esempio BitVector in XCone

La figura 3.6 mostra un esempio di applicazione della funzione di digest durante la fase di mapping dell'albero XCone. Nell'esempio specifico si è assunto di avere un dominio definito nel range [0,15] con ampiezza degli intervalli uguali e crescenti al crescere dell'altezza dell'albero.

Prima di passare ad una funzione di digest più sofisticata possiamo notare come la suddivisione del dominio attraverso k punti di separazione debba essere effettuata a priori. In particolare si ha la possibilità di selezionare un numero diverso di punti per ogni livello a distanza variabile. Questo procedi-

mento ottimizza la modellazione della distribuzione dei dati e, come vedremo in seguito, agevola anche le operazioni di aggiornamento dell'albero XCone.

L'altra funzione di digest esaminata trae fondamento dal campo delle reti di sensori. Presentata da [24] la funzione di digest prende il nome di *"q-digest"*. Come è noto nelle reti di sensori si hanno dei vincoli strutturali come la durata energetica dei sensori, la limitata quantità di memoria o la connessione di rete non sempre affidabile. Per rispondere a queste problematiche la ricerca si è spinta verso la definizione di soluzioni in grado di sintetizzare nel migliore dei modi i rilevamenti effettuati e di conseguenza ridurre i tempi di trasmissione delle informazioni. La funzione *q-digest* si pone esattamente in questa linea sfruttando le *proprietà matematiche dei quantili*, da cui deriva il nome, per sintetizzare una distribuzione di valori entro un margine di errore fissato.

Definiamo un insieme di valori n con dominio definito nel range $[1, \sigma]$. A tale dominio possiamo associare un albero binario T di altezza $\log \sigma$. L'albero T partiziona per ogni livello l'intero dominio attraverso dei range [min, max] assegnati ad ogni nodo. In particolare tutti i nodi foglia possiedono un ampiezza del range pari ad uno mentre i nodi interni corrispondo all'unione dei range presenti ai nodi figlio di livello inferiore. Ad ogni nodo è associato un contatore di frequenze *count* che indica il numero di valori presenti nel range.

La figura 3.7 mostra un esempio in cui il dominio ha ampiezza $\sigma = 8$ ed il numero totale di valori da sintetizzare è pari a $n = 15$. Il valore n è definito come $n = \sum count(v)$, $v \in LeafSet$.

Una volta definito l'albero T la funzione *q-digest* consiste nel selezionare un'insieme di nodi dell'albero in grado di catturare al meglio la distribuzione dei valori. Come è facile osservare la situazione ideale è rappresentata dalla memorizzazione dei nodi foglia ma, avendo un vincolo di memoria da soddisfare e ipotizzando che l'insieme delle chiavi risulti numeroso occorre effettuare una

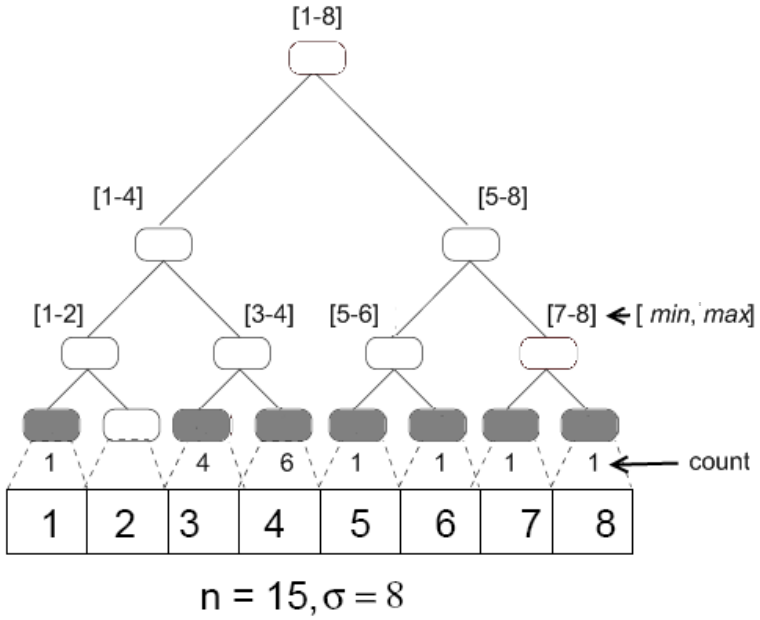

Figura 3.7: Albero logico q-digest

operazione di *compressione* dell'albero.

Fissato un parametro di compressione k, l'operazione di compressione consiste nel raggrupare le chiavi verso i nodi di livello maggiore, in particolare devono essere soddisfatte le seguenti proprietà (*q-digest properties*):

1. $count(v) \leq \lfloor \frac{n}{k} \rfloor$
2. $count(v) + count(v_p) + count(v_s) \succ \lfloor \frac{n}{k} \rfloor$

Dove il parametro v_p rappresenta il nodo padre e il parametro v_s il nodo fratello del nodo v. Se il numero di nodi che possiedono chiavi risulta inferiore o uguale al parametro di compressione k allora l'albero T verrà lasciato intatto senza l'applicazione delle operazioni di compressione. Altrimenti sarà necessario applicare la compressione dell'albero.

L'algoritmo di compressione verifica che le *q-digest properties* risultino rispettate. Nel caso in cui anche solo una risulti violata l'operazione di compres-

sione procede con azzerare i contatori dei nodi figlio ed assegnare al contatore del nodo padre la somma dei precedenti. Vediamo tale operazione con l'esempio in figura 3.8.

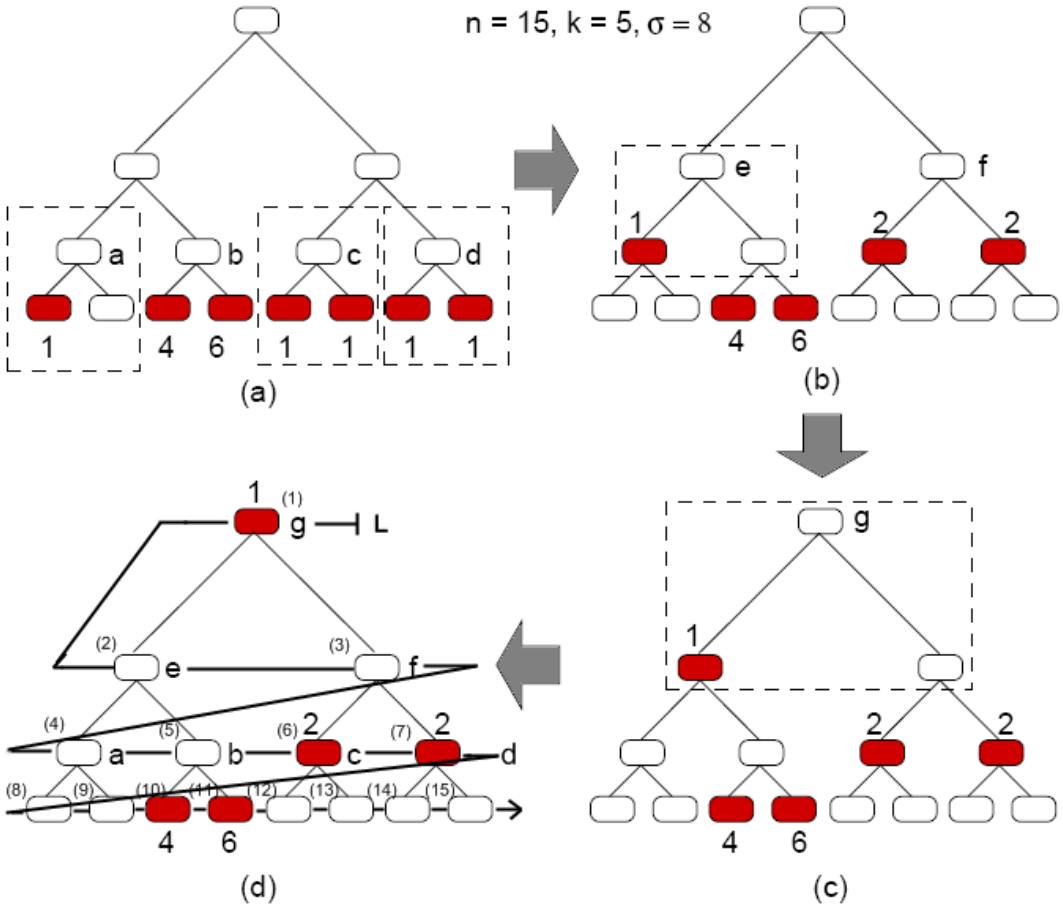

Figura 3.8: Compressione q-digest

Consideriamo un insieme di valori $n = 15$ appartenenti al range [0,15] e i loro rispettivi contatori di occorrenze (Figura 3.8 *(a)*). Fissando il parametro di compressione a k=5 si ottiene la soglia $\lfloor \frac{15}{3} \rfloor = 3$ utilizzata dalle *q-digest properties*. La figura 3.8 *(a)* mostra come i nodi **a**, **c** e **d** non soddisfino la seconda proprietà di digest. Procedendo con la compressione vengono raggruppate le chiavi nei rispettivi nodi padre portando alla situazione illustrata in figura 3.8

(b). Esaminando nuovamente l'albero risulta che il nodo **e** viola nuovamente la proprietà di digest. Applicando l'operazione di compressione si ha la situazione visibile in figura 3.8 *(c)*. Anche in questo caso occorre nuovamente applicare la compressione portando l'albero nello stato finale visibile in figura 3.8 *(d)*.

Una volta eseguita la compressione dei nodi dell'albero T viene definita l'informazione di digest *"q-digest"* attraverso la rappresentazione compatta dell'albero. Consideriamo i nodi dell'albero T come in una lista **L** (Figura 3.8 *(d)*) concatenata e numerata da destra a sinistra, dall'alto in basso. Il *q-digest* è quindi composto da un insieme di tuple della forma $\langle nodeid(v), count(v)\rangle$ tale per cui il contatore di occorrenze di ogni nodo esaminato sia maggiore di zero. Nell'esempio in figura 3.8 *(d)* il *q-digest* è quindi rappresentato nel seguente modo:

$$q - digest(L) = \{\langle 1,1\rangle, \langle 6,2\rangle, \langle 7,2\rangle, \langle 10,4\rangle, \langle 11,6\rangle\}$$

Si può vedere come il parametro di compressione k influenza l'aggregazione dei nodi, in particolare attraverso le proprietà di q-digest si mantiene l'operazione di compressione adattiva. In altri termini si tende a mantenere i nodi con range di ampiezza minore, quindi più accurati, nelle zone dell'albero in cui si ha maggiore concentrazione dei dati e ad accorpare nodi con range più ampi i nodi che posseggono dati più sparsi. Inoltre incrementando i valori da sintetizzare risulta inevitabile un sempre maggiore accorpamento dei nodi in range di ampiezza maggiore.

A questo punto la funzione di digest *d*, utilizzata in XCone, è definita come:

$$d(U) = q - digest(U) = (q - digest(\{U.key\}) \bigcup E_1[ST] \ldots \bigcup E_l[ST])$$

Viene calcolato l'informazione di q-digest che codifica la chiave locale del nodo fisico U, successivamente viene effettuata l'unione tra i q-digest contenuti

nelle entrate significative della Routing Table del nodo U. L'operazione di unione tra più *q-digest* consiste nell'ottenere un unico albero con i contatori di occorrenze di ogni nodo definiti come la somma dei relativi contatori dei singoli alberi q-digest e, se necessario, con l'applicazione dell'operazione di compressione al nuovo albero ottenendo la nuova informazione di q-digest.

L'esempio in figura 3.9 mostra un esempio di applicazione in XCone della funzione q-digest.

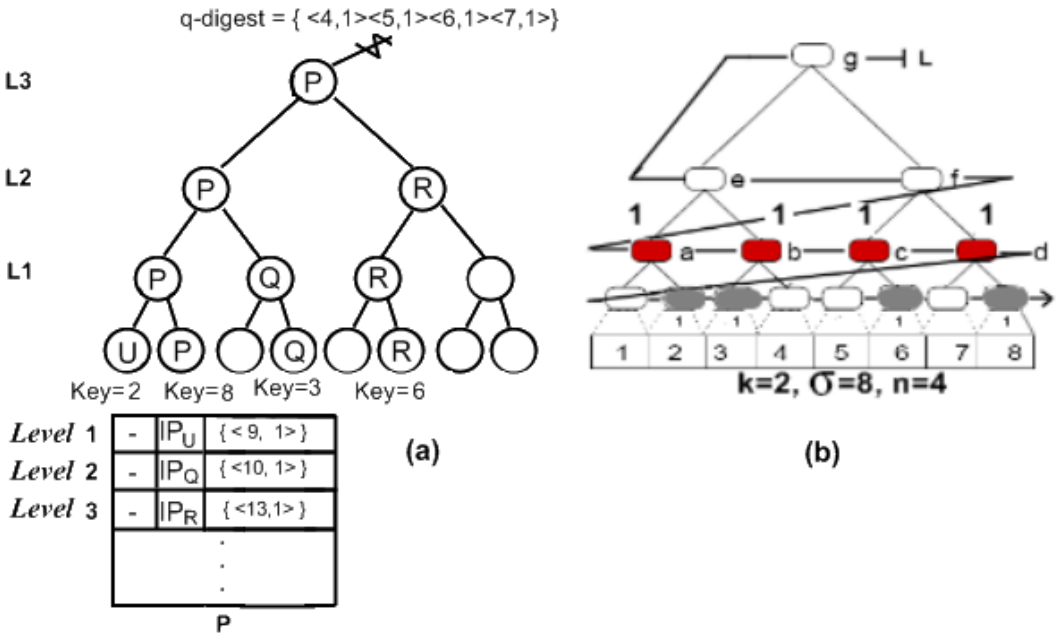

Figura 3.9: Esempio *q-digest* in XCone

Il nodo P calcola inizialmente l'informazione di q-digest a partire dalla propria chiave. Successivamente attraverso l'operazione di unione descritta precedentemente, costruisce ed effettua la compressione dell'albero di q-digest visibile in figura 3.9*(b)*. Da notare come l'unione sia effettuata tra le informazioni di q-digest contenute nella Routing Table di P. Infine P sintetizza il proprio albero radicato nel nodo logico di livello L3 attraverso l'informazione di q-digest visibile in figura.

Una volta ottenuta l'informazione q-digest le operazioni di ricerca effettuabili sono denominate *"quantile query"*. Definito un quantile q, una *quantile query* consiste nel trovare il valore x tale per cui la somma delle frequenze dei valori minori o uguali a x è uguale a q. Nel caso dell'informazione di *q-digest*, supponiamo di esaminare la lista L in maniera inversa dalla coda fino ad un nodo v, se sommiamo in c il numero di occorrenze di tutti i nodi visitati fino al nodo v il valore risultante rappresenta il numero di elementi inferiori o uguali al valore di massimo gestito dal range associato al nodo v.

Illustriamo meglio il concetto di *quantile query* attraverso l'esempio in figura 3.9. Consideriamo il quantile q=0,5, denominato anche quantile mediano, ed effettuiamo la query nell'informazione di q-digest rappresentata dalla lista L con n=4. $\{\langle 7,1\rangle, \langle 6,1\rangle, \langle 5,1\rangle, \langle 4,1\rangle\}$

La quantile query è definita come *"Dato un quantile $q \in (0,1)$, trovare un numero di elementi in L tale per cui la somma delle frequenze sia uguale a **qn**"*.

Se sommiamo le frequenze dei singoli contatori della lista L, già al secondo nodo $\langle 6,1\rangle$ risulta essere maggiore di **qn** = 0,5*4. Pertanto la risposta alla query consiste nell'aver trovato un numero di elementi pari al numero di occorenze, cioè 2, tali per cui siano valori inferiori o uguali al valore 6. Il valore 6 rappresenta il massimo valore gestito dal range del nodo [5-6].

Un'ulteriore classe di query effettuabile sul q-digest è rappresentata dalle *"inverse quantile query"*. In tale query si vuole determinare il numero di elementi inferiori ad un valore x. L'esecuzione avviene in maniera analoga al procedimento precedente. Si scorre la lista L dalla coda alla testa sommando il numero di occorrenze tali per cui il range gestito dal nodo abbia il valore massimo inferiore a x. Il risultato rappresenta il numero di elementi minimo inferiori al valore x. A questo punto le range query possono essere effettuate

come composizione di due *"inverse quantile query"*. Definito il range [low, high] per determinare il numero di elementi appartenenti a tale range si effettuano due *"inverse quantile query"* con x pari a *low* ed *high* e calcolata la differenza tra i risultati ottenuti.

3.5 Le Operazioni

Vediamo nel dettaglio le operazioni d'ingresso, uscita, ricerca ed aggiornamento effettuate da un nodo fisico in XCone.

3.5.1 Join

Per descrivere l'operazione consideriamo l'esempio in Figura 3.10. Il peer P_0 per unirsi alla rete esegue la classica operazione di *join* nella rete DHT. In questa fase si ha l'assegnamento dell'ID e quindi dell'assegnamento del relativo nodo logico foglia di XCone.

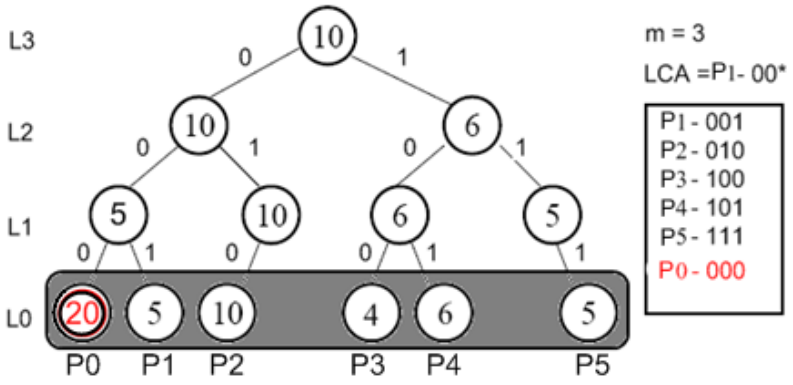

Figura 3.10: Join di un peer in XCone

Contestualmente all'ingresso in rete la DHT fornisce al peer P_0 un riferimento ad un nodo già presente in rete, nell'esempio P_1, in modo tale da condividere con P_0 il prefisso più lungo dell'ID. Formalmente P_1 viene definito il *"Least Common Ancestor"* (LCA) di P_0.

Prima di continuare con l'esempio descriviamo più approfonditamente il concetto di LCA e la sua localizzazione. In generale quanto un peer P si unisce ad XCone ottiene mediante la DHT un identificatore ID. Supponiamo che al momento dell'entrata di P nel sistema, sulla DHT e quindi in XCone, siano già presenti N nodi con identificatori ID_1, ..., ID_i, ID_N. E' necessario che P individui tra questi nodi un nodo M il cui identificatore ID_m abbia la più lunga porzione di prefisso a comune con ID. Da notare come sia possibile che esistano diversi nodi con questa caratteristica. Come mostrato in [22], il nodo LCA può essere individuato mediante la DHT. Infatti è possibile dimostrare che il predecessore o il successore di P sulla DHT è uno dei nodi il cui identificatore condivide con ID il più lungo prefisso. L'operazione di lookup, implementata in ognuna delle DHT attualmente esistenti, consente di reperire il successore ed il predecessore di un nodo e può quindi essere utilizzata per supportare la ricerca del LCA in XCone. Il peer P ottiene tramite la DHT il riferimento al suo predecessore ed al suo successore, quindi sceglie tra questi quello con cui condivide il più lungo prefisso dell'identificativo.

Si consideri, ad esempio, la figura 3.11. Si può notare che il nodo N1 è quello che condivide il più lungo prefisso con N2, mentre il nodo N3, pur essendo adiacente ad N2 risiede dalla parte opposta dell'albero XCone e pertanto non condivide alcun prefisso con N1. Una volta individuato il peer LCA viene calcolato il nodo logico di intersezione tra i due cammini, nell'esempio rappresenta il nodo O. P invia un messaggio di join ad N1 specificanto anche il livello L3 del nodo di intersezione O. Questo ulteriore parametro viene inserito in quanto come nell'esempio riportato in figura il peer R a cui è associato il nodo logico N1, a seguito di un nuovo mapping dei nodi, potrebbe non avere più in gestione il nodo logico O.

Riassumento la fase di join individua attraverso la DHT il nodo LCA con

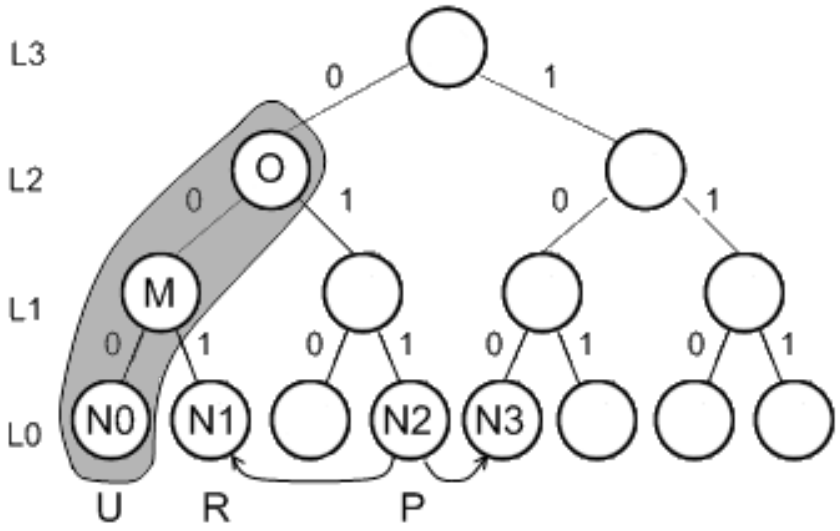

Figura 3.11: Localizzazione del LCA e del nodo logico di intersezione

il procedimento descritto precedentemente, a seguito attraverso il calcolo del nodo logico di intersezione il nodo fisico LCA provvede a verificare se risulta ancora essere il gestore del nodo logico di intersezione, nel caso in cui non lo fosse provvede ad inoltrare la richiesta di join al proprio nodo padre. Questo procedimento esegue una risalita dell'albero lungo il cammino che porta al peer avente in gestione il nodo logico di intersezione. A questo punto l'operazione di join può continuare regolarmente con le ulteriori operazioni.

Ritornando all'esempio iniziale in figura 3.10 sono stati indicati nel basso i peer assegnati ai rispettivi nodi logici foglia mentre all'interno di ogni nodo logico è riportata la chiave gestita dal peer a cui il nodo logico è stato associato.

Assegnato il nuovo ID ed identificato il LCA inizia la fase definita di risalita bit a bit o di *"trickling"*. P_0 confronta il proprio ID con quello di P_1 determinando in questo modo il livello di partenza, l=1 (Figura 3.12 *(a)*).

P_0 possiede un valore maggiore rispetto a quello gestito da P_1 e pertanto procede con l'inserire nella propria Routing Table l'entrata $(-, IP_{P1}, d(P1))$ Figura 3.12 *(b)*. Contestualmente viene informato anche il nodo P_1 sul nuovo

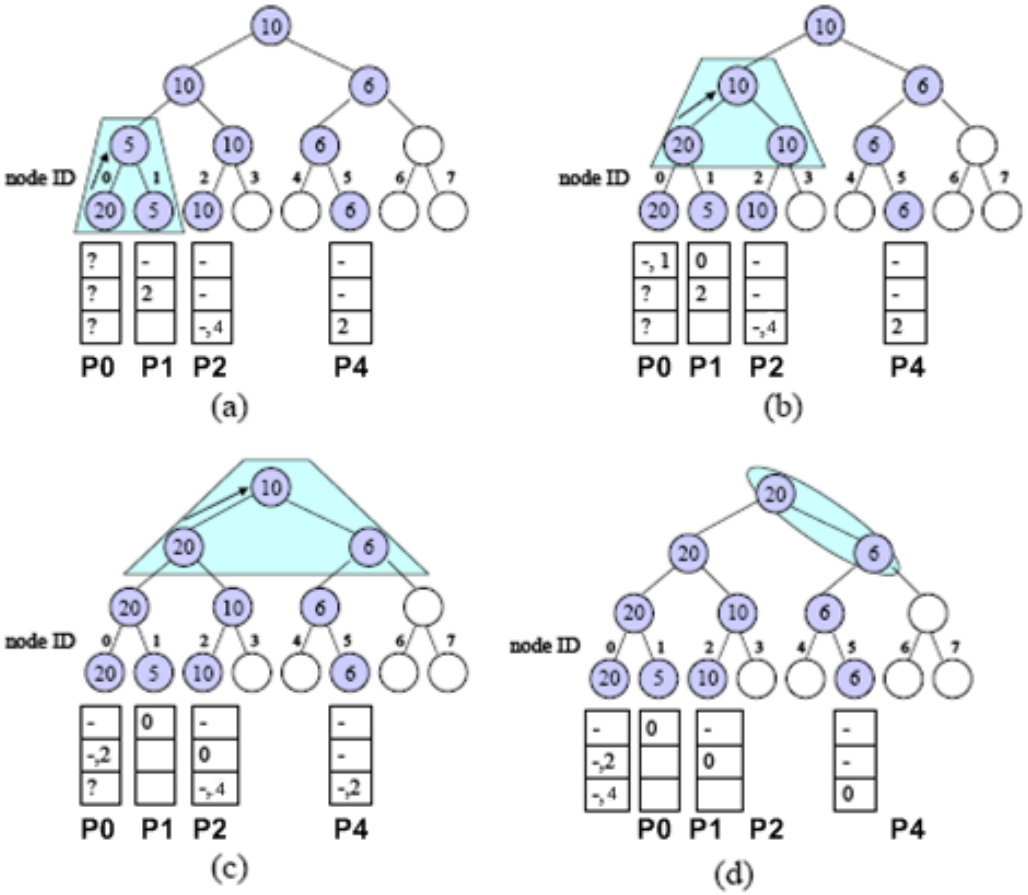

Figura 3.12: Fase di "trickling"

padre. P_1 di conseguenza aggiornerà la sua Routing Table inserendo al livello 1 l'entrata (IP_{P0}) e rimuovendo le informazioni ai livelli successivi (Figura 3.12 *(b)-(c)*).

Continuando la risalita al livello 2, P_0 determina il nodo con cui confrontarsi attraverso le informazioni recuperate dalla tabella di routing del nodo P_1. Viene effettuato un confronto con P_2 e anche in questo caso il nodo P_0 risulta essere il padre del livello 2. P_0 procede esattamente nello stesso modo aggiornando la Routing Table al livello 2 con il valore $(-, IP_{P2}, d(P2))$ ed informando P_2 sul cambio di padre (Figura 3.12 *(c)*).

La fase di *trickling* termina al terzo livello rilevando nuovamente un aggiornamento dell'albero e quindi una variazione nelle Routing Tables (Figura 3.12 *(d)*).

Analizziamo adesso la complessità dell'operazione di Join. La fase di inserimento nella DHT e di ricerca del LCA viene effettuato dal livello sottostante con costo al caso pessimo di $O(\log T)$, con T numero massimo di peer. La fase di *"trickling"*, come visto nell'esempio, può impiegare al caso pessimo $O(\log T)$ operazioni pari al numero di peer presenti nel percorso dalla foglia alla radice dell'albero. Pertanto l'operazione di Join risulta avere un costo complessivo logaritmico con il numero dei peer.

3.5.2 Leave

Quando un peer abbandona inaspettatamente la rete si possono creare al caso pessimo h sotto-alberi disconnessi nell'albero di mapping XCone, con h si definisce l'altezza dell'albero. Prendendo in esame questa situazione passiamo ad analizzare le operazioni di Leave in caso di fallimento.

Durante le normali operazioni di mantenimento un nodo fisico può determinare il fallimento del proprio nodo padre. In questo caso provvede ad

assumere il controllo del nodo logico al livello del nodo padre e ad iniziare la fase di *"reconnect"*. In figura 3.13 *(b)* viene considerato il caso in cui il nodo P_0 fallisce.

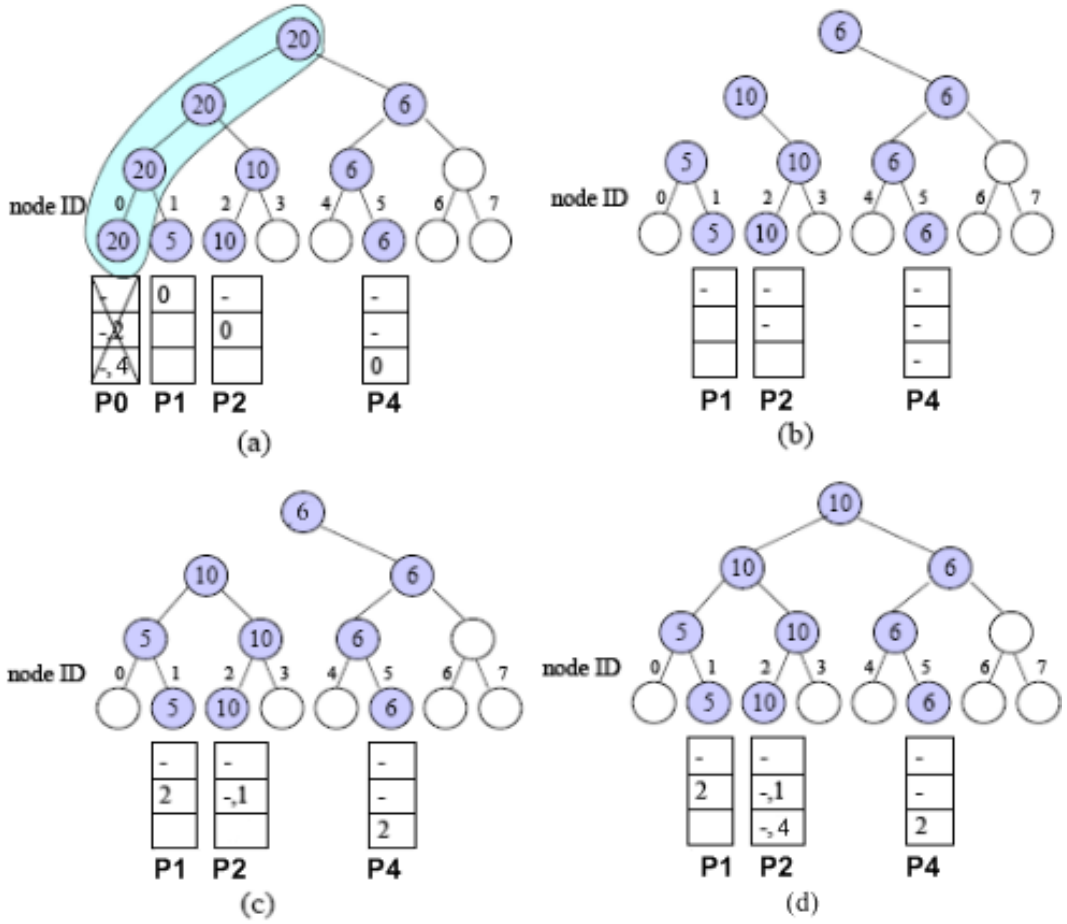

Figura 3.13: Leave di un nodo XCone

L'operazione di riconnessione sfrutta la struttura a prefissi con il quale XCone è stato costruito. Consideriamo il caso illustrato in figura 3.13. Il nodo P_1, il cui sotto-albero è identificato dal prefisso 00*, dopo aver rilevato il fallimento del proprio nodo padre inizierà a cercare un nuovo padre nel sotto-albero 0** di livello 2, attraverso una classica ricerca nella rete DHT. In pratica tale ricerca è analoga a quella effettuata nella fase di join e sfrutta

la conoscenza dei nodi predecessori e successori presenti sulla DHT dopo il fallimento del nodo P_1. Mediante questa ricerca il nodo P_1 localizza il nodo P_2.

P_1 procede alla riconnessione con il nodo padre sfruttando le informazioni contenute nella Routing Table del nodo P_2 Figura 3.13 *(c)*. In parallelo anche gli altri nodi effettueranno le stesse operazioni di riconnessione. Nel caso del nodo P_2 una volta connesso al nodo P_4 avendo un valore maggiore aggiornerà la Routing Table e segnalerà il cambiamento del nodo padre portando l'albero XCone nello stato consistente visibile in figura 3.13 *(d)*.

La complessità di un'operazione di leave può essere calcolata come somma delle operazioni di ricerca e di *"reconnect"*. L'operazione di ricerca consiste nel recuperare un nodo attraverso le operazioni della DHT. Per le proprietà della DHT in generale la ricerca di un nodo ha costo logaritmico. Una volta rilevato il nodo l'operazione di riconnessione risulta esattamente identica a quando fatto per la Join. Si ha quindi che il costo dell'operazione di leave al caso pessimo è pari a $O\left((\log T)^2\right)$. In realtà la complessità reale risulta inferiore potendo eseguire le operazioni in parallelo.

Nel caso in cui l'operazione di leave sia dovuta ad un'uscita volontaria il costo risulta semplicemente logaritmico in quanto viene evitata la procedura di ricerca del nodo.

3.5.3 Find

L'operazione di find può essere eseguita a partire da un qualunque peer P della rete. Consideriamo il caso in cui un peer P voglia effettuare una query, Q, in cui richiede K nodi con valore X compreso nell'intervallo $V \leq \mathrm{X} \leq S$. La ricerca consiste in una esplorazione dell'albero XCone attraverso le informazioni possedute nella Routing Table dei vari peer.

La procedura di ricerca può essere suddivisa in due operazioni. La prima operazione consiste nella risalita dell'albero XCone denominata anche operazione di *trickling*. La seconda operazione consiste *nell'esplorazione dei sotto-alberi*. In XCone la prima operazione viene effettuata in maniera sequenziale, in particolare l'esplorazione di ogni sotto-albero rappresentato da una entrata della routing table viene effettuata verificando mano a mano che il numero di risorse localizzate sia sufficiente e in tal caso interrotto il procedimento di risalita. Nel processo di esplorazione dei sotto-alberi la ricerca avviene in parallelo in maniera guidata dalle informazioni di digest. In altre parole un nodo dopo aver verificato attraverso l'informazione di digest la presenza di valori in grado di risolvere la query, propagata immediatamente la richiesta di esplorazione in basso verso i nodi foglia.

L'algoritmo 3.1 illustra lo pseudocodice eseguito da ogni peer in seguito alla richiesta di trickling di una query Q innescata da un messaggio del tipo FIND_MSG. Supponiamo che il generico peer P riceva una query Q in cui si richiedano K risorse e che la query sia stata originata dal peer denominato QueryNode.

Il generico peer P attraverso la funzione *matchValue* controlla se la propria chiave soddisfa la query Q. In caso affermativo P invia il proprio indirizzo come risultato al QueryNode. Il QueryNode tenendo traccia dei risultati raccolti, provvede a trasmettere al peer P la quantità di peer attualmente rilevati dalla query Q. Il peer P pertanto analizza la variabile targa collectedMatches e se risultano già raggiunti il numero di risorse K richieste termina l'operazione attraverso un uscita forzata.

Supponendo che le risorse localizzare non siano sufficienti il peer P procede a collezionare un insieme S di propri sotto-alberi attraverso le informazioni possedute nella propria Routing Table (RT). La funzione di digest consente di

selezionare esclusivamente i sotto-alberi in cui è stata rilevata la presenza di risorse compatibili con la query Q (digestStrategy.*estimate* che farà riferimento al campo ST contenente le informazioni di digest relative alle chiavi presenti nel sottoalbero).

Algoritmo 3.1: Algoritmo di ricerca: fase di trickling

```
Find(Q, K, QueryNode) {
    /* Check if local Key match with query Q */
    if Q.matchValue(keyManager.getCurrentValue()) {
        send(QueryNode, FIND_RESULT_MSG, IPNodeAddress)
        receive(QueryNode, collectedMatches)
        if(collectedMatches >= K)
            exit;
    }
    /* Check subtree that match with query Q */
    S = { }
    for i=0 to RT.lenght do {
        subTree = digestStrategy.estimate(Q, RT[i].ST);
        if(subTree > 0)
            S = S + RT[i].IP2;
    }
    /* Send for each subtree a find child */
    foreach child in S {
        send(child, FIND_CHILD_MSG, Q, K, QueryNode);
        send(QueryNode, GET_RESULTS_MSG);
        receive(QueryNode, collectedMatches);
        if(collectedMatches >= K)
            exit;
    }
    /* Check if continue trickling */
    if(S == 0) {
        send(XConeParentNode, FIND_MSG, Q, K, QueryNode);
    } else {
        send(QueryNode, FIND_TRICKLING_MSG)
        receive(QueryNode, collectedMatches);
        if(collectedMatches >= K)
            exit;
        else
            send(XConeParentNode, FIND_MSG, Q, K, QueryNode);
    }
}
```

Per tali sotto-alberi, in maniera sequenziale, P invia una richiesta di esplorazione attraverso un messaggio di tipo FIND_CHILD_MSG. Da notare come la richiesta sia inviata al peer che gestisce la radice del sotto-albero.

Una volta inviato il messaggio, prima di passare alla successiva esplorazione, il peer P attende di ricevere una risposta dal QueryNode sul numero di risorse localizzate nella precedente richiesta. In questo modo se il numero di risorse dovesse risultare sufficiente viene interrotta la fase di esplorazione dei successivi sotto-alberi e di conseguenza terminata la fase di trickling attraverso un uscita forzata. Questa operazione di sospensione dell'eplorazione ha come fine quello di ridurre al numero strettamente necessario il numero di nodi esplorati nell'albero XCone.

Terminata la fase di esplorazione il peer P controlla se non sono stati esplorati sotto-alberi, in questo caso effettua direttamente l'operazione di trickling inviando un messaggio di FIND_MSG al proprio nodo padre. Altrimenti nel caso in cui siano stati esplorati sotto-alberi, il nodo P sospende la ricerca inviando un messaggio di FIND_TRICKLING_MSG al QueryNode ed attende nuovamente il numero di risorse localizzate. Anche in questo caso la sospensione è effettuata al fine di controllare la propagazione verso il nodo radice delle operazioni di Find.

Per quanto riguarda le operazioni di esplorazione dei sotto-alberi, lo pseudocodice 3.2 illustra l'algoritmo eseguito da ogni radice R del sotto-albero a seguito di una richiesta di esplorazione verso il basso. La procedura viene attivata con la ricezione di un messaggio del tipo FIND_CHILD_MSG.

In questo caso in nodo radice R controlla preliminarmente se la propria chiave soddisfa la query Q. In tal caso procede ad inviare il proprio indirizzo al QueryNode e successivamente a ricevere il numero di risorse localizzate dal QueryNode.

Algoritmo 3.2: Algoritmo di ricerca: fase di esplorazione

```
FindChild(Q, K, QueryNode) {
    /* Check if local Key match with query Q */
    if Q.matchValue(keyManager.getCurrentValue()) {
        send(QueryNode, FIND_RESULT_MSG, IPNodeAddress)
        receive(QueryNode, collectedMatches)
        if(collectedMatches >= K)
            exit;
    }

    /* Check subtree that match with query Q */
    S = { }
    for i=0 to RT.lenght do {
        subTree = digestStrategy.estimate(Q, RT[i].ST);
        if(subTree > 0)
            S = S + RT[i].IP2;
    }

    /* Send for each subtree a find child */
    foreach child in S
        send(child, FIND_CHILD_MSG, Q, K, QueryNode);
}
```

Nel caso in cui le risorse non siano sufficienti, R procede ad inoltrare in parallelo le richieste di esplorazione dei propri sotto-alberi. La procedura di esplorazione procede verso i nodi logici foglia dell'albero XCone e come accennato precedentemente viene diretta ai soli nodi foglia che attraverso le informazioni di digest risultano compatibili con la query Q. In tal senso l'algoritmo procede nella maniera più veloce a recuperare le relative risorse.

Concludendo il costo di una operazione di ricerca in XCone è determinata dal costo della fase di trickling. Tale fase come osservato può coinvolgere al caso pessimo un intero cammino dalla foglia al nodo radice dell'albero, pertanto coinvolgere al massimo $O(\log T)$ nodi fisici, con T numero di peer nella rete. Analizzando invece i messaggi ricevuti dal QueryNode, possiamo determinare che avendo una esplorazione sequenziale dei sotto-alberi il QueryNode potrà ricevere al più K messaggi, corrispondenti ai risultati ricevuti, più $\log T$ messaggi

di richieste di trickling per un totale di $O(\log T + K)$ messaggi ricevuti.

3.5.4 Change

In XCone l'operazione di aggiornamento di un valore risulta poco costosa. Supponiamo che l'aggiornamento venga effettuato a partire dal nodo foglia N. L'aggiornamento comporta al caso pessimo l'aggiornamento di tutte le informazioni di digest dei nodi presenti sul cammino dalla foglia N alla radice dell'albero. Inoltre l'aggiornamento può comportare anche una modifica del mapping (nodi logici/nodi fisici), se la strategia di mapping utilizzata risulta essere legata alle chiavi gestite dai nodi. Ad esempio nel caso di XCone in cui la funzione di mapping sia legata al nodo con valore di chiave massimo, l'operazione di change comporterà sia l'aggiornamento delle informazioni di digest che quelle di mapping tra i nodi. Consideriamo questo ultimo caso, che risulta essere il più complesso, nell'ipotesi che la funzione di mapping sia la funzione di valore di chiave massimo.

Con riferimento alla figura 3.14, supponiamo che il peer R effettui un aggiornamento di chiave. Come prima operazione occorre verificare la validità della relazione di mapping tra il nodo R e il rispettivo nodo padre. Nel caso in cui tale relazione non risulti più valida allora il peer R provvede a diventare il nuovo nodo padre aggiornando le rispettive Routing Table e calcolando la nuova informazione di digest. L'operazione prosegue mediante un processo di "trickling" come quello illustrato nel paragrafo precedente. Il costo dell'operazione in questo caso è al massimo $\log T$ con T numero di nodi XCone.

Si noti come in questo caso la relazione di mapping tra R ed i nodi suoi figli risulta automaticamente verificata. Questa è assicurata dal fatto che la chiave di R è maggiore di quella del padre P e tutti i figli di R hanno chiave maggiore di P.

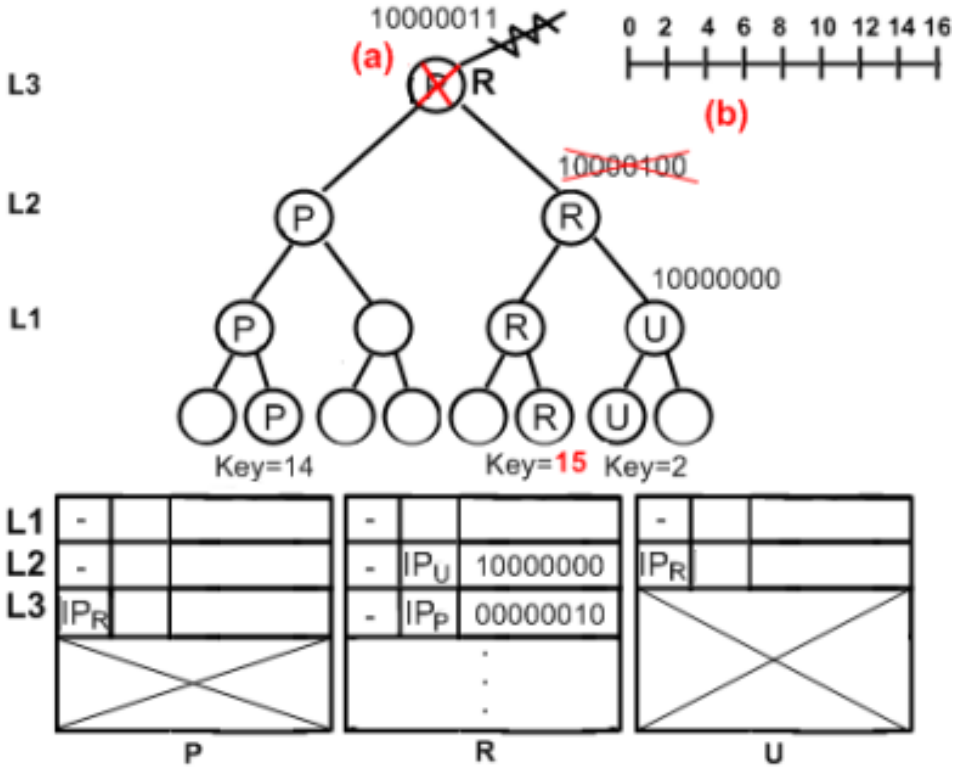

Figura 3.14: Esempio Change Nodo padre

Nel caso invece in cui il nodo padre P confermi la relazione di mapping, R procede con il verificare la relazione di mapping con tutti i nodi figlio a partire dal livello minore. Il primo figlio a partire dai livelli bassi della routing table che non conferma la relazione di mapping innesca l'aggiornamento dell'albero attraverso un procedimento analogo a quello di una join tra il nodo figlio e il nodo R (Figura 3.15). Il costo dell'operazione è identico al costo della *join* XCone.

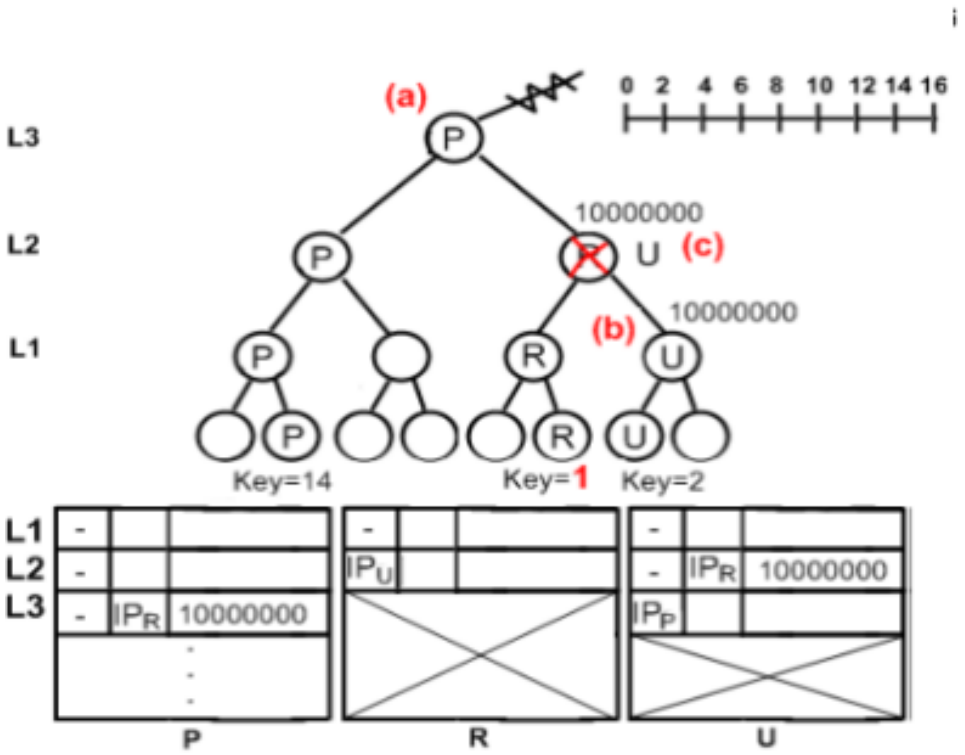

Figura 3.15: Esempio Change Figlio

Infine l'ultimo caso da esaminare è la situazione in cui la relazione di mapping è confermata da tutti i nodi figlio (Figura 3.16). In questo caso la funzione di mapping nodi logici-nodi fisici non cambia, ma come negli altri casi R termina la procedura di *change* calcolando la nuova informazione di digest e, se differente dalla precedente, provvede a notificarla al nodo padre P. Da nota-

re come la propagazione dell'aggiornamento può al caso pessimo coinvolgere un numero di nodi fisici pari al cammino dalla foglia alla radice dell'albero portando ad un numero massimo di $O(\log T)$ messaggi scambiati.

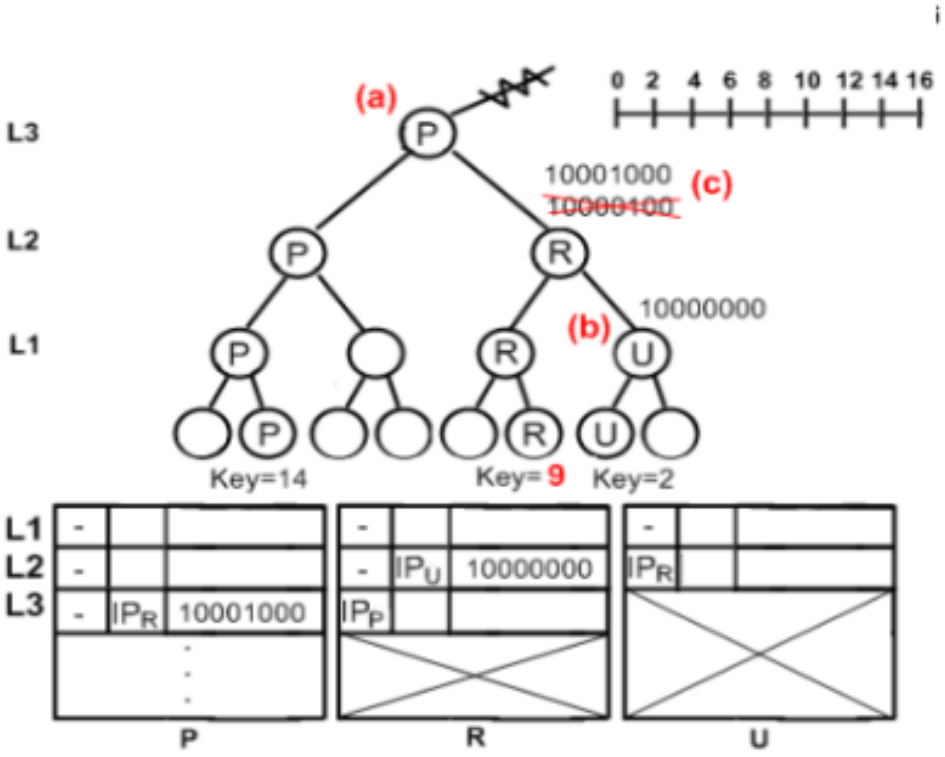

Figura 3.16: Esempio Change informazione digest

Capitolo 4

XCONE: Implementazione

In questo capitolo verrà illustrata l'implentazione del sistema XCone. Saranno mostrate le scelte architetturali, l'organizzazione dei packages e il diagramma delle classi. Infine verranno illustrate le principali interazioni tra le istanze del sistema.

4.1 Framework Overlay Weaver

Lo sviluppo di XCone si basa sull'ut ilizzo ed estensione del framework Overlay Weaver. Overlay Weaver è un framework *"Open Source"* realizzato come lavoro di ricerca da [35] e reperibile gratuitamente in rete. Le caratteristiche principali del framework consistono nella elevata modularità e strutturazione a livelli, questo consente di realizzare applicazioni di rete in grado di appoggiarsi a moduli esistenti e di ereditarne una elevata semplicità di configurazione. Il forte disaccoppiamento del framework consente infatti di effettuare agevolmente delle variazioni del comportamento dell'applicazione. Le reti strutturate già presenti in OW sono Chord, Pastry, Tapestry, Kademlia e Koorde. In particolare vi sono implementate come applicazioni ad alto livello una semplice shell DHT interattiva e una shell Multicast. Il framework mette a disposizione anche una serie di tools per lo sviluppo e il debugging come l'emulatore di

nodi, il generatore di scenari ed un visualizzatore grafico della topologia della rete.

4.1.1 Architettura del framework

L'architettura di Overlay Weaver, come già accennato, risulta fortemente strutturata a livelli in particolare possiamo individuare i seguenti livelli:

- Applicazioni
- Servizi ad alto livello
- Servizi di routing
- Servizi di memorizzazione

La figura 4.1 riassume graficamente l'architettura e consente di poter osservare le relazioni di dipendenza tra i vari livelli. Il livello delle applicazioni utilizza esclusivamente i servizi disponibili ad alto livello, analogamente i servizi ad alto livello si poggiano sui servizi di routing e di memorizzazione. Per quanto riguarda il livello di routing Overlay Weaver effettua un'ulteriore suddivisione basata sul concetto di *"Key-based routing"*, KBR, introdotto da [36] e che generalizza le funzionalità definite nelle principali DHT. In generale in una rete strutturata ogni nodo memorizza un sotto-insieme di collegamenti ad altri nodi, la scelta di tali nodi identifica la topologia della rete. Per effettuare la ricerca di una chiave il modello KBR, in accordo ad una metrica, consente di indirizzare la ricerca verso dei nodi via via sempre più vicini al nodo target. Overlay Weaver basandosi su tale modello suddivide ulteriormente il livello di routing in:

- **Routing Driver**, espone le principali operazioni basate sul modello KBR.

- **Routing Algorithm**, contiene le diverse strategie di routing.
- **Messaging Service**, contiene diverse strategie di comunicazione di rete.

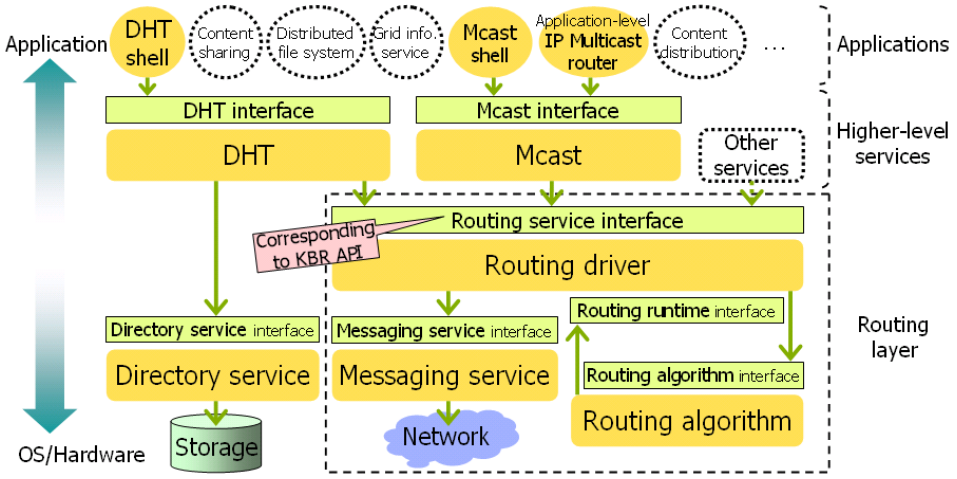

Figura 4.1: Architettura Overlay Weaver

Come implementazione del livello di *Routing Driver* Overlay Weaver fornisce due versioni, una iterativa ed una ricorsiva. La figura 4.2 illustra le diverse comunicazioni effettuate dai diversi tipi di routing.

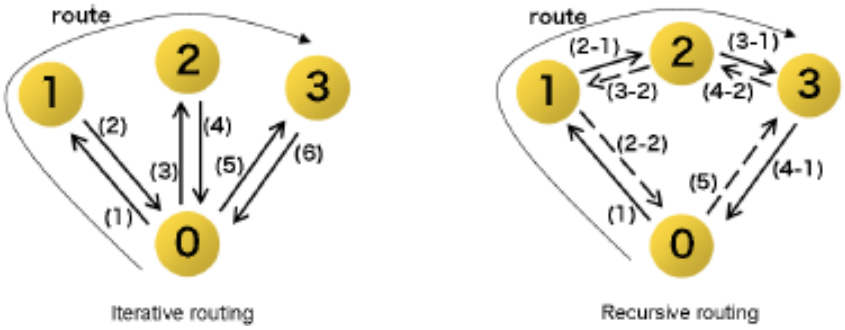

Figura 4.2: Tipologie di routing in Overlay Weaver

Le linee identificano i messaggi scambiati tra i vari nodi mentre i numeri annotati l'ordine di scambio. La notazione tratteggiata sta ad indicare i mes-

saggi che possono non essere necessariamente scambiati durante il routing, ma che servono in caso di utilizzo di protocolli non affidabili (Es. UDP).

Per il livello di *Routing Algorithm* OW implementa le strategie di routing Chord, Pastry, Tapestry, Kademlia e Koorde. Nel livello di *Messaging Service* sono implementati le classi per la comunicazione di rete attraverso i protocolli TCP, UDP e intra-thread utilizzato nel caso di emulazione.

Nel livello di memorizzazione OW offre la memorizzazione attraverso un database relazionale (Berkeley DB) o in alternativa la memorizzazione in memoria principale attraverso le hash table della Java standard class library.

4.1.2 Tools di sviluppo

Overlay Weaver mette a disposizione una serie di strumenti di sviluppo aggiuntivi come l'emulatore di rete, il generatore di scenari e un visualizzatore grafico di rete. Lo strumento di maggior utilizzo risulta essere l'emulatore. Attraverso l'emulatore è possibile testare le applicazioni in maniera immediata in un ambiente controllato. L'emulatore prevede due modalità di esecuzione, la *modalità locale* in cui i nodi emulati sono localizzati nella sola macchina fisica locale e la *modalità distribuita* in cui varie istanze di emulatori sono attive in macchine fisiche distribuite e la comunicazione tra i rispettivi nodi emulati avviene attraverso la rete. Per quanto riguarda la modalità di interazione con lo strumento possiamo distinguere la modalità interattiva e la modalità batch. Nel primo caso i comandi sono impartiti all'emulatore attraverso una console testuale, nel secondo caso viene fornito un *file di testo* o *scenario* contenente una serie di comandi in grado di essere interpretati ed eseguiti dall'emulatore.

La figura 4.3 illustra la struttura tipica di uno scenario. In particolare è possibile notare le relative direttive in grado di eseguire una serie ripetuta di comandi e una sequenza di comandi in grado di temporizzare l'esecuzione.

```
timeoffset 2000
# invoca il primo nodo
class ow.tool.dhtshell.Main
arg -p 10000
schedule 0 invoke
# invoca 3 nodi
arg
schedule 1000,1000,3 invoke
timeoffset 7000
# i 3 nodi entrano  nell'overlay
schedule 0 control 1 init emu0
schedule 1000 control 2 init emu0
schedule 2000 control 3 init emu0
# eseguo put e get
schedule 4000 control 1 put a_key a_value 600 secret
schedule 5000 control 1 get a_key
```

Figura 4.3: Scenario interpretabile dall'emulatore

Infine la creazione di uno scenario può essere assistita da un ulteriore tool presente nel framework denominato generatore di scenari. L'ultimo strumento presente nel framework è il visualizzatore grafico di rete visibile in figura 4.4.

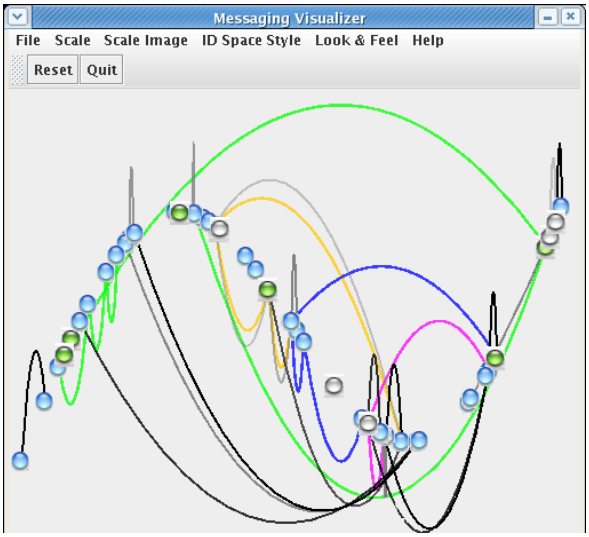

Figura 4.4: Esempio di visualizzazione grafica della rete

Il visualizzatore consente di rappresentare in varie modalità grafiche la dislocazione dei nodi nella rete e di visualizzare in tempo reale le comunicazioni

tra i nodi. In particolare è possibile distinguere graficamente le varie tipologie di messaggi scambiati.

4.2 XCone: Implementazione

In questa sezione si descrive il sistema XCone attraverso le principali classi e le relazioni fra di esse. Si illustreranno le principali scelte progettuali e le problematiche affrontate per l'integrazione con il framework OW. Infine saranno presentate attraverso diagrammi di sequenza i messaggi scambiati e le interazioni effettuate.

4.2.1 Packages e diagramma delle classi

In XCone si sono rispettate le linee guida che il framework OW propone, si è quindi realizzata un'architettura modulare in grado di integrarsi con il resto del framework. Il nuovo servizio ad alto livello *"XCone High-level service"* espone sia le operazioni di base sulle DHT tradizionali che le operazioni di interrogazione attraverso range query. Per testare il nuovo servizio si è implementato anche una semplice console interattiva come applicazione ad alto livello.

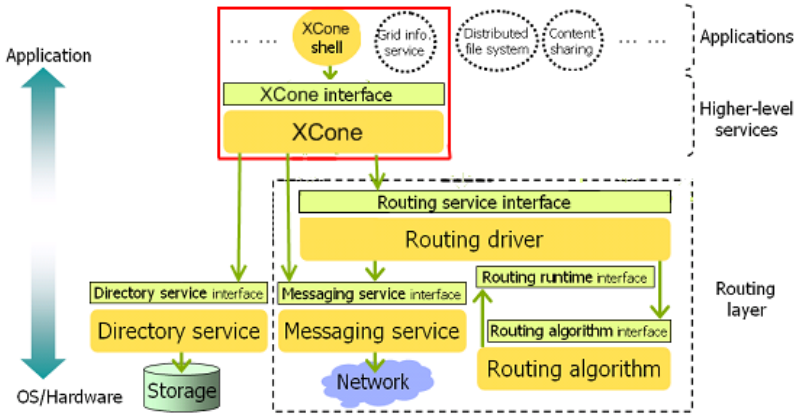

Figura 4.5: Integrazione XCone Framework OverlayWeaver

Package	Livello	Descrizione
ow.tool	Application	Applicazioni e gli strumenti di sviluppo
ow.dht	Service	Servizio di DHT standard
ow.ipmulticast	Service	Servizio di Multicast
ow.directory	Service	Servizio per la memorizzazione dei dati
ow.routing	Routing	Funzionalità di routing driver e algorithm
ow.messaging	Routing	Funzionalità di comunicazione e trasporto
ow.id	Routing	Gestione degli identificativi di rete

Tabella 4.1: Descrizione Packages Overlay Weaver

Oltre i package definiti in tabella 4.1, che rappresentano i packages generali del framework, XCone definisce il nuovo servizio introducendo i seguenti packages:

- ***ow.tool.xconeshell***, applicazione in grado di fornire una console interattiva XCone.
- ***ow.xcone***, servizio XCone.

La struttura interna del package ***ow.tool.xconeshell*** risulta estremamente semplice. Si è definita una classe principale **Main** ed una serie di classi che implementano l'interfaccia **Command**. All'interno di ogni tipologia di classe commando viene effettuata l'invocazione di un particolare servizio XCone.

Per quanto riguarda la struttura del servizio XCone prima di illustrare le classi che compongono il package ***ow.xcone*** occorre illustrare le principali scelte progettuali. In XCone un aspetto fondamentale è legato al *forte disaccoppiamento* tra le operazioni di mapping e ricerca dalle singole strategie algoritmiche. La separazione delle diverse strategie di mapping e di digest consente di introdurre una elevata manutenibilità e configurazione del sistema. Infine per quanto riguarda la gestione delle chiavi si vorrebbe predisporre il sistema alla gestione futura di valori multi-attributo.

Per soddisfare queste proprietà si è ricorso durante la progettazione del

sistema ad uno dei pattern fondamentali di progettazione denominato *"Strategy"*[37]. Il pattern *Strategy* consente di isolare un determinato algoritmo all'interno di un oggetto e di cambiare dinamicamente l'algoritmo o strategia utilizzata senza impattare sul resto dell'architettura.

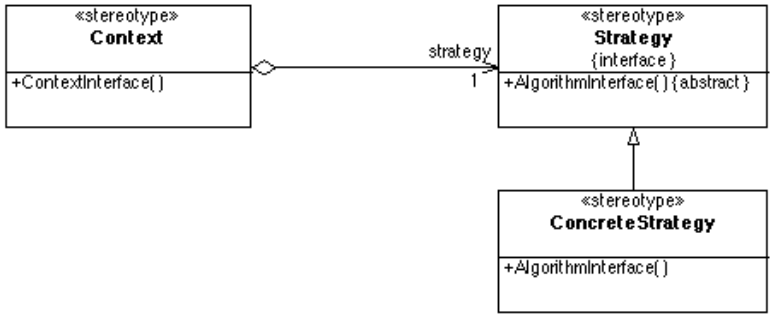

Figura 4.6: Pattern Strategy

Infine l'ultima scelta progettuale è stata quella di rendere il servizio *multithreading*. In particolare come vedremo dalla struttura delle classi i threads impiegati riguarderanno la gestione degli ingressi nella rete XCone, il processo di ricerca delle chiavi e il monitoraggio dei fallimenti dei nodi fisici. Il tutto gestendo le opportune politiche di *"lock"/"unlock"* delle strutture dati.

Vediamo adesso nel dettaglio l'architettura del servizio XCone descritta attraverso il diagramma delle classi (Figura 4.7). Iniziamo la descrizione del sistema partendo dalle interfacce introdotte.

- **XCone**, interfaccia che espone i metodi invocabili sul servizio XCone.
- **XConeMappingStrategy**, interfaccia che espone le operazioni di mapping dei nodi logici ai nodi fisici.
- **XConeDigestStrategy**, interfaccia che espone le operazioni di sintetizzazione di un insieme di chiavi.

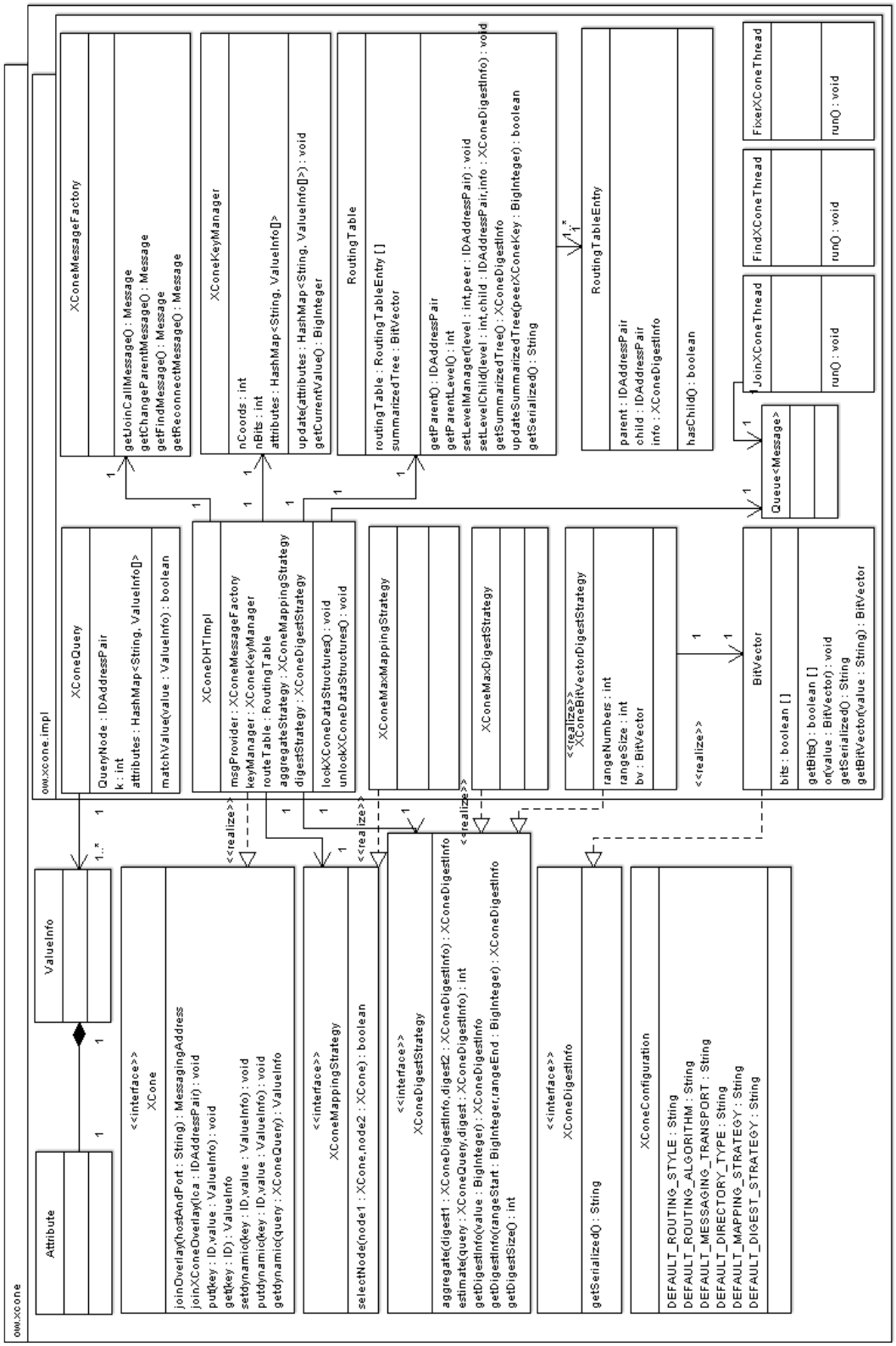

Figura 4.7: Diagramma delle classi servizio XCone

Nel dettaglio delle singole interfacce troviamo per l'interfaccia **XCone** i seguenti metodi:

+ *joinOverlay(String hostAndPort): MessagingAddress*, consente l'ingresso nella rete DHT attraverso un nodo di bootstrap, riferito da *hostAndPort*. Il valore di ritorno rappresenta il nodo di bootstrap codificato come endpoint del framework con uno specifico protocollo di comunicazione.

+ *joinXConeOverlay(lca: IDAddressPair): void*, effettua l'ingresso nella rete XCone costruita al di sopra della DHT. Da notare come l'argomento *lca* è un riferimento al least common ancestor restituito dal livello Routing Driver di Overlay Weaver, dopo la stabilizzazione della DHT, in seguito all'inserimento del nuovo nodo.

+ *put(key: ID, value: ValueInfo): void*, effettua l'inserimento di un valore nella rete DHT per poter effettuare query di tipo *"Exact Query"*.

+ *get(key: ID): ValueInfo*, consente di ricercare un valore attraverso una *"Exact Query"* nella DHT.

+ *setdynamic(key: ID, value: ValueInfo): void*, specifica il valore iniziale dell'attributo dinamico su cui poter eseguire *"Range Query"*.

+ *putdynamic(key: ID, value: ValueInfo): void*, aggiorna l'attributo dinamico.

+ *getdynamic(query: XConeQuery): ValueInfo[]*, viene effettuata una *"Exact Query"* sugli attributi dinamici posseduti dai peer della rete. I parametri della query sono contenuti in un'istanza della classe XConeQuery descritta in seguito.

Da precisare come nel metodo *joinOverlay* il meccanismo per la determinazione del LCA dipende dal tipo di DHT sottostante. Nel caso di XCone, in cui attualmente la DHT supportata è CHORD, la determinazione del LCA consiste nel verificare i riferimenti al nodo predecessore e successore dell'anello CHORD selezionando il nodo con cui si condivide il più lungo prefisso dell'identificativo. Per astrarre dal particolare routing algorithm è stato necessario modificare alcuni metodi del Routing Level. Il codice sviluppato è predisposto per supportare anche altre DHT (Pastry, Kademlia, etc.), ovviamente con le opportune implementazioni del routing level.

L'interfaccia **XConeMappingStrategy** risulta invece essere composta dal seguente metodo:

+ *selectNode(node1: XCone, node2: XCone): boolean*, confronta i due nodi XCone applicando la relativa strategia di mapping. L'operazione ha come risultato un valore booleano il quale indica se il primo dei due nodi è stato selezionato. Nel caso in cui si vogliano effettuare delle strategie di mapping più evolute il metodo agirà direttamente sulle strutture dati dei nodi *node1* e *node2*.

Per quanto riguarda l'interfaccia **XConeDigestStrategy** sono esposti i seguenti metodi:

+ *aggregate(digest1: XConeDigestInfo, digest2:XConedigestInfo): XConeDigestInfo*, consente di creare una nuova informazione di tipo XConeDigestInfo che rappresenta l'unione delle informazioni contenute nei rispettivi digest passati per argomento.

+ *estimate(query: XConeQuery, digest:XConedigestInfo): int*, effettua una stima dei valori, sintetizzati attraverso l'informazione di digest, che soddisfano la query passata per argomento.

Attraverso l'astrazione fornita dalle interfacce **XConeMappingStrategy** e **XConeDigestStrategy** è possibile isolare le diverse strategie algoritmiche di mapping e di digest.

Passiamo adesso alla descrizione delle classi presenti in XCone. Le classi che implementano le funzionalità di mapping e di aggregazione sono le seguenti:

- **XConeDHTImpl**, rappresenta la classe principale e il contesto di esecuzione del servizio XCone. Fornisce i vari handlers per i messaggi ricevuti dal livello di routing.

- **XConeMaxMappingStrategy**, classe che implementa una concreta strategia di mapping basata sulla scelta del nodo fisico che gestisce la chiave di valore massimo.

- **XConeBitVectorDigestStrategy**, classe che implementa una concreta strategia di digest basata sulla struttura BitVector.

- **XConeMaxDigestStragety**, classe che implementa una concreta strategia di digest basata sulla memorizzazione del valore massimo tra un insieme di chiavi.

- **XConeDigestInfo**, esprime l'informazione risultato delle operazioni di digest.

Continuando la descrizione del sistema possiamo identificare le classi che implementano la strutture dati XCone. In particolare troviamo il gestore delle chiavi e la Routing Table.

- **XConeKeyManager**, fornisce il supporto per la gestione della chiave locale.

- **RoutingTable**, contiene le informazioni sulla routing table del nodo fisico XCone.
- **RoutingTableEntry**, rappresenta la singola entrata di una routing table e le operazioni definite su di essa.
- **XConeQuery**, rappresenta la range query. Le variabili di stato possedute dalla classe memorizzano le informazioni sul numero di elementi da recuperare k, sul range richiesto e naturalmente sull'indirizzo del nodo che ha originato la query *QueryNode*.
- **Attribute**, definisce gli attributi ed in particolare la chiave.
- **ValueInfo**, rappresenta i valori che compongono un attributo.

Illustrando per le principali classi i metodi esposti troviamo per la classe **XConeKeyManager** il cui compito risulta quello di gestire l'attributo dinamico il metodo:

+ *update(attributes: HashMap* $\langle String, ValueInfo[]\rangle$*): void*, consente di effettuare l'aggiornamento dell'attributo dinamico.

Per la classe **RoutingTable** troviamo i classici metodi di gestione della tabella con l'inserimento e cancellazione delle relative informazioni per ogni entries:

+ *getParent(): IDAddressPair*, restituisce il nodo fisico XCone padre.
+ *getParentLevel(): int*, restituisce il livello logico nell'albero XCone contenente il primo nodo logico gestito dal peer padre.
+ *setLevelManager(level: int, peer: IDAddressPair): void*, memorizza il nodo fisico passato per argomento come gestore del livello logico.

+ *setLevelChild(level: int, child: IDAddressPair, info: XConeDigestInfo): void*, memorizza il nodo fisico figlio al livello logico passato per argomento con la relativa informazione di digest sul contenuto del sotto-albero radicato nel peer figlio.

+ *getSummarizedTree(): XConeDigestInfo*, restituisce un'informazione di digest che sintetizza l'intero albero gestito dal nodo fisico corrente.

+ *updateSummarizedTree(peerXConeKey: BigInteger, level: int): boolean*, aggiorna l'informazione che sintetizza l'intero albero gestito con la nuova chiave del peer. Restituisce un valore booleano che indica se la nuova informazione di digest risulta diversa da quella precedentemente posseduta.

+ *getSerialized(): String*, restituisce la tabella di routing in maniera serializzata.

Per la classe **XConeQuery** il metodo principale è rappresentato da:

+ *matchValue(value: ValueInfo): boolean*, consente di verificare se un valore è verificato per la relativa query.

Il servizio XCone è stato realizzato in maniera multi-threading pertanto sono stati introdotti i seguenti threads:

- **JoinXConeThread**, con il compito di gestire le richieste di join nell'albero XCone ricevute dai peer entranti. Se le strutture dati interne risultano in stato di *lock* attenderà lo sbocco mediante attesa passiva.

- **FindXConeThread**, consente di gestire la propagazione della ricerca nell'albero XCone. Il thread viene attivato dal nodo fisico che effettua la

range query e gestisce la propagazione del processo di query nell'albero XCone.

- **FixerXConeThread**, controlla ad intervalli regolari lo status di connessione del rispettivo nodo padre. In caso di fallimento inesca la procedura di riconnessione.

Da notare come nel caso dell'operazione di join è stato necessari introdurre una coda dei messaggi *joinQueueMSG* di tipo **Queue**$\prec Message \succ$. Questo in quanto un nodo fisico potrebbe in determinati istanti non avere le proprie strutture dati consistenti in quanto in fase di aggiornamento. Basta considerare il caso di aggiornamento della Routing Table. In questi casi le strutture risultano bloccate e pertanto i nuovi messaggi di ingresso dovranno essere gestiti attraverso una coda priopritaria rappresentata dall'istanza della classe **Queue**$\prec Message \succ$.

Terminando l'analisi delle classi che compongono il sistema troviamo alcune classi di utilità e configurazione dell'intero servizio XCone.

- **XConeMessageFactory**, fornisce le operazioni per la costruzione di tutti i messaggi di comunicazione da inoltrare nella rete.

- **XConeConfiguration**, possiede le configurazioni di base del servizio.

Da notare come nel caso della classe **XConeConfiguration** di seguito venga riportato un estratto dei parametri di configurazione;

- *DEFAULT_ROUTING_STYLE*, definisce la strategia di routing utilizzata per esempio Iterativa o Ricorsiva.

- *DEFAULT_ROUTING_ALGORITHM*, definisce il routing algorithm utilizzato per esempio Chord, Kademlia, Pastry, etc.

- *DEFAULT_MESSAGING_TRANSPORT*, definisce il protocollo di comunicazione utilizzato per esempio TCP, UDP.

- *DEFAULT_DIRECTORY_TYPE*, definisce la strategia di memorizzazione come per esempio BerkeleyDB, PersistentMap o VolatileMap.

- *DEFAULT_MAPPING_STRATEGY*, definisce la strategia di mapping come per esempio il MAX.

- *DEFAULT_DIGEST_STRATEGY*, definisce la strategia di digest come per esempio il BitVector Digest o Max Digest.

Da notare che alcune variabili si riferiscono a parametri richiesti per la configurazione di Overlay Weaver mentre le ultime due sono utilizzate per la configurazione del servizio XCone.

4.2.2 Diagrammi di sequenza

Illustriamo adesso le principali interazioni e sequenze di messaggi scambiati tra i nodi della rete. Da adesso in poi ogni riferimento al concetto di nodo è inteso essere il nodo fisico o peer della rete.

Operazione di JOIN

In figura 4.8 è mostrato il caso in cui un nuovo nodo denominato *newNode* voglia effettuare l'ingresso in rete. Per semplicità le operazioni riportate nel diagramma di sequenza iniziano dall'invocazione del metodo *joinXConeOverlay* sul nodo *newNode*. Bisogna però tenere presente che il *newNode* effettua come operazione preliminare la join nella rete DHT. Una volta stabilizzata la struttura dati DHT, attraverso una callback, viene restituito l'identificativo di un nodo già presente in rete denominato *lcaNode* (LCA Node). Da questo mo-

Messaggi	Descrizione
JOIN_CALL	Richiesta ingresso overlay XCone
JOIN_OK	Ingresso confermato
JOIN_CONTINUE	Ingresso non confermato
UPDATE_PARENT	Aggiornamento riferimento nodo fisico
UPDATE_DIGEST_INFO	Aggiornamento informazioni di digest

Tabella 4.2: Messaggi di Join

mento viene invocata la chiamata al metodo *joinXConeOverlay* con argomento il nodo *lcaNode*.

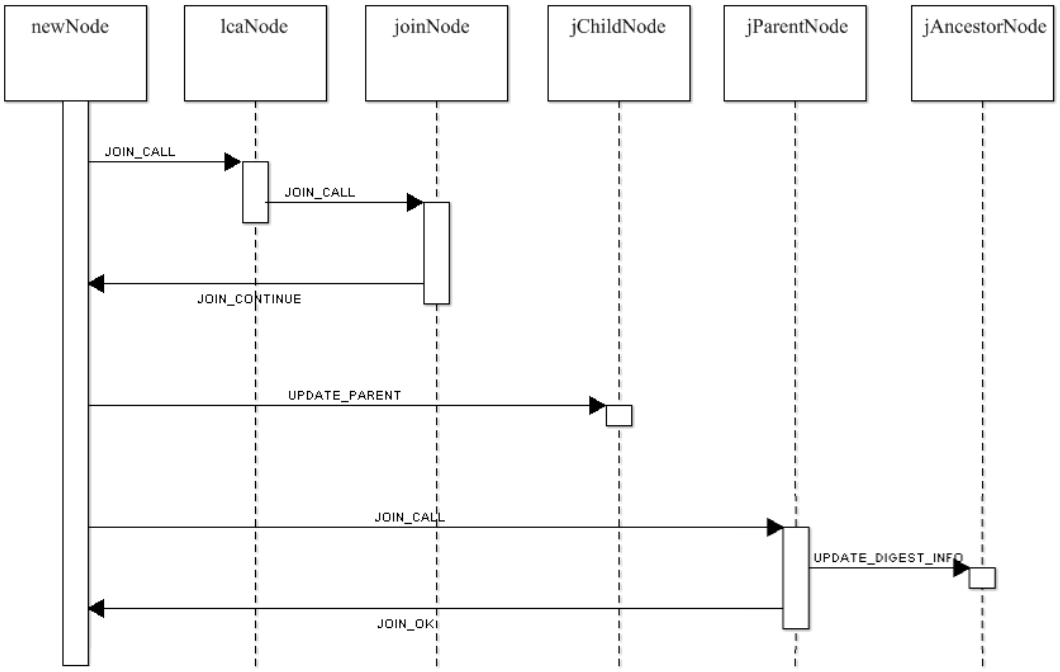

Figura 4.8: Diagramma di sequenza Join

Il *newNode* invia il messaggio di JOIN_CALL al proprio *lcaNode* inserendo come dati il valore della propria chiave e il livello logico dell'albero in cui incontra il nodo *lcaNode*. Una volta ricevuto il messaggio, *lcaNode* verifica se gestisce ancora il nodo logico con cui si incontra il nuovo nodo *newNode*. Questa situazione può essere chiarita attraverso un rapido esempio in figura

4.9. Supponiamo che il mapping dei nodi *newNode*, *lcaNode* e *joinNode* sia quello riportato in figura. Il *newNode* interseca nel proprio cammino dell'albero il nodo *lcaNode* nel nodo logico collocato al livello 2. Pertanto invia questa ulteriore informazione nel messaggio di JOIN_CALL. *lcaNode* ricevuto il messaggio verifica se gestice il nodo logico collocato al livello di intersezione con il *newNode*. Se non risulta più gestito allora provvede ad inoltrare il messaggio di join al proprio padre, nell'esempio il nodo *joinNode*.

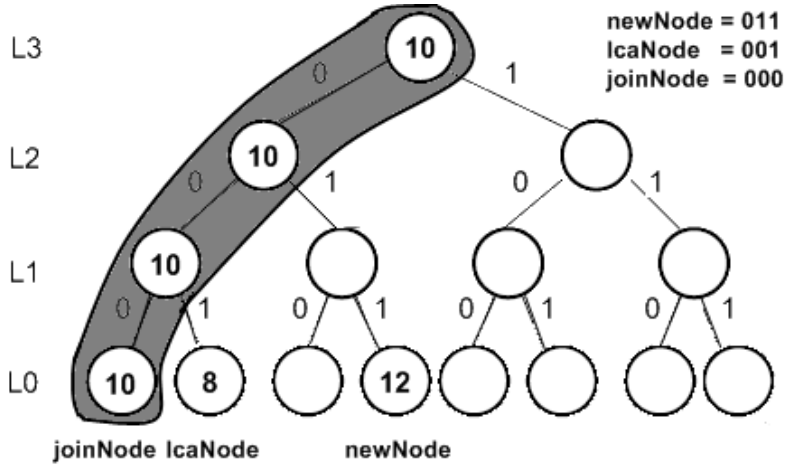

Figura 4.9: Problematica LCA

Ipotizzando tale situazione possiamo tornare a descrivere il comporamento dei nodi attraverso il diagramma di sequenza. Una volta ricevuto il messaggio di JOIN_CALL e verificato che il nodo logico di intersezione non risulta più gestito, *lcaNode* propaga il messaggio di join all'effettivo LCA identificato dal nodo *joinNode*. Il nodo *joinNode* a questo punto verificata la correttezza della proprietà di LCA applica la funzione di mapping per determinare se il nuovo nodo entrante sia il gestore del livello logico. Supponendo che il risultato confermi che il nodo *newNode* sia il nuovo gestore del livello, allora *joinNode* provvede ad inviare il messaggio JOIN_CONTINUE con la propria routing

table e successivamente ad aggiornare il riferimento al nuovo nodo padre, che diviene *newNode*.

Il *newNode* riceve il messaggio di JOIN_CONTINUE e provvede ad estrarre le informazioni sulla routing table del *joinNode*. Inizia a questo punto la fase di *"trickling"* dell'albero XCone. Attraverso la routing table ricevuta per tutti i livelli logici in cui *joinNode* risulta gestore, *newNode* provvede a diventare il nuovo gestore inviando agli eventuali nodi figlio, riferiti in figura con *jChildNode*, il nuovo riferimento a se stesso mediante un messaggio di UPDATE_PARENT. Il *newNode* ottiene quindi un riferimento al nodo che gestisce il nodo logico padre del nodo di livello più alto gestito da *joinNode* (*jParentNode*) ed invia nuovamente un messaggio di JOIN_CALL con la propria chiave e il nuovo livello logico risalito.

Il nodo *jParentNode* riceve il messaggio di join e supponendo che in questo caso la funzione di mapping confermi il *jParentNode* come gestore del livello, viene inviato al *newNode* il messaggio di risposta JOIN_OK concludendo la join di *newNode* nella rete XCone.

In realtà l'operazione di join per quanto riguarda il *jParentNode* non risulta ancora teminata. Infatti il *jParentNode* aggiorna la propria routing table con il nuovo nodo figlio nel rispettivo livello logico e provvede ad aggiornare l'informazione di digest che sintetizza le chiavi gestite dal sotto-albero logico radicato in *jParentNode*. Nel caso in cui questa informazione di digest risulti cambiata il *jParentNode* provvede ad aggiornare il proprio nodo padre, *jAncestorNode*, con un messaggio di UPDATE_DIGEST_INFO.

Tutti i possibili messaggi generati attraverso l'operazione di join sono stati raggruppati in tabella 4.2.

Operazione di CHANGE

Descriviamo adesso le operazioni effettuate per la procedura di aggiornamento. Come per la procedura di join illustriamo il procedimento attreverso il diagramma di sequenza in figura 4.10 e supponiamo che il nodo *chNode* effettui un cambiamento alla propria chiave.

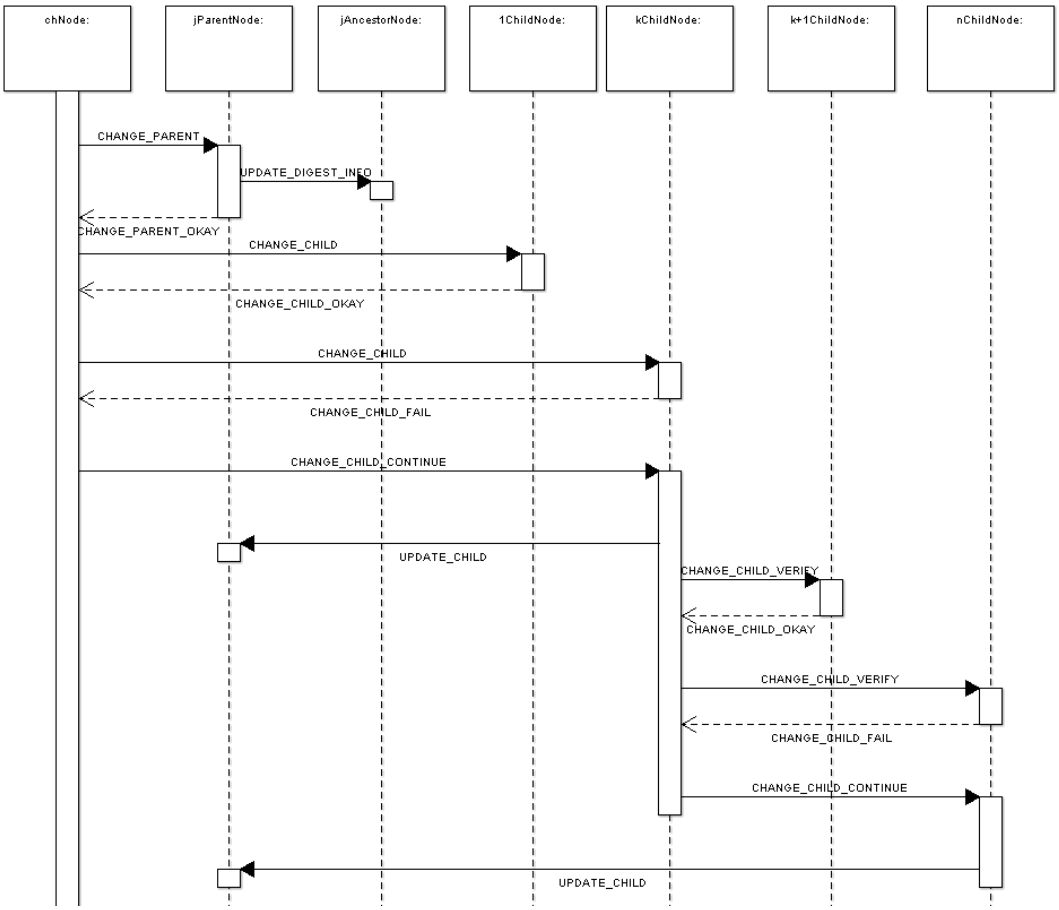

Figura 4.10: Diagramma di sequenza Change

La prima operazione effettuata da *chNode* consiste nel controllo del vincolo di mapping tra nodo padre e nodo figlio. Per effettuare tale controllo viene inviato il messaggio di CHANGE_PARENT con il nuovo valore della chiave. Se il nodo padre *jParentNode*, applicando la funzione di map-

Messaggi	Descrizione
CHANGE_PARENT	Controllo mapping figlio-padre
CHANGE_PARENT_OKAY	Relazione figlio-padre valida
CHANGE_PARENT_FAIL	Relazione figlio-padre non valida
CHANGE_CHILD	Controllo mapping padre-figlio
CHANGE_CHILD_CONTINUE	Relazione padre-figlio non valida
CHANGE_CHILD_OKAY	Relazione padre-figlio valida
CHANGE_CHILD_VERIFY	Verifica mapping padre-figlio
CHANGE_CHILD_FAIL	Relazione padre-figlio non valida
UPDATE_CHILD	Aggiornamento riferimento nodo padre

Tabella 4.3: Messaggi di Change

ping, conferma la relazione di mapping, viene inviato al *chNode* il messaggio CHANGE_PARENT_OKAY. Naturalmente *jParentNode* provvede ad aggiornare le informazioni di digest sulle chiavi contenute nel proprio sotto-albero logico e se necessario ad aggiornare l'informazione di digest del nodo padre, *jAncestorNode*, attraverso il messaggio di UPDATE_DIGEST_INFO.

Ritornato il controllo al *chNode* si inizia la verifica delle relazioni di mapping con tutti i nodi figli inviando anche in questo caso un messaggio di CHANGE_CHILD con il valore della chiave. I nodi figlio nel caso in cui la relazione sia ancora valida provvedono ad informare il *chNode* con il messaggio CHANGE_CHILD_OKAY, viceversa verrà inviato un messaggio di CHANGE_CHILD_FAIL. L'esempio in figura mostra entrambe le situazioni ed in particolare il caso di violazione nel nodo figlio *kChildNode*.

A questo punto il nodo *chNode* avendo rilevato che la relazione di mapping con il nodo figlio è violata, non potrà gestire i nodi logici a partire da quello del nodo figlio verso la radice. Il nodo *chNode* provvede ad inviare la propria routing table al *kChildNode* attraverso il messaggio CHANGE_CHILD_CONTINUE e contestualmente ad aggiornare il riferimento ad il nuovo padre, di livello inferiore, con il nodo *kChildNode*.

Il nodo *kChildNode* ricevendo la routing table inizia in maniera analoga

a quanto fatto nel caso di una join le operazioni di *"trickling"* dell'albero. Si verificano le proprietà di mapping con il resto dei nodi figli di *chNode* attraverso il messaggio CHANGE_CHILD_VERIFY. Se i nodi confermano la relazione viene restituito un messaggio di CHANGE_CHILD_OKAY, altrimenti si ha un messaggio di CHANGE_CHILD _FAIL. In quest'ultimo caso viene ripetuto il processo di invio della routing table e di aggiornamento al nuovo nodo padre.

La tabella 4.3 schematizza tutti i possibili messaggi generati.

Operazione di FIND

Come ultima operazione descriviamo la procedura di ricerca in XCone. Il diagramma di sequenza in figura 4.11 mostra l'operazione di ricerca a partire dalla situazione generica in cui il nodo *searchNode* abbia già ricevuto una query, originata dal nodo *queryNode*.

Il *searchNode* come passo iniziale provvede a controllare se la propria chiave soddisfa la query. In caso positivo invia il proprio identificativo al *queryNode* attraverso un messaggio del tipo FIND_RESULT. Il *queryNode* ricevuto il risultato provvede a replicare con un messaggio di FIND_RESULT_REPLY contentente il numero di elementi fino a quell'istante localizzati.

A questo punto il *searchNode*, ipotizzando che il numero di elementi non sia sufficiente, provvede ad esplorare i propri sotto-alberi alla ricerca di chiavi compatibili con la query. L'esplorazione dei sotto-alberi avviene in due fasi distinte. Nella prima fase vengono selezionati dal *searchNode* solamente i sotto-alberi che sulla base delle informazioni di digest locali possono contenere delle chiavi in grado di soddisfare la query. Nella seconda fase vengono inviati i messaggi di esplorazione FIND_CHILD. L'invio dei messaggi di esplorazione può avvenire in due modalità, la modalità di esplorazione parallela e quella sequenziale. Nella prima modalità vengono inoltrate in parallelo tutte le ri-

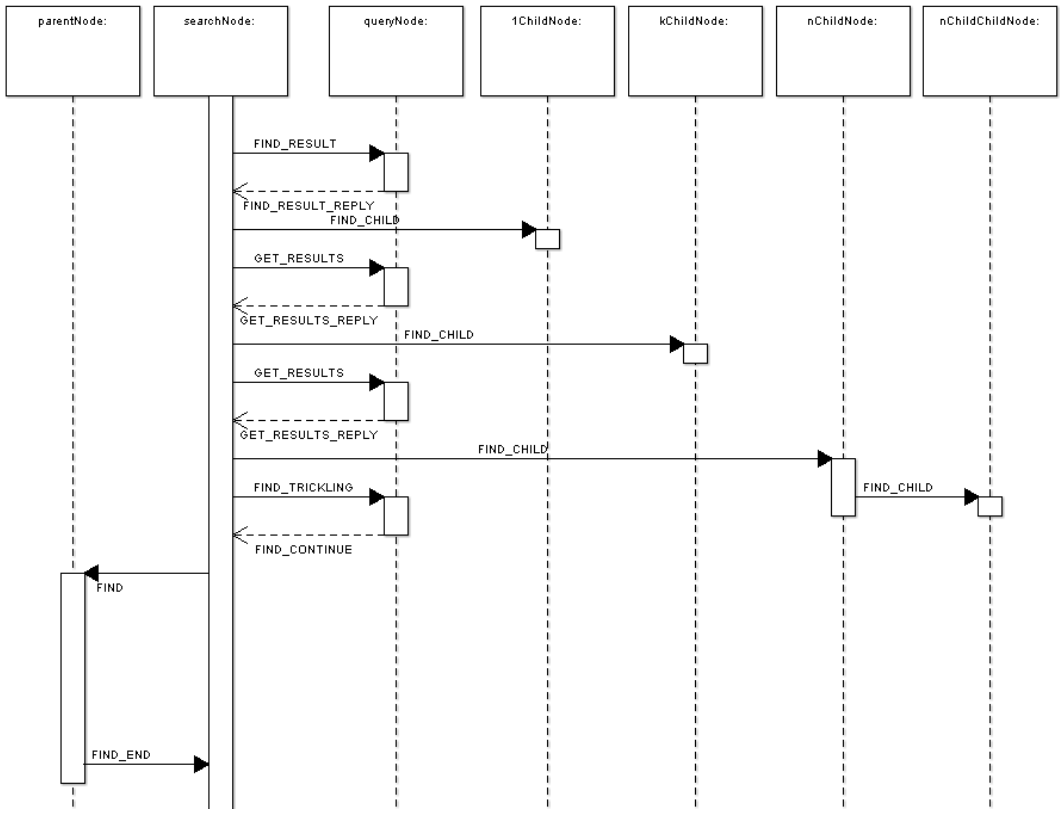

Figura 4.11: Diagramma di sequenza Find

Messaggi	Descrizione
FIND	Richiesta di ricerca
FIND_RESULT	Invio risultato compatibile
FIND_RESULT_REPLY	Numero risorse attualmente localizzate
FIND_CHILD	Richiesta di ricerca al nodo figlio
GET_RESULTS	Richiesta risorse attualmente localizzate
GET_RESULTS_REPLY	Numero risorse attualmente localizzate
FIND_TRICKLING	Sospensione della ricerca
FIND_CONTINUE	Riattivazione della ricerca
FIND_END	Terminazione della ricerca

Tabella 4.4: Messaggi di Find

chieste di FIND_CHILD. Nella seconda modalità, illustrata in figura, prima di inviare un messaggio di FIND_CHILD al nodo che gestisce il sotto-albero successivo viene verificato che il numero di risultati attualmente raccolti non sia sufficiente.

Per effettuare tale controllo il *searchNode* invia un messaggio di GET_RESULTS al *queryNode* ed attende come risposta il messaggio di GET_RESULTS_REPLY con il numero di risorse raccolte. Ipotizzando che il numero di nodi localizzati sia ancora non sufficiente, *searchNode* provvede ad inoltrare la richiesta di esplorazione al sotto-albero successivo. Supponiamo che terminata l'esplorazione dei sotto-alberi il *searchNode* rilevi nuovamente che il numero di elementi raccolti non sia sufficiente. A questo punto viene sospesa la ricerca inviando il messaggio di FIND_TRICKLING al *queryNode*.

Il *queryNode* ha il controllo sulla propagazione delle operazioni di ricerca verso i nodi fisici collocati nei livelli alti dell'albero logico XCone. Se non sono stati raccolti un numero sufficiente di elementi o è trascorso un tempo di attesa in cui sia ragionevole considerare l'esplorazione dei sotto-alberi conclusa, il *queryNode* invia un messaggio FIND_CONTINUE al nodo sospeso per far riprendere le operazioni di find.

Il *searchNode* ricevendo un messaggio di FIND_CONTINUE provvede im-

mediatamente ad inoltrare la richiesta al proprio nodo padre attraverso il messaggio FIND. Da questo punto in poi il nodo padre replicherà il comportamento del *searchNode* e nel caso in cui i nodi raccolti siano sufficienti o sia stata raggiunta la radice verrà inviato il messaggio di FIND_END al *queryNode* che concluderà le operazioni di ricerca.

I possibili messaggi scambiati durante l'operazione sono raggruppati in tabella 4.4.

Capitolo 5

Risultati sperimentali

In questo capitolo dopo aver descritto i tests realizzati saranno presentati i risultati sperimentali ottenuti. Saranno valutate le prestazioni delle diverse funzioni di digest, la distribuzione del carico tra i nodi fisici e il traffico di controllo generato.

5.1 Introduzione

Per la valutazione di XCone è stato utilizzato l'emulatore presente nel framework ed attraverso diversi scenari è stato possibile realizzare agevolmente tutti i tests per analizzare le prestazioni di XCone nelle diverse configurazioni.

I tests effettuati in XCone non risultano confrontabili con quelli ottenuti in [22] in quanto come già ampiamente illustrato XCone differisce da Cone sotto due aspetti fondamentali:

1. L'esecuzione di range query *"Find k values with* $S \leq X \leq V$*"*
2. L'esplorazione di sotto-alberi in relazione alla funzione di digest utilizzata $P \leftarrow f(N, M, d(M))$

Inoltre tali estensioni introducono nei tests XCone alcuni parametri aggiuntivi quali:

- L'intervallo di valori [a,b] in cui cercare risorse
- Il numero di risorse k
- Il rapporto tra l'ampiezza dell'intervallo, il numero di risorse cercate e le risorse disponibili nella rete

Nella prossima sezione verrà illustrata la metodologia utilizzata, gli scenari utilizzati ed infine sintetizzati ed analizzati i risultati ottenuti.

5.2 Organizzazione dei Tests

Come già accennato l'ambiente di valutazione dei test consiste nell'emulatore fornito da OverlayWeaver. I tests effettuati sono stati progettati per analizzare le seguenti proprietà di XCone:

- **Analisi delle funzioni di digest**, il cui obiettivo consiste nell'analizzare la quantità di nodi coinvolti nell'esecuzione di range query. In particolare nel valutare la quantità di nodi fisici esplorati dalle diverse funzioni di digest.
- **Analisi del data traffic**, in questo test l'obiettivo principale consiste nell'analizzare la suddivisione del carico tra un insieme di nodi in grado di risolvere una determinata range query. Nello specifico si vuole analizzare la proprietà di *Data Traffic*, per come è stata introdotta nella sezione 2.4.6, in cui si esamina il fattore di sbilancimento tra il nodo contattato il minor numero di volte e quello contattato maggiormente.
- **Analisi del control traffic**, il cui obiettivo consiste nell'analizzare la distribuzione dei messaggi di controllo, introdotti per l'esecuzione di una range query, su tutti i nodi della rete. In particolare si vuole analizzare

la proprietà di *Control Traffic*, introdotta nella sezione 2.4.6, in cui viene analizzato il fattore di sbilancimento tra il nodo contattato il minor numero di volte e quello contattato maggiormente.

La rete DHT utilizzata per effettuare i tests è CHORD con identificativi da 160bits. Il domino delle chiavi è definito dal range [0,99] e la distribuzione utilizzata è di tipo Zipf[38]. Tale distribuzione modella la situazione in cui un elevato numero di chiavi viene a concentrarsi verso una porzione ristretta di valori del dominio. La figura 5.1 visualizza la distribuzione delle chiavi evidenziando come la maggioranza delle chiavi sia concentrata in valori prossimi allo zero.

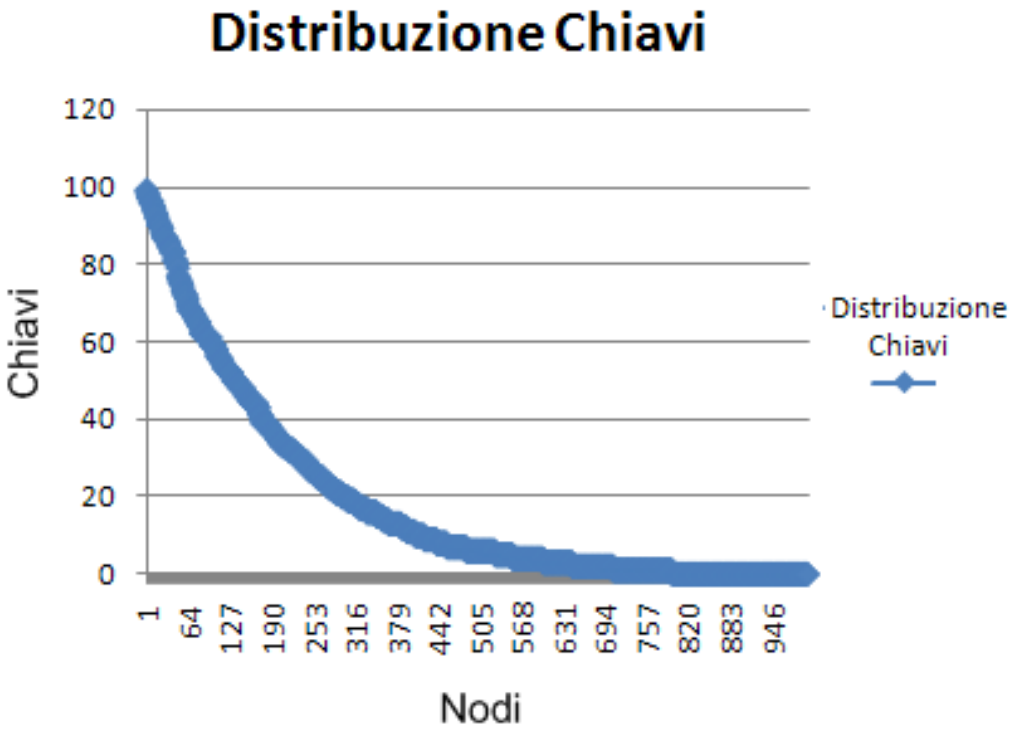

Figura 5.1: Distribuzione delle chiavi di tipo Zipf

Per quanto riguarda la definizione delle range query è stato utilizzato il seguente metodo. Definita una range query attraverso il range $[a,\ b]$ e un

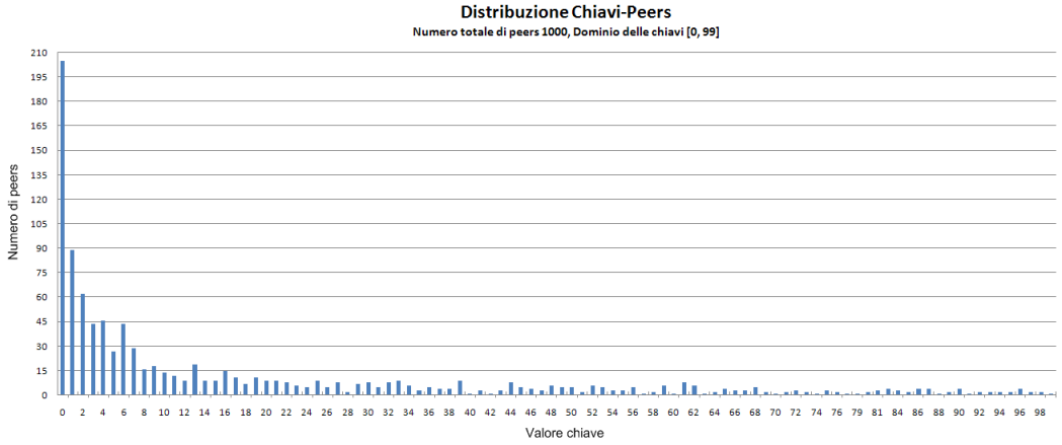

Figura 5.2: Assegnazione chiavi/peer

numero k di risorse, il limite inferiore a è determinato attraverso una distribuzione zipfiana identica alla distribuzione delle chiavi; il limite superiore b è determinato a partire da un valore casuale c. Il parametro c è determinato in maniera uniforme tra i soli valori definiti nella porzione di dominio da a al valore massimo. In questo modo viene determinata l'ampiezza del range e definito di consenguenza il parametro b come segue: b=a+c. Il terzo parametro k è calcolato in maniera probabilistica. La funzione di ripartizione F(X) di una particolare distribuzione associa per ciascun valore x la probabilità che *"il valore assunto dalla variabile X risulti minore o uguale ad x"*. Pertanto definito un intervallo [a,b] la probabilità che le chiavi assumino tali valori è uguale a F(b) - F(a). Assunto che il numero di chiavi presenti nel sistema sia uguale al numero di nodi T, il numero di chiavi stimate all'interno di un range è calcolato come $K_{Stim} = T * (F(b) - F(a))$. Il parametro k è quindi selezionato come una frazione delle chiavi stimate:

$$k = \alpha K_{Stim} \text{ con } 0 \prec \alpha \leq 1$$

5.3 Analisi funzioni di digest

Nell'esecuzione di questo test sono stati emulati 1000 nodi e selezionati 100 nodi in maniera casuale il cui compito è generare una range query generata con la metodologia descritta nella sezione precedente. I parametri analizzati sono stati il numero di nodi esplorati per ogni query, l'analisi dei nodi selezionati per ogni query e di conseguenza la stima dei nodi aggiuntivi visitati. Il test è stato realizzato al variare delle funzioni di digest e del parametro alpha ($\alpha = 1, 0.5, 0.25$).

Le funzioni di digest analizzate sono:

- **Digest MAX**, in cui viene mantenuto tra un insieme di valori il valore il massimo.
- **Digest BitVector Equal Witdh**, in cui viene mantenuto un BitVector con nove punti di separazione equamente distribuiti nel dominio dei valori [10,20,30,40,50,60,70,80,90].
- **Digest BitVector Zipf**, in cui viene mantenuto un BitVector con nove punti di separazione ma in grado di modellare la distribuzione delle chiavi sul dominio dei valori [20,40,50,60,70,75,80,85,90,95].

Le figure 5.3 e 5.4 mostrano i risultati sperimentali ottenuti. In particolare la figura 5.4 mostra la percentuale di nodi aggiuntivi visitati rispetto al totale dei nodi effettivamente localizzati.

Dall'andamento dei grafici possiamo notare come la funzione di digest MAX riduca il numero di nodi esplorati rispetto alle altre strategie utilizzate, se pur lievemente. Questo risultato apparentemente controintuitivo risulta essere invece legittimo. Per giustificare tale comportamento occorre mettere in relazione la distribuzione delle chiavi con le differenze tra le diverse metodologie

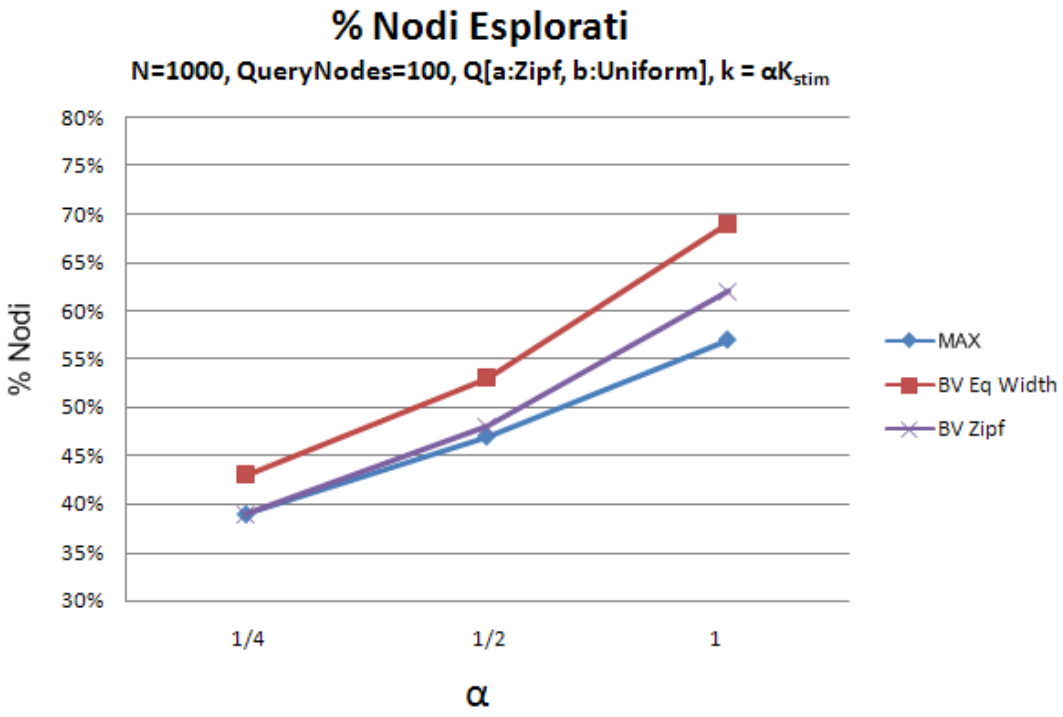

Figura 5.3: Percentuale nodi esplorati

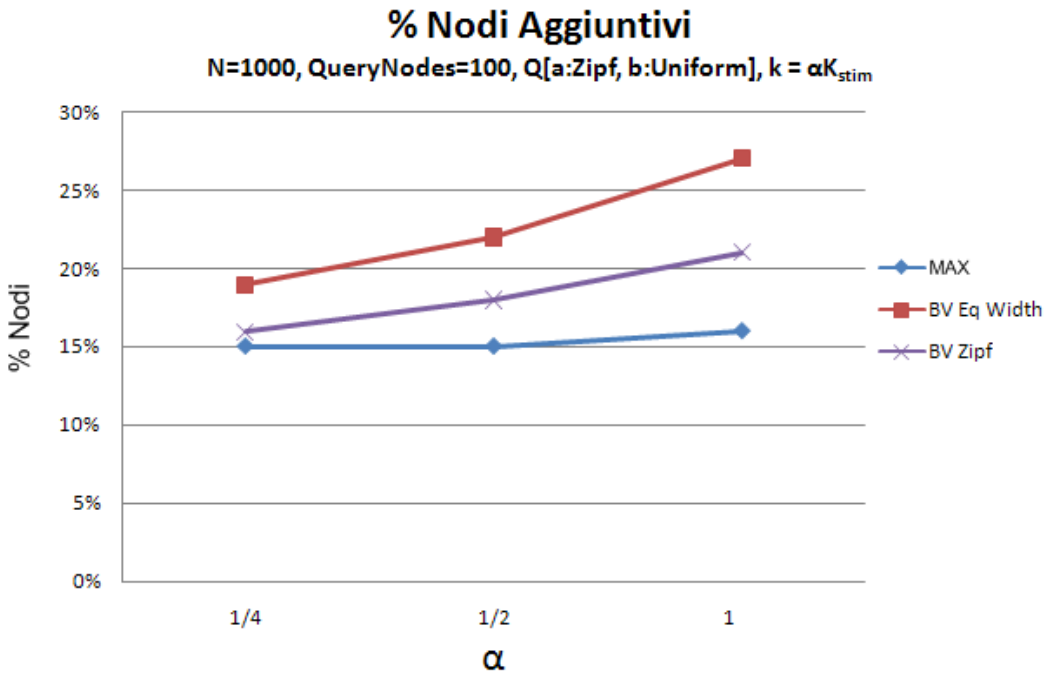

Figura 5.4: Percentuale nodi aggiuntivi

di digest. La maggioranza delle chiavi come visibile in figura 5.1 è localizzata nella coda della distribuzione e pertanto con valori prossimi a zero. La funzione di digest MAX per decidere se esplorare un sotto-albero effettua un controllo tra il massimo valore contenuto come informazione di digest e il limite inferiore della range query a. In questa maniera viene effettuato un taglio esatto sul limite inferiore ma nessun taglio sul limite superiore b portando potenzialmente all'esplorazione di tutti i nodi con valori anche maggiori di b. Il BitVector elimina questo comportamento codificando le range query attraverso un insieme di bits nei rispettivi intervalli. A questo punto avendo intervalli ad ampiezza fissata può succedere che il valore a venga codificato in un intervallo che include potenzialemente anche valori minori. Pertanto sebbene il BitVector consenta di non visitare i nodi con valori maggiori dell'intervallo in cui è codificato b, nel caso di una distribuzione come quella considerata il numero

di nodi scartati non riesce a compensare la percentuale di nodi introdotti per l'approssimazione fatta dall'intervallo.

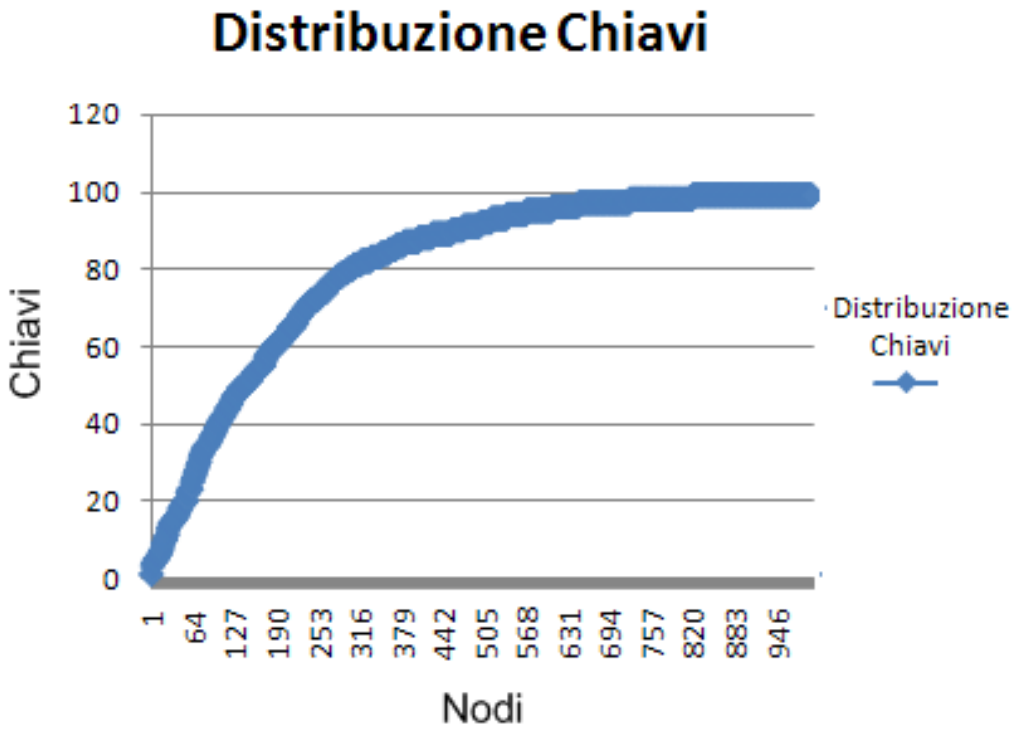

Figura 5.5: Distribuzione delle chiavi con Zipfiana Inversa

Avendo chiarito il fenomeno si è variata la distribuzione delle chiavi invertendo la distribuzione zipfiana in modo tale da avere la coda della distribuzione verso i valori alti del dominio. I grafici 5.6 e 5.7 illustrano in questo secondo caso i risultati sperimentali ottenuti.

5.3.1 Conclusioni

L'analisi delle diverse funzioni di digest ha confermato come le strategie di digest più informate tendono ad ottimizzare il numero di nodi esplorati. Sebbene da una prima analisi le tecniche più informate abbiano avuto un incremento dei nodi aggiuntivi esplorati, in ultima analisi si è potuto osservare come questo

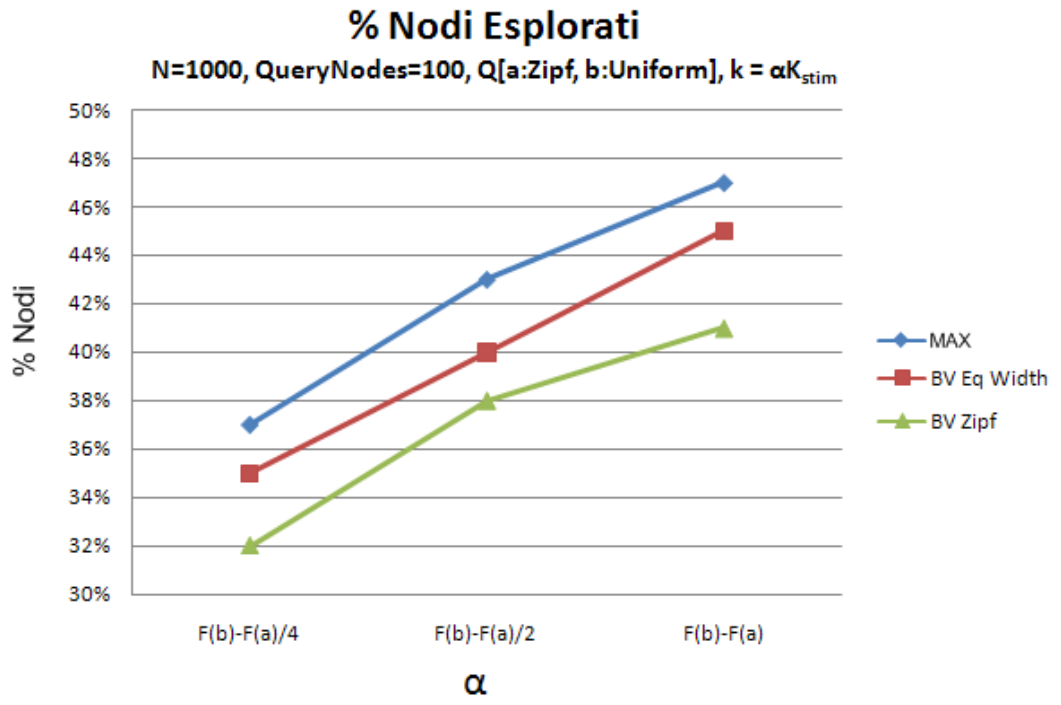

Figura 5.6: Percentuale nodi esplorati

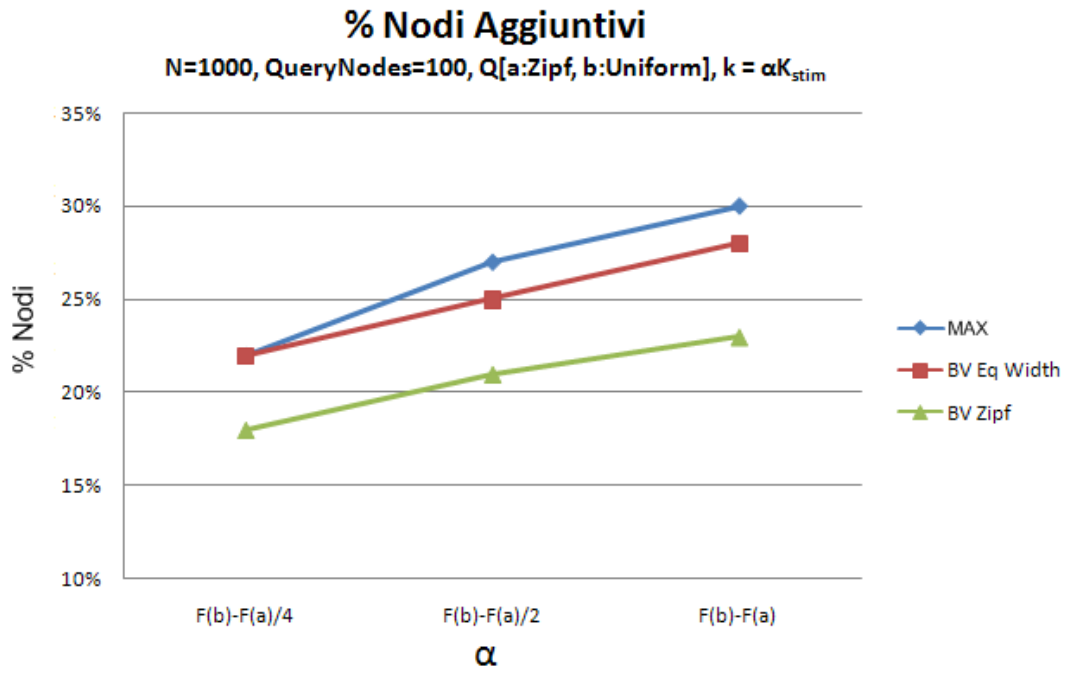

Figura 5.7: Percentuale nodi aggiuntivi

fenomeno sia stato condizionato dalla specifica proprietà della distribuzione dei dati. Nel secondo ciclo di test si è infatti rilevato che le tecniche di digest più informate apportino un reale guadagno sul numero di nodi esplorati.

In conclusione è emerso come la conoscenza della distribuzione possa migliorare le performance e l'introduzione di bitvector più accurati o tecniche di digest più evolute possano migliorare le prestazioni.

5.4 Analisi bilanciamento del carico

Nell'esecuzione del test sono stati emulati 100 nodi, si considerano quindi 10 range query ed ogni range query viene eseguita a turno in modo round robin da ognuno dei nodi fisici della rete, per un totale di 1000 query. La strategia di digest utilizzata è basata sui BitVector con dieci punti di separazione in modo tale da modellare al meglio la distribuzione Zipfiana. Ad ogni iterazione ogni nodo compatibile con la range query ha memorizzato il numero di volte in cui è stato selezionato.

It. 1	**It. 2**	**It. 3**	**It. 4**	**It. 5**
57:77 1	77:97 2	59:78 1	74:98 3	84:99 5
It. 6	**It. 7**	**It. 8**	**It. 9**	**It. 10**
90:99 5	96:98 3	78:89 2	98:99 5	78:92 3

Tabella 5.1: Queries (a:b k) effettuate nel test

Si è pertanto valutato ad ogni ciclo completo di una iterazione il contatore di ogni nodo selezionato. In questa maniera è stato possibile analizzare il data traffic della rete. In particolare la figura 5.8 illustra i risultati ottenuti per una singola iterazione mentre in figura 5.9 è analizzato il fattore di sbilanciamento per ogni iterazione.

Il fattore di sbilanciamento è calcolato come il rapporto tra il numero di contatti ricevuti dal nodo che è stato selezionato il maggior numero di volte

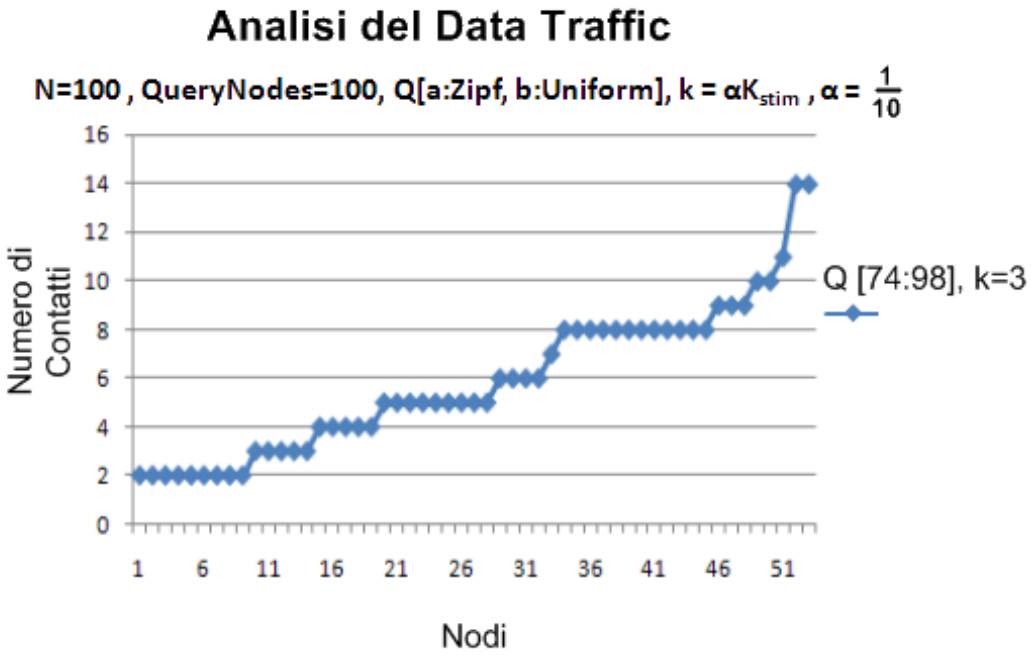

Figura 5.8: Risultato singola iterazione

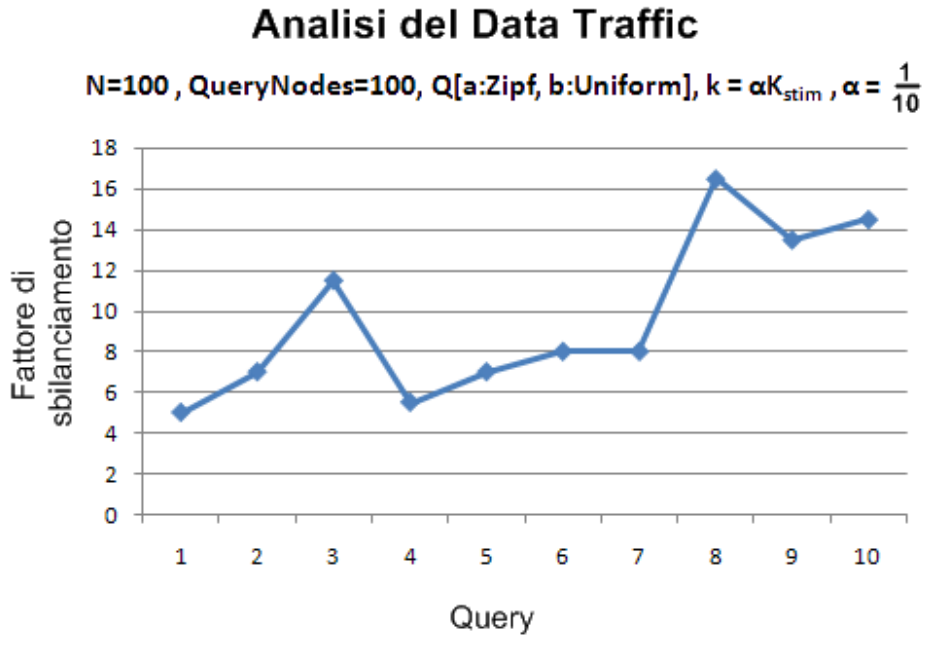

Figura 5.9: Analisi Fattore di Sbilanciamento Data Traffic

ed il numero di contatti del nodo selezionato il minor numero di volte. In altre parole viene analizzato se le queries tendono a concentrarsi verso un solo sotto-insieme dei nodi che soddisfano una particolare range query.

5.4.1 Conclusioni

Il fattore di sbilanciamento è stato in media di 9.85, in realtà di poco sopra allo sbilanciamento previsto h=7 con T=100. Il caso di maggiore sbilanciamento si è avuto nella ottava iterazione in cui la range query richiesta è stata di Q[78:89] con k=2. In questo caso un nodo all'interno del range è stato selezionato solamente 2 volte mentre un altro nodo è stato selezionato ben 33 volte creando un fattore di sbilanciamento pari a 16,5.

Possiamo quindi osservare come il comportamento di XCone è stato in media di poco superiore a quanto previsto al caso ideale. Il carico si è quindi distribuito tra i nodi in maniera bilanciata a meno di un fattore di sbilanciamento h.

5.5 Analisi traffico di rete

Nell'esecuzione del test sono stati emulati 100 nodi. Si considerano quindi 10 range query ed ogni query viene eseguita da tutti i nodi in maniera round-robin. Vengono quindi eseguite in totale 1000 query. Ad ogni iterazione ogni nodo ha incrementato un contatore per tenere traccia di quante volte è stato contattato sia direttamente che indirettamente per effetto di una range query.

It. 1	**It. 2**	**It. 3**	**It. 4**	**It. 5**
57:77 1	77:97 2	59:78 1	74:98 3	84:99 5
It. 6	**It. 7**	**It. 8**	**It. 9**	**It. 10**
90:99 5	96:98 3	78:89 2	98:99 5	78:92 3

Tabella 5.2: Queries (a:b k) effettuate nel test

Si è pertanto valutato al termine di ogni ciclo di iterazione i contatori di tutti i nodi della rete. In questo modo è stato possibile analizzare il control traffic generato nella rete. In particolare la figura 5.10 illustra i risultati ottenuti per una singola iterazione, mentre in figura 5.11 è analizzato il fattore di sbilanciamento in ogni iterazione.

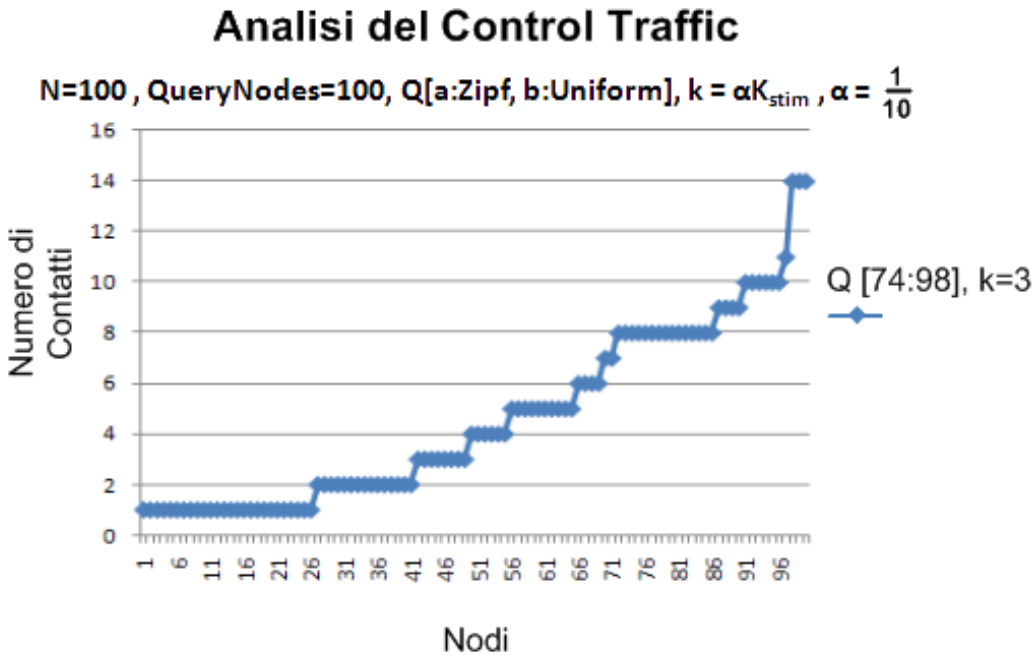

Figura 5.10: Esempio andamento Control Traffic

Anche in questo caso il fattore di sbilanciamento è calcolato come il rapporto tra il nodo contattato il maggior numero di volte e il nodo contattato il minor numero di volte.

5.5.1 Conclusioni

Il fattore di sbilanciamento massimo è stato in media di 26.5, decisamente al di sopra dello sbilanciamento previsto al caso ideale. In questo modo si è potuto costatare come la selettività introdotta dall'ampiezza del range e dal numero di

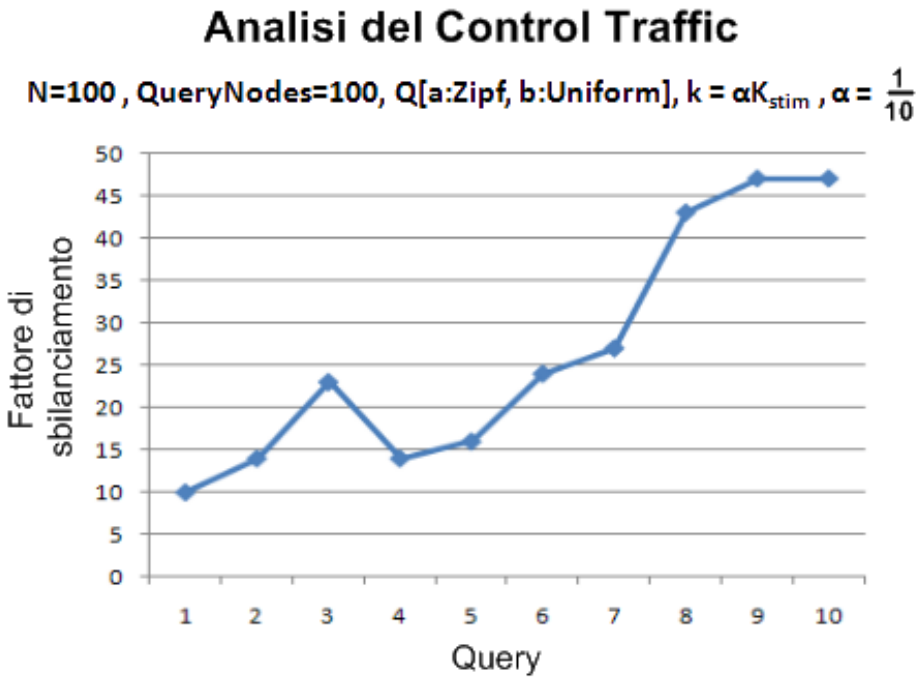

Figura 5.11: Analisi Fattore di Sbilanciamento Control Traffic

risorse richieste porta come previsto ad una esplorazione maggiore dell'albero. Questo comportamento porta inevitabilmente a caricare una minoranza di nodi collocati verso i livelli alti dell'albero XCone.

Capitolo 6

Considerazioni finali

6.1 Conclusioni

Il contributo che questa tesi ha cercato di dare è relativo alla realizzazione di un sistema P2P in grado di gestire efficacemente l'espressività delle operazioni di ricerca basate su range query. Il sistema proposto è basato su una struttura di tipo gerarchico progettata in maniera distribuita e tale da consentire diverse strategie di mapping ed aggregazione delle informazioni.

L'introduzione di un nuovo approccio è conseguenza di una attenta analisi delle proposte attualmente presenti in letteratura. In particolare prendendo spunto dai risultati ottenuti in [22] è stato possibile realizzare il sistema XCone introducendo le soluzioni innovative che garantiscono una maggiore espressività delle operazioni di ricerca.

Per quanto riguarda l'implementazione, il framework utilizzato è stato Overlay Weaver. Il vantaggio nell'adozione di tale framework è stato sicuramente la realizzazione di una struttura estremamente modulare che minimizza lo sviluppo del codice XCone.

I primi test effettuati hanno evidenziato come l'introduzione delle tecniche di digest abbia apportato un reale guadagno delle prestazioni nell'esplorazione delle risorse. Si sono altresì analizzate le proprietà di bilanciamento della

struttura. I risultati sperimentali hanno confermato la stretta dipendenza tra il numero di risorse richieste e l'ampiezza degli intervalli di ricerca e evidenziato fattori di sbilanciamento mediamente contenuti.

6.2 Sviluppi futuri

Nell'attuale versione di XCone si è utilizzata la funzione di aggregazione massimo come funzione di mapping dei nodi logici dell'albero ai peer della rete. In realtà XCone è una struttura estremamente generica pertanto uno dei possibili sviluppi consiste nell'analizzare diverse funzioni di mapping. Per esempio il mapping potrebbe tener conto di altri fattori come la banda di connessione dei vari nodi, la potenza di calcolo o anche il carico di nodi logici gestito.

Per quanto riguarda il problema del bilanciamento del carico, una possibile proposta interessante consiste nelle tecniche di bilanciamento basate sulle *Load Balance Matrix* (LBM) introdotte in [14]. Secondo questo approccio la funzione di mapping dei nodi logici potrebbe suddividere tra più peer la gestione dello stesso nodo logico. Questa soluzione risulta valida soprattutto per bilanciare il carico dei peer che gestiscono nodi situati verso la radice dell'albero XCone.

Infine per quanto riguarda il supporto alle range query multi-attributo è stata analizzata attentamente una possibile soluzione che include l'impiego delle space filling curves. L'idea consiste nel generare per ogni risorsa, descritta da un'insieme di attributi, una chiave surrogata ottenuta dall'applicazione della funzione di space filling. La chiave surrogata verrà quindi utilizzata per la costruzione dell'albero XCone. In particolare distribuendo le chiavi surrogate tra i peer dell'albero si annullerebbe il limite delle space filling curves (per esempio presente in [15]) che concentra i dati nel caso di distribuzioni non uniformi verso un sotto-insieme di peer della rete.

Dopo aver costruito la rete XCone, le ricerche per range query di tipo multidimensionale dovranno essere opportunamente trattate. In particolare la range query identifica una zona dello spazio multidimensionale. All'interno di tale spazio vi è definita una funzione di space filling che tocca ogni possibile valore del dominio una sola volta senza mai incrociarsi. La range query multidimensionale consiste nell'individuare porzioni di tale curva e selezionare i relativi dati. Naturalmente per le proprietà della space filling curves la curva ricopre un intero dominio dei valori di uno spazio in maniera indipendente dalla concentrazione effettiva dei dati. Occorre pertanto introdurre un meccanismo che guidi la raffinazione degli intervalli della curva in maniera guidata dai dati. Un procedimento analogo viene proposto già in [15] dal sistema Squid.

XCone risulta essere progettato in maniera modulare, in particolare per quanto riguarda la gestione delle chiavi di ogni nodo si ha la presenza di un *"Key Manager"*.

6.3 Ringraziamenti

La stesura della presente tesi ha come significato non solo il frutto finale di un lungo percorso di studi ma racchiude anche un percorso di vita segnato da grandi soddisfazioni e al tempo stesso da tanti sacrifici. Mi sembra quindi doveroso ringraziare tutti coloro che ne sono stati partecipi. In primo luogo ringrazio la mia famiglia e la mia ragazza per avermi saputo sostenere e consigliare nei momenti difficili. Ringrazio i miei colleghi dell'@System, del CVSLab e tutti i miei amici per i confronti avuti e per avermi pazientemente sopportato. Non per ultimi di importanza vorrei ringraziare la professoressa Laura Ricci e il professore Massimo Coppola per avermi dato l'opportunità di realizzare la presente tesi in un ambito di ricerca estremamente stimolante.

Concludo con la mia personale speranza che questo sia il punto di partenza di un nuovo percorso sia professionale che di vita.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] R. Matei, A. Iamnitchi, and P. Foster. Mapping the gnutella network. *Internet Computing, IEEE*, 6(1):50–57, 2002.

[2] Napster. http://www.napster.com/, 2006.

[3] The bittorrent p2p file-sharing system: Measurements and analysis. pages 205–216. 2005.

[4] Li Gong. Jxta: a network programming environment. *Internet Computing, IEEE*, 5(3):88–95, 2001.

[5] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. pages 112–120, 2003.

[6] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. Seti@home-massively distributed computing for seti. *Computing in Science & Engineering*, 3(1):78–83, 2001.

[7] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[8] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[9] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, Frans M. Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, February 2003.

[10] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[11] C. Buragohain, D. Agrawal, and S. Suri. A game theoretic framework for incentives in p2p systems. pages 48–56, 2003.

[12] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. pages 118–128. 2003.

[13] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[14] Jun Gao. *A Distributed and Scalable Peer-to-Peer Content Discovery System Supporting Complex Queries*. PhD thesis, Computer Science, Carnegie Mellon University, October 2004.

[15] C. Schmidt and M. Parashar. Analyzing the search characteristics of space filling curve-based indexing within the squid p2p data discovery system. Technical report, Rutgers University, December 2004.

[16] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: a multi-attribute addressable network for grid information services. In *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 184–191, 2003.

[17] H. V. Jagadish, Beng C. Ooi, and Quang H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st Internatio-*

nal Conference on Very Large Data Bases (VLDB '05), pages 661–672. VLDB Endowment, 2005.

[18] Sriram Ramabhadran, Joseph Hellerstein, Sylvia Ratnasamy, and Scott Shenker. Prefix hash tree - an indexing data structure over distributed hash tables.

[19] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Record*, 30(1):19–24, 2001.

[20] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. *Lecture Notes in Computer Science*, 1832:20–??, 2000.

[21] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.

[22] G. Varghese, R. Bhagwan, and G. M. Mahadevan, P.and Voelker. Cone: A distributed heap-based approach to resource selection. Technical report, University of California, 2004.

[23] M. Marzolla, M. Mordacchini, and S. Orlando. Tree vector indexes: efficient range queries for dynamic content on peer-to-peer networks. In *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, pages 8 pp.+, 2006.

[24] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference*

on Embedded networked sensor systems, pages 239–249, New York, NY, USA, 2004. ACM.

[25] Natalya F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, December 2004.

[26] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks, 2002.

[27] D. S. Milojicic, V. Kalogeraki, R. Lukose, and K. Nagarajan. Peer-to-peer computing. Technical report, HP Labs, 2002.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[29] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks, 2001.

[30] 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.

[31] Petar Maymounkov and David Maziã¨res. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. 2002.

[32] Hans Sagan. *Space-Filling Curves*. Springer, 1 edition, September 1994.

[33] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. *Lecture Notes in Computer Science*, 1832:20–??, 2000.

[34] F. Korn, B. U. Pagel, and C. Faloutsos. On the "dimensionality curse" and the "self-similarity blessing". *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):96–111, 2001.

[35] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. Overlay weaver: An overlay construction toolkit. *Computer Communications*, (2):402–412, February. Framework available at `http://overlayweaver.sourceforge.net`.

[36] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays, 2003.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

[38] M. E. J. Newman. Power laws, pareto distributions and zipf's law, December 2004.