UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Specialistica in Scienze e Tecnologie Informatiche

Tesi di Laurea

Un sistema gerarchico basato su Voronoi per la risoluzione di query multiattributo

Relatori:
Prof. Laura Ricci
Dott. Ranieri Baraglia

Candidato:
Luca Ferrucci

Anno Accademico 2008/2009

Alla mia famiglia per la pazienza che hanno
mostrato in questi lunghi anni,
alla mia ragazza Laura per avermi sempre
sostenuto e per aver creduto in me,
a Matteo e a Michele per il sostegno che mi
hanno dato per la stesura
e a tutti i miei amici e compagni presenti e futuri
grazie di cuore!!!!

Indice

1 Introduzione **1**
1.1 Strutturazione dei capitoli della tesi 6

2 Stato dell'arte **8**
2.1 Sistemi non strutturati 9
2.1.1 Napster 10
2.1.2 Gnutella 0.4 14
2.1.3 Gnutella 0.6 18
2.1.4 P2P con RoutingIndex 22
2.1.5 Conclusioni sui metodi non strutturati 24
2.2 Sistemi strutturati 25
2.2.1 Chord 27
2.2.2 CAN 29
2.2.3 MAAN 32
2.2.4 SWORD 34
2.2.5 VORONET 36
2.2.6 SWAM-V 42
2.2.7 VoRaQue 44
2.2.8 Conclusioni sui metodi strutturati 48

3 Hierarchical VoRaQue: l'architettura **50**
3.1 Motivazioni 50
3.2 Reti gerarchiche basate su SuperPeer 56
3.3 HVoraque: struttura generale 58
3.4 HVoraque: le operazioni 63
3.4.1 Le strutture dati 63
3.4.2 Query 64
3.4.3 Inserzione 68
3.4.4 Elezione di SuperPeer 80
3.4.5 Studio della complessità degli algoritmi 90

4 Implementazione **101**
4.1 Query: implementazione 102
4.2 Inserzione: implementazione 109
4.3 Elezione: implementazione 119

5 Test **124**
5.1 Strumenti: PeerSim e Vast 124
5.1.1 PeerSim 124
5.1.2 VAST 127
5.2 Hierarchical Voraque: misurazioni effettuate 127
5.3 Parametri dei test e risultati 130
5.3.1 Inserimento: risultati dei test 130
5.3.2 Query: risultati dei test 137

6 Conclusioni e sviluppi futuri **145**

Bibliografia **148**

Elenco delle tabelle

5.1 Numero di siti fissati al variare della grandezza della rete 132

Capitolo 1

Introduzione

Le applicazioni distribuite basate sul modello P2P[1] (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 risultano 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 [3] 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 fisica, 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, ma non esistono informazioni relative alla loro locazione e la ricerca risulta essere difficoltosa a causa della mancanza di organizzazione all'interno del sistema. Questo è vero in particolare per Gnutella, dove la decentralizzazione e la mancanza di una strutturazione logica della rete ha costretto ad utilizzare un algoritmo di flooding per risolvere le query, riducendone enormemente la scalabilità con l'aumentare della dimensione della rete. Per limitare gli effetti del flooding, viene introdotto in

questi sistemi un limite massimo di nodi attraversabili, il TTL, ma in questo modo si vanno a perdere potenziali risposte alle query.

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 maggior parte di esse utilizza una struttura chiamata **DHT**, come [4, 5, 6]. L'aspetto chiave delle DHT[1] è l'utilizzo di funzioni hash per garantire il bilanciamento del carico. I nodi ottengono degli **identificatori** appartenenti ad uno spazio uniformemente popolato. Inoltre, una funzione hash applicata alla descrizione di una risorsa restituisce la sua **chiave**, che appartiene allo stesso spazio degli identificatori dei nodi, permettendo di allocare la risorsa sulla DHT. Le DHT permettono di superare i limiti dei sistemi P2P non strutturati, poichè ogni nodo mantiene una tabella di routing di dimensione $O(\log N)$, e se una risorsa è presente nel sistema verrà sicuramente localizzata in $O(\log N)$ hop, dove N rappresenta il numero di nodi che fanno parte della rete. Le DHT si sono rilevate il miglior sistema per risolvere le **query esatte**, ma non sono in grado di risolvere efficientemente query per range di valori. Ciò dipende dall'uso di funzioni hash che mappano le risorse in modo casuale sulla rete e non consentono di conservare alcuna nozione di località.

Negli ultimi anni si è assistito all'utilizzo di applicazioni P2P in nuovi contesti come il Gaming online o il Grid Computing. 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 strutturati. Considerando per esempio le applicazioni P2P in un contesto come il Grid Informazion Service, cioè il sistema che gestisce la ricerca di informazioni in un ambiente di griglia dove più organizzazioni virtuali condividono risorse in modo controllato e sicuro, appare chiaro come tali applicazioni debbano effettuare la ricerca delle risorse condivise in modo che le risposte siano immediate, e che il sistema sia efficiente e scalabile. L'espressività della ricerca risulta il punto cruciale delle applicazioni P2P. Nel caso del Grid Information Service, contesto in cui la presente tesi si è focalizzata, risulta fondamentale fornire un supporto all'esecuzione di **range query multiattributo**. Una range query multiattributo è una query che fa riferimento contemporaneamente a più attributi che rappresentano la risorsa da ricercare. Per ognuno di questi attributi, la query può ulteriormente limitare la ricerca ad un valore esatto oppure ad un range di valori appartenenti al dominio dell'attributo, utilizzando operatori come $\leq$ e $\geq$. Ad esempio una risorsa può essere rappresentata dai tre attributi CPU (che rappresenta la velocità della CPU di una macchina espressa in GHertz, ipotizzando uno spazio dell'attributo [0...5]), MEM (che rappresenta la

[1] Distributed Hash Table

quantità di memoria RAM di una macchina espressa in Mbyte, ipotizzando uno spazio dell'attributo [0...64]) e spazio di DISCO (che rappresenta la quantità di memoria non volatile di una macchina espressa in GByte, ipotizzando uno spazio dell'attributo [0...4096]). Una possibile range query multiattributo è la seguente:
$A = \{3 \leq CPU \leq 4, 1.5 \leq MEM \leq 2.0, 100 \leq DISCO \leq 300.0\}$

É possibile aumentare l'espressività introducendo ulteriori operatori come una wildcard per indicare che si intende considerare tutti i valori di un determinato attributo.

Emerge chiaramente che i modelli P2P fin quì non siano in grado di soddisfare tali requisiti e anzi risultino fortemente limitati sotto l'aspetto dell'espressività della ricerca delle risorse pubblicate.

Per superare questo problema, alcuni sistemi come Maan [7] definiscono delle multi-DHT per poter risolvere anche range query multiattributo: in questo sistema, ogni attributo è mappato su un ring Chord tramite l'uso di una funzione di hashing che preserva la località, cioè risorse con valori vicini vengono mappati su chiavi vicine. Successivamente le range query vengono risolte utilizzando il concetto di selettività di un attributo: in questo modo, scegliendo l'attributo più selettivo, si può ridurre il costo relativo alla risoluzione di una range query su DHT. Altri sistemi, come Squid [8], utilizzano una DHT mappando gli attributi tramite l'uso di una funzione di hashing che preserva la località che mappa uno spazio d-dimensionale in uno ad 1 dimensione, chiamata **Hillbert Space Filling Curve** [9]: in questo modo più attributi possono essere mappati su un'unica DHT.

Il difetto di questi ultimi sistemi è però il costo di inserimento delle informazioni nelle DHT e l'inefficienza nella risoluzione delle range query; infatti nonostante essi risolvano il problema della risoluzione delle range query multiattributo, le soluzioni proposte hanno una complessita superiore a $O(\log N)$, dove N rappresenta il numero di nodi che fanno parte della rete, tipica della risoluzione delle query esatte. Inoltre, in presenza di attributi dinamici, occorre reindicizzare la risorsa sulla rete.

Un'ulteriore classe di sistemi proposti utilizzano approcci alternativi alle DHT come quelli basati sulla tassellazione di Voronoi [10]. I diagrammi di Voronoi hanno la caratteristica di restringere l'insieme dei vicini di un peer, nel caso di 2 attributi, ad un insieme statisticamente costante. Inoltre, sfruttando le caratteristiche dello Small World [11, 12], definiscono un protocollo di routing greedy polilogaritmico. Le range query vengono risolte in modo naturale grazie al fatto che questi sistemi indicizzano oggetti invece di nodi fisici, rappresentando gli oggetti in base al valore dei loro attributi come punti su uno spazio multi-dimensionale, dove ogni dimensione rappresenta un attributo. Gli oggetti che sono match per una determinata range query sono raggiunti per mezzo del routing greedy. I diversi metodi basati su Voronoi si differenziano per come vengono raccolti i match della

range query: una volta raggiunto lo spazio descritto dalla range query SWAM-V [13] effettua un flooding per recuperare tutti i peer, mentre Voronet [14] non esplicita il metodo utilizzato.

Per superare l'inefficienza della tecnica basata sul flooding, sistemi successivi come quelli descritti in [15, 16, 17] costruiscono uno spanning tree distribuito per minimizzare i messaggi mandati sulla rete.

Analizziamo in particolare Voraque[15, 16], utilizzato come base per costruire l'architettura di Hierarchical Voraque. L'obiettivo fondamentale di VoRaQue è quello di migliorare il supporto alle range query sfruttando un algoritmo diverso dal flooding usato da SWAM-V per visitare tutti i peer che fanno parte dell'**Area di Interesse** della range query, cioè il poligono che rappresenta sul piano tutti i siti che rientrano nei limiti della query. L'esecuzione di una range query in Voraque si svolge in due fasi:

- nella prima fase viene utilizzato un algoritmo di routing greedy per raggiungere un peer che faccia parte dell'Area di Interesse, definito **peer radice**.
- nella seconda fase viene eseguito un algoritmo di **compass routing** a partire dal peer radice che permetta di costruire in modo distribuito uno spanning tree dell'Area di Interesse senza usare messaggi ridondanti. L'albero di copertura sfrutta i collegamenti esistenti tra i siti dell'Area di Interesse rappresentati dalla rappresentazione duale dei diagrammi di Voronoi, la Triangolazione di Delaunay.

I metodi basati su diagrammi di Voronoi tentano di limitare il costo di mantenimento abbassando il numero di vicini di un peer ad un numero costante, ma al costo di supportare un numero ridotto di attributi. Per poter supportare range query multiattributo, devono essere utilizzati diagrammi di Voronoi multidimensionali, cioè con un numero di dimensioni maggiore di 2, che però tendono ad avere un numero di vicini che **cresce esponenzialmente all'aumentare degli attributi**. Inoltre, in presenza di oggetti con attributi identici, si viene nuovamente a perdere il numero di vicini costante.

A loro favore c'è una miglior efficienza nel risolvere le range query rispetto alle DHT ed una migliore fault tollerance.

Questa tesi propone **Hierarchical Voraque**, un sistema che estende l'architettura di Voraque per poter gestire un numero di attributi maggiore di 2 in modo efficiente, senza incorrere nei problemi dei diagrammi di Voronoi multidimensionali. Nel caso di un numero di attributi maggiore di 2, è molto più probabile che alcuni oggetti abbiano in comune il valore di alcuni o tutti gli attributi con cui sono rappresentati: questo significa che in un sito del diagramma di Voronoi può essere stato pubblicato più di un oggetto, perdendo la corrispondenza sito-peer.

Questo si ripercuote sul numero di vicini di un peer, che in genere risultano non più limitati superiormente da una costante. Hierarchical Voraque si occupa anche di gestire questo problema senza aumentare la complessità della pubblicazione di nuovi oggetti o della risoluzione delle query.

Allo stato attuale della nostra conoscenza il nostro lavoro è il primo che propone la difinizione di una struttura gerarchica basata su tassellazione di Voronoi.

Illustriamo in sintesi la nostra architettura. Hierarchical Voraque è una rete gerarchica che divide l'insieme degli M attributi che descrivono gli oggetti, in coppie di attributi. La prima coppia di attributi viene mappata su una rete di Voronoi bidimensionale, che rappresenta il livello più alto dell'architettura. Nel caso in cui più peer pubblichino risorse con il valore degli attributi mappati al primo livello identici, ma con possibili valori diversi per gli altri attributi, si forma un **cluster**. In questo caso più peer vengono associati allo stesso sito p_i del diagramma di Voronoi del livello superiore. Per avere un limite superiore sul numero di vicini, per ogni cluster vengono eletti un certo numero di SuperPeer. Gli altri peer del cluster non hanno invece visibilità della rete di Voronoi di primo livello, ma hanno un riferimento alla lista dei SuperPeer. Si viene quindi a formare una Overlay Network tra gruppi di SuperPeer tramite la Triangolazione di Delaunay. Il metodo con cui viene eletto un SuperPeer è ininfluente per la nostra tesi. Il numero dei SuperPeer è variabile in base alla dimensione del cluster in modo da poter scalare con esso. Affinché questo numero sia il più possibile ridotto, ma sufficiente a mantenere un livello accettabile di servizio, il numero di SuperPeer è $\log(K)$, dove K rappresenta la dimensione del cluster. Il numero di SuperPeer variabile permette di non avere un singolo punto di fallimento per il cluster. Le altre coppie di attributi vengono mappati in successivi livelli della rete gerarchica: per ogni sito di un certa rete a livello L_i, se si forma un cluster sufficientemente grande, viene generata una nuova rete di Voronoi, utilizzando gli attributi di indice $[A_{2i}, A_{2i+1}]$, ed in questa rete di Voronoi sono mappati i peer del cluster. Un numero di peer pari al logaritmo dei peer del cluster sono eletti SuperPeer e servono da collegamento tra questa rete e quella di livello immediatamente superiore, e sono gli unici peer visibili alla rete superiore del cluster. Questo procedimento viene reiterato fino al livello $L_{Max} = M/2$ dove sono mappati gli ultimi due attributi. In sostanza, otteniamo una **struttura gerarchica multilivello** che prende la forma di un albero: ogni livello, che mappa una coppia di attributi diversa dagli altri livelli, è composto da una o più reti di Voronoi bidimensionali, al massimo una per ogni sito delle reti di livello superiore associato ad un cluster sufficientemente grande. Utilizzando reti bidimensionali come nodi dell'albero è possibile sfruttare le caratteristiche e gli algoritmi di Voraque localmente ai vari livelli, in particolare il routing greedy ed il routing compass. Essi però sono stati modificati per supportare le nuove

procedure di pubblicazione di un oggetto e di risoluzione di una range query multiattributo.

Questo tipo di architettura ha mostrato nei test e nel calcolo delle complessità teoriche, un buon livello di scalabilità. Infatti, grazie al fatto di aver limitato i SuperPeer ad un numero logaritmico, il numero dei vicini di un qualsiasi peer inserito nella rete cresce molto lentamente. Questo ha limitato la complessità dell'inserzione a $\log N$ con N numero dei peer dell'intera rete, ma soprattutto questo risultato è **indipendente** dal numero dei livelli e quindi degli attributi. Infine, l'algoritmo di risoluzione delle query utilizzato ha una complessità che è dipendente dal numero degli attributi, ma mitigata molto dal fatto che sfrutta la loro selettività. Questo deriva dal fatto che visita Aree di Interesse che contengono siti che sono almeno soluzione alla query per tutti gli attributi mappati nelle reti dei livelli superiori grazie al funzionamento di tipo top-down. Risultato notevole è che l'ordine in cui sono mappati gli attributi sulla rete non incide sul costo totale della query.

Possiamo concludere che Hierarchical Voraque può essere utilizzato efficacemente come Grid Information Service per i seguenti motivi:

1. la sua migliorata resistenza ai fallimenti rispetto a Voraque, ottenuta grazie all'uso di SuperPeer multipli perfettamente intercambiabili
2. il supporto delle range query multiattributo efficiente e indipendente dall'ordine in cui gli attributi sono mappati sui vari livelli della rete.
3. il bilanciamento del carico introdotto dall'uso dei SuperPeer multipli
4. il supporto ad un algoritmo di inserzione a basso costo, paragonabile a quello di una rete di Voronoi bidimensionale, ma indipendente dal numero dei livelli

1.1 Strutturazione dei capitoli della tesi

Nel capitolo 2 vengono analizzati in maggior dettaglio i sistemi P2P esistenti in letteratura che si sono presentati nel tempo, valutandone i punti di forza e di debolezza.

Nel capitolo 3 viene illustrata in dettaglio l'architettura di Hierarchical Voraque, spiegando con esempi i vari algoritmi utilizzati e le principali strutture dati. A fine capitolo viene aggiunto il calcolo delle complessità degli algoritmi.

Nel capitolo 4 vengono specificati gli algoritmi introdotti in Hierarchical Voraque. La descrizione viene fatta ad alto livello, specificando gli algoritmi tramite **pseudocodice** e descrivendo i campi dei messaggi utilizzati.

Nel capitolo 5 vengono illustrati i test effettuati su una implementazione di Hierarchical Voraque fatta con l'uso di un software di simulazione di reti: PeerSim[36]. Nel capitolo 6 vengono tratte le conclusioni sul lavoro svolto e introdotti alcuni possibili sviluppi futuri.

Capitolo 2

Stato dell'arte dei sistemi per la pubblicazione e ricerca di risorse

In questo capitolo viene data una panoramica dei sistemi per la pubblicazione e ricerca di risorse presenti in letteratura, classificandoli secondo la loro Overlay Network: questi possono essere Non strutturati, Strutturati ed una terza categoria di sistemi Ibridi, che cercano di mantenere i benefici di entrambi i sistemi precedenti rilassando alcune restrizioni dei sistemi strutturati, come ad esempio il protocollo Kademlia [18], Pastry [6] o [19]. L'**Overlay Network** non è altro che una rete virtuale stabilita tra i peer mediante connessioni TCP per lo scambio di informazioni di controllo. Questa è completamente indipendente dalla effettiva conformazione della rete fisica e può essere come ho detto sopra più o meno strutturata. I sistemi ibridi, che possono essere a loro volta centralizzati o no, vengono classificati tra quelli strutturati o non strutturati in base a quale dei due si avvicinano maggiormente come architettura. Nonostante la maggior parte dei sistemi descritti sia in grado di memorizzare un certo numero **m** di attributi, a cui viene associato un valore in un determinato dominio, e sia in grado di reperire i nodi che possiedono le risorse descritte dai valori di questi attributi, essi si differenziano in base al supporto che danno alla risoluzione delle **query**. Vi sono sistemi a cui possono essere sottomesse soltanto query che specificano una condizione su un unico attributo chiave, detti sistemi con supporto a query **monoattributo**, e sistemi in cui possono essere specificati condizioni di ricerca su più attributi, detti sistemi con supporto a query **multiattributo**. A sua volta, ci sono sistemi, di entrambi i tipi, che gestiscono solo query **esatte**, cioè che specificano un solo valore per ogni attributo ricercato, o **range** query, dove per ogni attributo è possibile specificare un insieme di valori possibili (in questo caso i vari sistemi definiscono diversi insiemi di operatori possibili per specificare i range). Infine, a seconda della frequenza con cui i nodi della rete pubblicano ed aggiornano i valori degli attributi delle risorse, questi possono essere divisi in **statici**, come per esempio

la cpu di un nodo, o **dinamici**, come ad esempio il carico di lavoro istantaneo di un sistema di elaborazione. Ogni sistema può avere un diverso modo di supportare i due insiemi di attributi in modo da garantire un certo livello di scalabilità. Nei prossimi due paragrafi riporto brevemente una serie di sistemi, descrivendone l'architettura base e il modo in cui gestiscono gli attributi, le query e l'Overlay Network, inserendo alla fine di ogni paragrafo i pregi ed i difetti della classe di sistemi descritta.

2.1 Sistemi P2P con Overlay Network non strutturata

Questi sistemi hanno la caratteristica, come ho detto nel capitolo precedente, di non avere una struttura di rete logica predefinita: alcuni di questi, come LDAP[20], Napster[2] e BitTorrent[21], hanno uno o più server che servono le richieste dei client, quindi non sono dei veri e propri sistemi P2P per PCComputing o GIS[1]. Il primo dei due è stato introdotto perché, nonostante non sia una architettura P2P, è utilizzato come GIS o come IS per sistemi client/server aziendali, mentre Napster, nato per il file sharing, anche se utilizza un index server centrale, scambia i file tra i peer in modo decentralizzato, utilizzando tecniche per superare i firewall. Per finire, BitTorrent, un sistema di file sharing recente, non usa server centralizzati, ma elegge alcuni nodi a **tracker** e divide i file in pezzi, assegnando ad ogni pezzo una checksum ed un identificatore unico per permettere di scaricare simultaneamente da più peer e poter infine ricostruire il file originale. Le informazioni su quante parti, quanto grandi, il nome e la grandezza del file, i nodi che ne possiedono una copia completa ed altre informazioni sul file stesso sono contenute in un file specifico creato da colui che ha pubblicato per la prima volta il file, definito **seeder**, con estensione .torrent. Non esiste una rete P2P per reperire questi file, ma sono pubblicati su siti Internet appositi: il sistema è classificato come P2P perché lo scambio dei pezzi è decentralizzato, appoggiandosi soltanto ai tracker per sapere chi possiede i vari pezzi di un certo file.

Altri sistemi, come Gnutella 0.4 [3], sono completamente decentralizzati, altri ancora come l'evoluzione 0.6 di Gnutella [22] e Kazaa [23] sono invece basati su un approccio a **SuperPeer**, nodi particolari che sono automaticamente eletti per servire una piccola parte dei nodi della rete in modo diretto, prendendosi carico delle risorse pubblicate da questi ultimi, mantenendo un indice dei nodi collegati in un determinato istante e inserendo in una cache le risorse da questi pubblicate, e servono le query immesse o dirette ai nodi da loro gestiti, in modo da limitare il numero di peer presenti sulla rete.

[1]Grid Information Service

Mancando una struttura di rete predefinita, le risorse rimangono localizzate sui nodi che le mettono a disposizione di chi ne fa richiesta. Il problema principale di questi sistemi sta nei metodi di routing e di ricerca delle informazioni nella rete, spesso effettuati brutalmente tramite algoritmi di flooding, nei sistemi completamente decentralizzati. Per ovviare parzialmente a questo problema, alcuni sistemi più recenti introducono dei meccanismi basati su **RoutingIndex** come ad esempio [19], implementato tipicamente tramite strutture ad albero. Inoltre, molti di questi sistemi, in particolare quelli decentralizzati puri, non permettono sempre di restituire l'insieme completo delle risorse che rispondono ad una query, per limitare l'inondazione della rete. Dall'altra parte però, nella maggior parte dei casi, l'inserzione e la rimozione di nodi è un'operazione poco costosa in questi sistemi.

2.1.1 Napster

Napster[2] venne introdotto nel 1999 come sistema di file sharing, per permettere agli utenti di poter scambiarsi file musicali basati principalmente sulla compressione MP3[2]. Esso rappresenta il primo sistema P2P mai realizzato, con architettura di tipo parzialmente centralizzata. Napster è caratterizzato da un server centrale a cui si collegano, secondo una conformazione di rete a stella, tutti i peer che entrano a far parte del sistema, che lo utilizzano come broker, cioè come indice di tutti i file presenti in ogni peer connesso. Le connessioni tra i peer avvengono solamente **on demand**, cioè solo al momento in cui due peer si scambiano effettivamente il contenuto di un file richiesto: in questo modo il server è esonerato dal memorizzare le risorse, che sono lasciate al controllo dei peer. Nella figura 2.1 possiamo vedere uno schema di questa architettura.

In Napster non viene indicato come deve essere costruita la struttura dati che costituisce l'indice dei file, quindi possono essere utilizzati anche database tradizionali. La prima operazione che esegue un peer è il login e l'autenticazione sul server centrale: quando il peer si connette per la prima volta, il messaggio al server contiene anche un nickname, una password per l'autenticazione e la porta a cui il server invia i messaggi di coordinamento, con la richiesta di verificare se il nickname è unico. Il server risponde sulla porta comunicata nel messaggio se il nickname è già usato da un altro peer oppure no e salva la password. Successivamente il peer invia il vero e proprio messaggio di **login**, specificando il proprio nick (unico), il suo indirizzo IP e la porta dalla quale accetterà connessioni dagli altri peer e la versione del client utilizzato, a cui il server risponde con un messaggio di **login ack** o di rifiuto della connessione. Come ultima operazione di login, il peer invia una serie di messaggi di tipo **notification of shared file**, uno

[2]MPEG 1 layer 3, formato audio molto diffuso su Internet, basato su compressione a perdita di dettaglio con modello psicoacustico

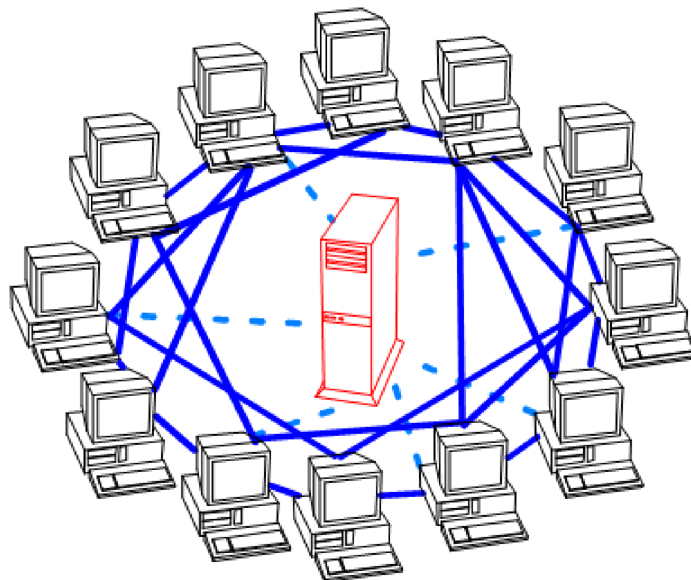


Figura 2.1: Struttura di Napster a server unico

per ogni file che vuole pubblicare e condividere, in cui specifica il nome del file e soprattutto la stringa ottenuta dall'esecuzione dell'algoritmo MD5[3] sul nome del file, che viene utilizzato per garantire l'unicità di un file MP3, così da poter tenere traccia dei file duplicati e facilitarne il reperimento; in coda ad ogni messaggio di notifica vengono inseriti altri dati sul file, tra cui la dimensione, il bitrate e la frequenza. Altre operazioni che il peer può fare sono la cancellazione di un file dall'indice del server, il logout ed infine effettuare query per ricercare file, utilizzando messaggi di tipo **search**. La ricerca di un file sul server può essere effettuata specificando il nome del file da ricercare e gli altri attributi sul file memorizzati sul server, il numero massimo di risposte che si vuole ricevere e le caratteristiche della connessione con il peer che possiede il file ricercato. Il sistema è anche in grado di utilizzare una serie di operatori per meglio specificare le condizioni di ricerca, come per esempio solo una parte del nome, un minimo od un massimo bitrate o frequenza. In conclusione il sistema è in grado di supportare range query multiattributo, anche se limitato soltanto alla condivisione di file musicali, e quindi non general purpose.

[3]Message Digest Algorithm 5: è un algoritmo che, presa in input una stringa di lunghezza arbitraria, restituisce una stringa di 128bit, in modo tale da rendere improbabile che due stringhe diverse generino stringhe MD5 uguali. Inoltre rende complesso risalire alla stringa di origine a partire dalla stringa MD5 generata

Il server risponde con uno o più messaggi di tipo **search response**, se nel proprio database è contenuto almeno un file che soddisfa il messaggio di search. Ogni messaggio di risposta contiene il nome completo del file, il nickname, l'indirizzo IP e la porta dei peer che lo possiedono, il checksum MD5 del file, che viene utilizzato per verificare l'integrità del file trasferito tra i peer. A questo punto avviene il trasferimento del file tra il peer richiedente P1 ed uno dei peer P2 contenuto in uno del messaggi di response ricevuti dal server. Il trasferimento, effettuato tramite protocollo HTTP con un messaggio di tipo GET, avviene direttamente tra i peer, ma con il coordinamento del server centrale (che chiamerò S), in modo tale che non possano ricevere connessioni che non sono desiderate e che non siano prima certificate dal server. Ecco le operazioni che il sistema esegue per effettuare il trasferimento di un file tra P1 e P2 usando S come coordinatore:

1. P1 notifica a S che intende scaricare un file da P2
2. S notifica a P2 la richiesta di P1
3. P2 notifica ad S la propria disponibilità ad accettare la connessione da P1 e indica la porta su cui P1 si deve connettere per trasferire il file
4. S notifica a P1 la risposta
5. P1 apre una connessione sulla porta di P2 ricevuta tramite S e P2 trasferisce il file su questa connessione a P1

Possiamo vedere da questo protocollo che il sistema centralizzato introduce un livello di sicurezza: un peer non accetta trasferimenti se prima non ha ricevuto una richiesta in tal senso dal server S per conto di un altro peer P. Il sistema risolve anche il problema del trasferimento di file nel caso in cui alcuni peer si trovino a monte di firewall: in questo caso bisogna che almeno uno dei due peer accetti connessioni dall'esterno. Supponendo sempre il trasferimento di file descritto sopra, nel caso in cui P2 sia a monte di un firewall, esso notifica a S che non può accettare connessioni in ingresso, ma che è disponibile per il trasferimento. S notifica a P1 la risposta di P2 indicando che quest'ultimo non può accettare connessioni: P1 allora manda a S un ack indicando che sarà lui ad accettare la connessione in ingresso indicando la porta su cui riceverla. S manda a P2 la notifica che P1 accetta la connessione, inviandogli la porta, P2 effettua il collegamento con P1 ed inizia il trasferimento del file. Nella figura 2.2 possiamo vedere un esempio di interazione tra il server e due peer, relativo a tutte le operazioni che questi possono fare.

In una versione successiva, per migliorare la scalabilità, è stato sostituito il server centrale con un **cluster di server**. Il cluster si trova in un'unica locazione geografica, ed ha un unico punto di accesso al sistema dei server, in modo che i peer si colleghino attraverso un unico nome simbolico (`server.napster.com`,porta

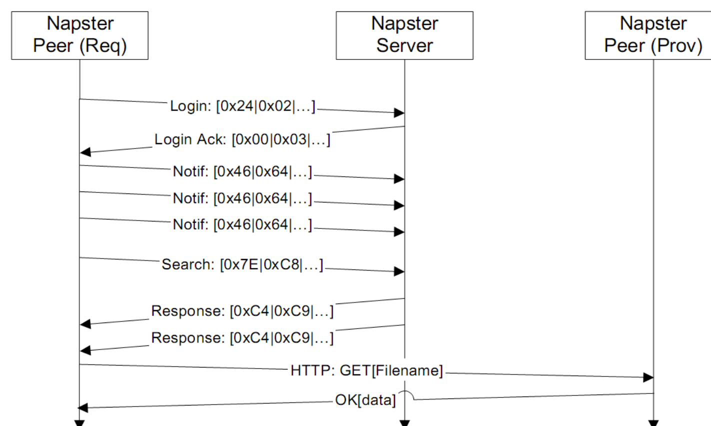


Figura 2.2: Esempio di interazione tra peer e server

8875). Per bilanciare il carico, i server vengono divisi in due gruppi e viene creata una gerarchia a due livelli: il primo gruppo è costituito dai **lookup servers**, i quali ricevono le richieste dai peer e le smistano al secondo gruppo costituito dai **brokers** con un algoritmo che bilanci il carico sui brokers. I brokers contengono l'indice che contiene i metadati sui file della rete, distribuito fra i vari brokers: essi sono poi collegati fra loro per poter accedere all'intero indice e sono collegati a tutti i lookup servers. Quando un peer vuole effettuare il login a Napster, manda il messaggio di login al nome simbolico che ho indicato sopra. Il sistema di DNS locale dei lookup servers può fungere da bilanciatore del carico tra questi, restituendo al peer l'elenco degli indirizzi dei lookup server, ruotando ad ogni nuova richiesta l'ordine degli indirizzi: basandosi sul fatto che il peer sceglie di solito il primo indirizzo che riceve, si ottiene così il bilanciamento ricercato. A questo punto il peer si connette al broker, il quale avrà in carico le stesse funzioni che aveva prima il server unico e riceve dal peer i messaggi di notifica dei file da questo condivisi.

Nella figura 2.3 vediamo la struttura di questa evoluzione di Napster semplificata.

Come in LDAP, Napster ha come svantaggio il fatto che i server costituiscono un punto di fallimento, limitano la scalabilità dell'intero sistema al crescere del numero dei nodi (il numero dei server è infatti indipendente dal numero dei nodi) e costituiscono un collo di bottiglia, come tutti i sistemi centralizzati, per le query, aumentandone il tempo di risposta. Il trasferimento diretto dei file tra i peer permette ai servers di ridurre il proprio carico di lavoro, inoltre i peer non devono scambiare messaggi per il mantenimento della rete e non devono mantenere strutture dati per supportarlo, minimizzando il numero di messaggi presenti sulla rete stessa. Il metodo riesce a risolvere, come abbiamo visto, range query multiattri-

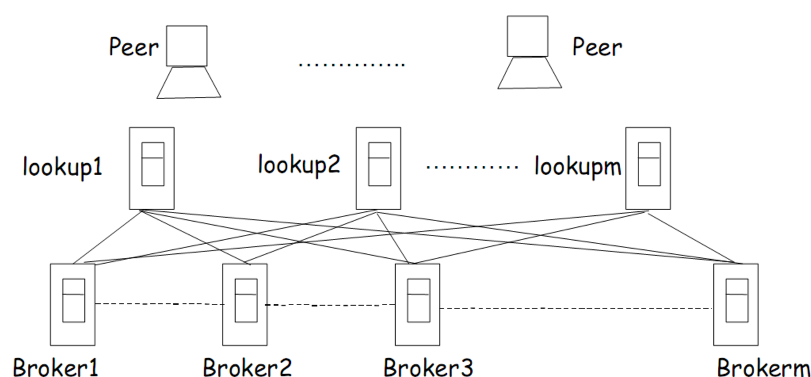


Figura 2.3: Struttura di Napster con cluster di server

buto, ma limitate soltanto al proprio ambito di utilizzo (sharing di file musicali), e riesce, diversamente dai sistemi decentralizzati puri, a localizzare tutte le risorse rispondenti ad una query se sono presenti sulla rete.

2.1.2 Gnutella 0.4

Gnutella 0.4[3] è stata una delle prime proposte di P2P completamente decentralizzato finalizzato al file sharing tra utenti. É stato proposto nel 2000 da Frankel e Pepper ed è basato appunto sulla mancanza di un'unità centralizzata nella Overlay Network che funga da indice delle risorse o da coordinatore, ma i nodi si connettono direttamente tra loro utilizzando un'applicazione che funziona contemporaneamente sia da client che da server, la quale si occupa di propagare o rispondere alle query e di acquisire conoscenza sulla rete: i peer in Gnutella 0.4 vengono chiamati **servent**. In questa architettura, i servent non notificano i file che intendono condividere, quindi non esiste un modo di sapere quale peer contiene un file ricercato e quale è la struttura attuale della rete per poter costruire un algoritmo di routing che si adatti alla rete virtuale.

A causa di questa mancanza di conoscenza, i peer utilizzano un protocollo di tipo **enhanced flooding** per risolvere le query ed esplorare la rete per scoprire l'esistenza di nuovi peer.

Per connettersi alla rete, un servent inizialmente deve conoscere almeno un altro peer già connesso: essendo il sistema privo di punti di centralizzazione con un nome specifico, un nuovo nodo non può ricavare questa informazione facilmente. Per risolvere questo problema, vengono utilizzati dei WebServer appositi chiamati **bootstrap servers** che possiedono una lista, aggiornata abbastanza frequentemente, di un certo numero di nodi già attivi su Gnutella.

A questo punto il servent si collega tramite TCP ad un certo numero di peer presenti nella propria peer cache o nella lista ricevuta da un WebServer: il messaggio contiene la stringa GNUTELLA CONNECT, con la versione del protocollo, alla quale l'altro peer può rispondere con un messaggio contenente la stringa GNUTELLA OK oppure rifiutarla. Il rifiuto dipende da una politica di gestione delle connessioni che è locale ad ogni servent, in quanto dipende dalla banda di entrata/uscita che caratterizza il servent: un servent quindi decide quante connessioni massime creare e ne riserva una parte per le connessioni in ingresso da altri peer (superate le quali risponde con messaggi di rifiuto), le altre costituiscono il numero massimo di peer a cui si connette durante la fase di bootstrap. Che un servent rifiuti o no una connessione, nel messaggio di risposta inserisce un **X-try Header**, che contiene un insieme di coppie (Indirizzo IP, Porta) che individuano una lista di servent Gnutella noti al servent, contenuti nella sua peer cache.

Nelle fasi successive di esplorazione della rete e di invio di query, i messaggi scambiati sulla rete possono essere di tipo **keep-alive**, usati per segnalare la presenza di un peer sulla rete e sono costituiti da messaggi di tipo **ping** a cui viene risposto con messaggi di tipo **pong**. Oppure possono essere di richiesta di file condivisi, costituiti dai messaggi di tipo **query**, a cui viene risposto con messaggi di tipo **queryHit**.

Una volta che un servent si è collegato ad un certo insieme di vicini, inizia la fase di esplorazione della rete:

- il nodo invia a tutti i propri vicini un messaggio di tipo ping, con un determinato GUID e con un valore di TTL[4] scelto in base al livello di profondità a cui si desidera esplorare la rete. Inoltre memorizza in una tabella la corrispondenza tra il GUID e il peer a cui è stato inviato per poi riconoscere la risposta.

- Il nodo che riceve un messaggio di ping esegue il seguente algoritmo:

  1. Decrementa il TTL del messaggio: se TTL=0 il messaggio viene scartato per evitare di inondare la rete.
  2. Se il peer aveva già ricevuto lo stesso messaggio (ogni peer mantiene una tabella dinamica dove memorizza il GUID, il tipo e il canale di tutti i messaggi ricevuti o spediti), confrontando il GUID con il contenuto della tabella suddetta, lo scarta per evitare cicli.
  3. Altrimenti il messaggio viene spedito a tutti i vicini, escluso il peer da cui si è ricevuto il messaggio. Incrementa il campo HOPS di 1
  4. Genera un messaggio di tipo pong di risposta con il GUID identico al messaggio di tipo ping ricevuto, TTL identico e il campo Hops=0 e

[4] Time to Live

lo invia al servent da cui ha ricevuto il messaggio di ping (estraibile dalla tabella suddetta sfruttando il GUID unico): questo tipo di metodo per generare e inoltrare i messaggi di risposta si definisce **backward routing**, in quanto il messaggio di risposta segue lo stesso percorso a ritroso del messaggio originale.

Quando invece un peer riceve un messaggio di tipo Pong, controlla se il ping originario non era stato inoltrato da lui (controllando il GUID del messaggio e la propria tabella dei messaggi originati), altrimenti lo invia al servent da cui aveva ricevuto il messaggio di ping corrispondente, eseguendo il **backward routing**, decrementa il campo TTL e incrementa il campo HOPS di 1. Inoltre, il peer sfrutta i messaggi di pong che riceve, indipendentemente dal servent a cui il messaggio è diretto, utilizzandoli come mezzi per sapere quanti altri peer sono ancora presenti sulla rete (keep-alive). Ogni servent, a intervalli regolari, reinvia un messaggio di ping per mantenere la propria visione della rete.

Vediamo nella figura 2.4 un esempio di esplorazione della rete da parte di un peer con indirizzo 2000 connesso ad un peer con indirizzo 4000.

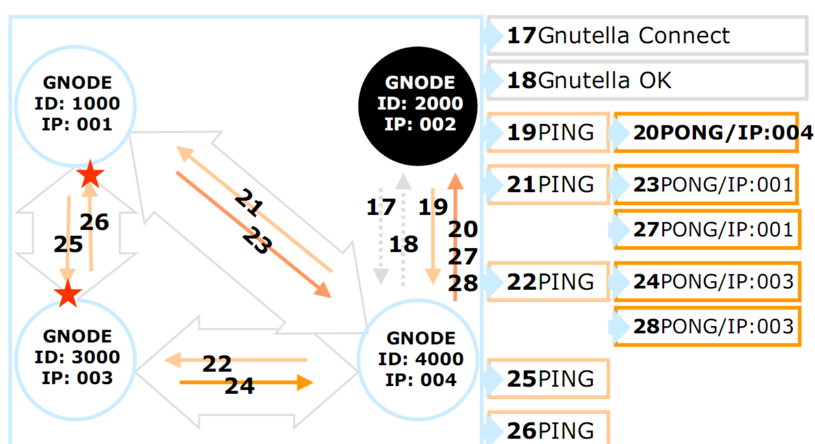


Figura 2.4: Esempio di esplorazione della rete: le frecce tratteggiate rappresentano i messaggi di bootstrap

Un esempio più completo lo possiamo vedere invece nella figura 2.5.

Per quanto riguarda la fase di produzione e sottomissione di query al sistema, viene utilizzato lo stesso tipo di protocollo per i messaggi di tipo Query ed i relativi messaggi di tipo QueryHit.

Quando il peer originale che ha sottomesso la query riceve almeno un messaggio di QueryHit relativo alla propria richiesta (identificato tramite il GUID), può aprire direttamente con il servent che ha generato il QueryHit una connessione diretta di tipo HTTP per il trasferimento del file.

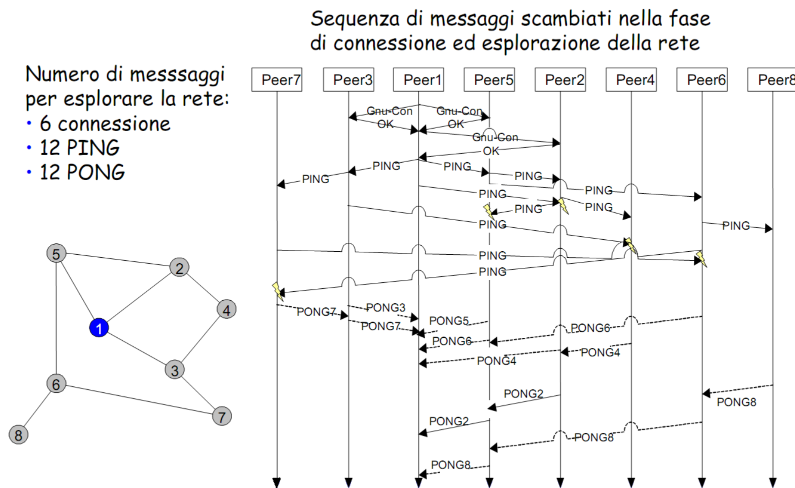


Figura 2.5: Sequenza dei messaggi scambiati in una fase di esplorazione della rete in maggiore dettaglio

Infine, quando un peer vuole abbandonare la rete, non deve fare altro che salvare la propria peer cache per essere usata in sessioni successive, non inoltrare nessun altro messaggio di tipo Ping o Query e chiudere le connessioni TCP con i propri vicini. Se invece il nodo cade improvvisamente, i suoi vicini lo cancellano dopo un po' di tempo quando non ricevono più messaggi di tipo Pong da lui.

Rispetto ai due metodi centralizzati descritti nei due paragrafi precedenti, in Gnutella 0.4 la scalabilità è migliorata in quanto non esiste un'entità centralizzata che contiene l'intero indice delle risorse come il server di Napster e che rappresenta un singolo punto di fallimento. Al contrario, Gnutella è fault tolerant in quanto l'uscita di un nodo non compromette il funzionamento della rete, né la disponibilità delle informazioni, perché ogni nodo è responsabile solo delle proprie risorse. L'entrata e l'uscita dei nodi dalla rete è un altro punto di forza, in quanto i singoli nodi devono eseguire un numero molto limitato di operazioni, che rappresenta un beneficio in presenza di un elevato tasso di entrata/uscita (detto **churn rate**). Anche le strutture dati che un nodo deve mantenere sulla conoscenza della rete sono limitate (ogni nodo ha infatti un numero limitato e costante di vicini, che può cambiare nel tempo soltanto in base alla propria politica relativa alle connessioni).

Il vero problema di questi sistemi è la loro stessa natura di essere non strutturati: questo comporta l'utilizzo di un algoritmo di ricerca e di keep-alive basato sul flooding, che nel caso peggiore comporta, per ogni singolo messaggio inviato (di tipo ping o query) la visita di un numero di nodi che è $O(N)$, con N numero di peer della rete. Il flooding inoltre genera un volume di traffico inteso come numero di messaggi presenti in rete che non è scalabile, anzi aumenta con il numero dei nodi: infatti esso crea un volume di traffico che spesso non è necessario perché

raggiunge nodi che per esempio non sono risposta ad una query. Per limitare questo fenomeno, Gnutella ha introdotto il meccanismo basato su TTL, ma esso effettivamente suddivide la rete in tante sottoreti, costituite dai nodi che un peer riesce a raggiungere in base ad un certo valore del proprio TTL. In questo modo, non si riesce a raggiungere tutti i nodi, e quindi non si riesce a localizzare tutte le possibili risorse rispondenti ad una query.

Per limitare il numero di messaggi presenti sulla rete, successivamente sono stati implementati sistemi per il controllo del flusso come per esempio il meccanismo definito Dynamic Querying[24] , dove un peer manda una query soltanto ad un sottoinsieme dei suoi vicini con un limitato TTL. Se non riceve abbastanza risposte, procede ad inviare un ulteriore messaggio ad un altro sottoinsieme dei suoi vicini incrementando il TTL, fino a quando non ci sono più vicini. Altri metodi utilizzati per ridurre il traffico di rete sono il random walk, il multiple random walk, il multiple parallel random walk[25] e metodi ibridi che combinano fooding e random walk[26].

In sostanza, le reti decentralizzate pure e non strutturate come Gnutella sono indicate laddove vi è un alto dinamismo nell'entrata e uscita dei nodi, la rete non è troppo grande e l'operazione più comune è la ricerca per parole chiave. Per abbattere in parte il problema della scalabilità dei messaggi sulla rete, in una versione successiva del protocollo Gnutella, viene introdotto il concetto di **SuperPeer**.

2.1.3 Gnutella 0.6

Gnutella 0.6[22] è nato come evoluzione di [3] per permettere una maggiore scalabilità del protocollo originale. Esso si basa su una gerarchia dinamica a più livelli dei peer; una parte di questi, definiti **SuperPeer**, è assegnato il compito di servire una piccola sottoparte dei peer della rete indicizzando e facendo il caching dei file contenuti in essa, gli altri, definiti **LeafNode**, sono collegati direttamente ad uno o più (solitamente soltanto uno) dei SuperPeer e scambiano informazioni soltanto con essi, mantenendo un numero limitato di connessioni. I SuperPeer vengono eletti dinamicamente in base alla capacità di banda di rete e di elaborazione.

In Gnutella 0.6 i SuperPeer sono gli unici nodi della rete che si collegano tra loro seguendo i meccanismi del protocollo di Gnutella 0.4: essendo solo una parte dei nodi dell'intera rete, la quantità di messaggi generati dal flooding viene nettamente ridotta, ma il carico dei SuperPeer è ovviamente maggiore rispetto al carico richiesto ai peer nella rete completamente decentralizzata. La topologia di rete che si viene a creare viene definita di tipo ibrido, parzialmente decentralizzato: i SuperPeer infatti fungono da server tipo Napster per i propri LeafNode, mentre fra loro generano una rete identica a quella di [3].

Nel resto della descrizione, presento solo i caratteri peculiari di questa architettura rispetto a [3]: l'elezione dei SuperPeer, l'entrata e l'uscita di un peer e i messaggi scambiati tra LeafNode e SuperPer.

Innanzitutto passiamo a vedere come avviene il bootstrap di un nodo in Gnutella 0.6. Prima di tutto, un nodo decide, in base a parametri a lui locali e autonomamente, se ha capacità da SuperPeer. A questo punto, con meccanismi simili a quelli di Gnutella 0.4 (attraverso liste reperite da WebServer appositi o da una propria PeerCache), il nodo sceglie un peer S al quale vuole connettersi.

Il peer manda ad S un messaggio di bootstrap, contenente i seguenti campi:

1. un campo contenente la stringa GNUTELLA CONNECT/0.6

2. un campo contenente il tipo e la versione del client utilizzato

3. un particolare campo booleano che indica se il peer ha le capacità per diventare SuperPeer oppure no. In caso affermativo, esso si identifica come SuperPeer Capable (può essere anche un SuperPeer effettivo).

4. il proprio indirizzo IP e la porta da cui ricevere connessioni.

S può essere un LeafNode oppure un SuperPeer. Vediamo in dettaglio come risponde a seconda di uno dei due casi.

Se S è un SuperPeer e il nodo N che effettua il bootstrap non è SuperPeer Capable, S manda a N un messaggio con i seguenti campi:

1. un campo contenente la stringa GNUTELLA/0.6 200 OK per dire che è un nodo di una rete Gnutella 0.6. Il SuperPeer può accettare o rifiutare la connessione a seconda del numero di LeafNode che gestisce e di connessioni che ha verso altri SuperPeer.

2. un campo contenente il tipo e la versione del client utilizzato

3. un campo denominato **X-Ultrapeer**, di tipo booleano, che indica se il nodo è un SuperPeer oppure no.

4. un campo, denominato **X-Ultrapeer-Needed**, di tipo booleano, che indica se c'è bisogno di un nuovo SuperPeer oppure no. Questo campo viene settato in base a diversi parametri valutativi che il superpeer S prende in considerazione: in particolare, se S ha un numero troppo elevato di LeafNode questo campo viene settato a true.

5. un campo, denominato **X-try-SuperPeers**, che contiene una lista di possibili SuperPeer che il nodo N può contattare per collegarsi, se viene rifiutata la connessione da questo SuperPeer. Questo campo viene indicato anche

se la connessione è accettata, in quanto verrà utilizzato da N se il nodo S cade o perde lo status di SuperPeer, oppure se N vuole collegarsi anche ad un altro SuperPeer. I dati in esso contenuti sono prelevati da una cache di S ottenuta tramite i messaggi di Pong ricevuti dalla Overlay Network dei SuperPeer.

Se il nodo N viene accettato, esso diventa un LeafNode di S e rifiuta qualunque tipo di connessione di tipo Gnutella 0.4 ed invia a S (e agli altri eventuali SuperPeer a cui è connesso) una QRP [5], il cui contenuto verrà descritto più avanti. A questo punto termina il bootstrap e N fa parte della rete Gnutella 0.6, oppure può continuare a provare a connettersi ad altri peer.

Se S è un LeafNode, esso ha almeno un SuperPeer S2 a cui è connesso. In questo caso S rifiuta la connessione ed il campo X-Ultrapeer è settato a false: il nodo S inserisce nel campo X-try-SuperPeers l'indirizzo IP e la porta del SuperPeer S2 e degli altri SuperPeer a cui è connesso. A questo punto N prova a connettersi a S2 secondo il protocollo indicato nel caso precedente.

Se S è un LeafNode senza SuperPeer, significa che fa parte di una rete Gnutella 0.4 e i due nodi si collegano secondo il vecchio protocollo.

Se S è un SuperPeer e N è SuperPeer Capable possono succedere due casi:

- se S ha settatto X-Ultrapeer-Needed a true, N rimane un SuperPeer: se per un determinato periodo di tempo non riceve connessioni da nuovi LeafNode, N riproverà a connettersi nuovamente ad un altro SuperPeer. Se S ha settato X-Ultrapeer-Needed a false, N risponderà ad S con un messaggio di ack in cui setta X-Ultrapeer a false e diventa un LeafNode di S. Se il nodo N possiede LeafNode, , questa connessione viene considerata come una normale connessione TCP tra SuperPeer nella loro Overlay Network.

In questo ultimo caso, ad intervalli regolari, un LeafNode che sia SuperPeer Capable si ripropone al proprio SuperPeer nuovamente seguendo il protocollo appena descritto, per verificare se non ci sia bisogno di un nuovo SuperPeer.

Terminata la fase di bootstrap, se il nuovo nodo è una foglia, invia al proprio SuperPeer una QRP: essa contiene le informazioni sulle risorse che il nodo vuole condividere con il resto della rete, e verrà utilizzato dal SuperPeer per costruire una **tabella QRP**. Gnutella 0.6 usa una serie di strategie per contenere la grandezza di questa tabella, fra cui tecniche di hashing (basate sulle parole chiave comunicate tramite QRP dai nodi foglia) e di compressione. La tabella QRP è usata dal SuperPeer per effettuare il routing delle query che riceve dalla rete dei SuperPeer per inoltrarle ai LeafNode che hanno risorse che soddisfano la query.

Vediamo che cosa succede quando un nodo N genera un messaggio di tipo Query:

[5]Query Routing Table

1. Se il nodo N è una foglia, invia il messaggio al proprio SuperPeer, senza propagarlo a nessun altro peer.

2. Quando un SuperPeer S genera o riceve un messaggio di query da un LeafNode, controlla nella propria tabella QRP se il file richiesto è offerto da uno dei propri LeafNode: se il controllo è positivo S invia ai nodi individuati nella QRP il messaggio di query.

3. Il Superpeer S invia la query ai SuperPeer a cui esso è connesso nella overlay network per individuare altri hosts che condividono il file richiesto, utilizzando il normale protocollo Gnutella 0.4. Inoltre controlla se esso non ha file che soddisfano i criteri della query, altrimenti genera un QueryHits e lo invia a N.

4. Quando un SuperPeer S riceve un messaggio di query da un altro SuperPeer, prima effettua i normali controlli secondo il protocollo di Gnutella 0.4 (per esempio controlla se il TTL è uguale a 0, se il messaggio non è un duplicato, ...) poi esegue i punti 2 e 3 sopra descritti.

Per quanto riguarda i messaggi di QueryHit, i LeafNode li inviano ai propri SuperPeer, che usano il Backward Routing[3] per farli giungere al SuperPeer che ha inoltrato il messaggio di Query (per proprio conto, o per un suo LeafNode).

I messaggi di KeepAlive (Ping e Pong), vengono scambiati solo tra i SuperPeer secondo il protocollo di Gnutella 0.4. Essi non vengono mai propagati ai LeafNode, per evitare che questi siano invasi da messaggi inutili per loro, che non fanno parte della rete dei SuperPeer. Se un LeafNode manda un Ping a un SuperPeer, questo risponde con un messaggio di Pong, senza propagare il Ping, inserendo la lista di SuperPeer che conosce, per permettere al LeafNode di collegarsi ad altri SuperPeer in caso lui abbandoni la rete o sia soggetto ad un crash.

Un protocollo simile è quello della rete Kazaa[23], che come Gnutella 0.6 utilizza SuperNodes e OrdinaryNodes (rispettivamente i SuperPeer e LeafNode di Gnutella) ed è quindi una rete ibrida: i principali punti di forza rispetto a Gnutella 0.6 sono la possibilità di cambiare le connessioni sulla Overlay Network dinamicamente verso altri SuperNodes, in modo da coprire una maggior quantità di risorse, e mantenere un rapporto di vicinanza tra SuperNodes misurando la latenza di connessione, in modo da approssimare sufficientemente la conformazione della rete sottostante su cui si appoggia (Internet). Per il resto, soffre delle stesse limitazioni di Gnutella 0.6.

Concludendo, il fatto che Gnutella 0.6 utilizzi una rete a due livelli gerarchici migliora la scalabilità del sistema e limita il fenomeno del flooding e la propagazione di messaggi inutili. Il limite al flooding è favorito dal basso numero di nodi costituenti la rete dei SuperPeer e dal fatto che essi preservino i LeafNode

dal traffico di KeepAlive e dai messaggi di Query non diretti a loro, utilizzando la QRP come tabella di routing. La scalabilità è influenzata positivamente anche dal fatto che i SuperPeer bilanciano il carico che devono sopportare da parte dei LeafNode rifiutando le connessioni che non sono in grado di sostenere, oppure facendo diventare SuperPeer alcuni dei propri LeafNode. La riduzione drastica del diametro della rete, limitata ai SuperPeer, permette inoltre di poter esplorare un maggior numero di peer e quindi di poter espandere la vista del sistema di ogni SuperPeer e, conseguentemente, dei LeafNode collegati a lui: tutto ciò va a beneficio del numero di risorse che un peer può raggiungere sulla rete con un messaggio di Query, limitando il numero di nodi che non possono essere raggiunti ed il numero di messaggi necessari.

Inoltre, diversamente dai sistemi centralizzati o parzialmente centralizzati come Napster e LDAP, i SuperPeer non rappresentano punti di fallimento, in quanto non gestiscono le risorse dei LeafNode e, in caso di crash improvviso, i LeafNode possono collegarsi ad altri SuperPeer o diventare SuperPeer grazie ai messaggi di Pong che contenevano gli indirizzi di altri SuperPeer; una volta ricollegati, comunicano al nuovo SuperPer l'indice delle proprie risorse, che entrerà a far parte della QRP del nuovo SuperPeer. In sostanza, Gnutella 0.6 rappresenta effettivamente un'evoluzione di GNutella 0.4 in quanto ne migliora sia la scalabilità che l'efficienza per quanto riguarda il numero di messaggi propagati, ma eredita dall'altro protocollo i difetti che aveva, che si ripresentano , seppur attenuati, nella rete dei SuperPeer.

2.1.4 Peer-to-Peer Systems for Discovering Resources in a Dynamic Grid

In [19] vengono proposti due sistemi adatti a gestire informazioni per Griglie computazionali, che si sono evoluti dai protocolli di rete non strutturati e decentralizzati tipo [3], e che sono basati sui **Routing Indexes**. Questi sistemi nascono per limitare il fenomeno del flooding, cercando di raggiungere un buon trade-off tra la gestione efficiente del routing delle query e la necessità di limitare i messaggi relativi agli aggiornamenti nei valori degli attributi, anche in caso di attributi dinamici. I sistemi inoltre si pongono l'obiettivo di superare la limitazione sul numero di risorse raggiungibili da una query causato dall'uso del TTL, usando la tecnica del Routing Index per tenere un indice distribuito ed utilizzarlo per instradare le query verso i peer che potenzialmente contengono le risorse ricercate, e contemporaneamente costruiscono un supporto per le range query multiattributo.

La prima soluzione prende il nome di **Tree Vector** ed è costituita fondamentalmente da una rete di Overlay strutturata ad albero, costruita su una rete simile a quella su cui si basa [3], dove i peer si connettono ad un numero limitato di vicini

e le risorse sono gestite direttamente da ogni peer, ed utilizzano meccanismi di KeepAlive per venire a conoscenza dei peer ancora presenti sulla rete.

Ogni risorsa ha una serie di attributi che la identificano, costituiti da una coppia (nome, valore); per ogni attributo, viene identificato il dominio dei valori che lo caratterizzano e viene suddiviso in k sottointervalli, con k diverso per ogni attributo. Data una specifica risorsa, l'indice per l'attributo A della risorsa è rappresentato da un vettore di k bit, con il bit che rappresenta il sottointervallo in cui rientra il valore di A settato a 1, e gli altri a 0. Il **bitmap index** locale dell'attributo A di tutte le risorse di un peer P è dato dall'OR logico di tutti i vettori di bit delle risorse contenute in P relativi a quell'attributo. Per costruire i Bitmap Index che condensano il valore degli attributi degli alberi radicati nei vicini di P, esso riceve da ognuno di questi un vettore di indici costituito dall'OR bit a bit di tutti i vettori di bit locali ai peer di ogni sottoalbero radicato in quel vicino, e li memorizza nel proprio Bitmap Index.

Quando un peer P riceve una range query multiattributo (sono possibili diversi operatori descritti in [19]), il peer la decompone in sotto-query, una per ogni attributo. Da ogni sotto-query viene costruito un vettore di bit che la rappresenta che è suddiviso negli stessi intervalli dell'attributo che rappresenta. Tutti i bit che rappresentano un sottointervallo contenuto nella range query vengono settati a 1, gli altri a 0. Tutti i vettori delle sotto-query vengono prima di tutto confrontati con il Bitmap Vector delle risorse locali, per verificare se ci sono risorse che soddisfano la query ed in caso affermativo viene spedito un QueryHit. Successivamente, per fare il routing della query solo verso vicini che hanno potenzialmente almeno un peer che può soddisfare la query, vengono utilizzati i Bitmap Index dei vicini, facendo lo stesso controllo che viene fatto in locale, cioè confrontando bit a bit i vettori delle sotto-query con i Bitmap Index dei vari attributi: la query viene poi inoltrata sul link di quei vicini che hanno i Bitmap Index che soddisfano tutte le sotto-query della query.

Per aggiornare i valori di un attributo di una risorsa, un peer P calcola il nuovo Bitmap Index e lo confronta con il vecchio: se non ci sono differenze, esso non viene propagato ai vicini in quanto non varierebbe il loro Routing Index, altrimenti viene inviato a tutti i vicini. Se un nodo riceve un nuovo Bitmap Index da un vicino, calcola il nuovo Bitmap Index di ogni suo vicino facendo l'OR bit a bit delle proprie risorse locali e dei Bitmap Index di tutti i vicini diversi da quello preso in considerazione, e lo propaga ai vicini soltanto se ci sono differenze con il loro vecchio Bitmap Index.

Grazie al Routing Index, il sistema migliora la scalabilità dei metodi puramente decentralizzati come [3], e supporta le range query multiattributo per ambienti di Grid Computing; inoltre con questo metodo si evita di spedire un messaggio di query verso quei pezzi dell'albero che non contengono sicuramente nessun peer con risorse che soddisfano la query. Il sistema ha però anche dei problemi, so-

prattutto nel mantenimento del sistema degli indici, in quanto l'inserimento o la cancellazione di un peer comporta l'aggiornamento forzato dell'insieme degli indici, anche se in [19] viene dimostrato che soltanto un insieme costante di peer viene coinvolto nel processo di aggiornamento. Inoltre, un altro problema riguarda il fatto che i nodi che hanno vicini con sotto-alberi grandi, avranno i Bitmap Index di molti attributi con molti bit a 1, portando ad un'inondazione di query verso questi nodi rispetto a nodi più vicini alle foglie dell'albero che rappresenta l'Overlay Network.

Per questo motivo, viene presentato un secondo sistema come evoluzione del primo. In questo caso, la rete assume la struttura di una **Foresta di alberi**, dove ogni peer appartiene ad un singolo albero, chiamato **gruppo**. Di conseguenza, un nodo può essere connesso con i vicini locali, che fanno parte dello stesso gruppo, e con vicini esterni, appartenenti a gruppi diversi. Chiaramente, si ha un cambiamento nella modalità con cui avviene la descrizione delle risorse a disposizione sui vicini. Infatti, per ogni peer si definiscono due indici fondamentali:

1. l'internal bitmap index, che descrive solo le risorse disponibili in un dato gruppo;

2. l'external bitmap index, che da una descrizione delle risorse disponibili seguendo un determinato link esterno ad un peer di un certo gruppo verso un peer di un altro gruppo

Quando un nodo P riceve una query, questa viene inoltrata a uno dei vicini interni solo se è presente un match, altrimenti viene inviata ad almeno uno dei gruppi esterni connessi al nodo P. Questo tipo di routing risulta meno efficace perchè i gruppi appartengono a una rete non strutturata, dove possono essere presenti dei loop. Per evitare questo problema, la query mantiene la lista dei gruppi attraversati. Inoltre, vengono aggiunte tecniche di query cache affinché ogni peer processi la richiesta una volta soltanto. Se un nodo genera un aggiornamento all'interno del gruppo G, per prima cosa questo messaggio viene inviato a tutti i nodi del gruppo G, e l'aggiornamento viene inviato anche ai nodi esterni appartenenti a gruppi connessi a G, se l'external bitmap index di G risulta cambiato.

2.1.5 Conclusioni sui metodi non strutturati

Possiamo concludere che i sistemi non strutturati non sono adatti a supportare sistemi di Information Service evoluti, con supporto alle range query multiattributo. Infatti, nei sistemi centralizzati o parzialmente centralizzati come [2, 20], i problemi maggiori sono dati dalla mancanza di scalabilità e dalla mancanza di tolleranza ai guasti, in quanto il malfunzionamento dei server comporta la perdita

di connessione tra i peer e l'impossibilità di accedere ai dati sulle risorse. Nei sistemi decentralizzati come [3], l'assenza di server centrali ha portato ad ottenere sistemi in grado di scalare con l'aumentare del numero dei peer, ad avere una buona tolleranza ai guasti (l'uscita di un peer in [3] non comporta nessuna perdita di dati, e l'eventuale partizionamento della rete non comporta inconsistenza, ma solo il raggiungimento di un minor numero di risorse) ed un protocollo semplice e flessibile di entrata e uscita dei nodi grazie alla mancanza di strutture di routing o di indicizzazione delle risorse da aggiornare; in compenso però, la mancanza di una struttura di rete che effettuasse un routing mirato delle query verso i peer che possiedono le risorse che le soddisfano in modo efficiente, ha portato ad utilizzare protocolli di propagazione delle query che usano il flooding, i quali portano ad un'inondazione di messaggi (nel caso peggiore ogni query viene risolta con un numero di hop dell'ordine di $O(N)$, con N numero dei peer della rete), peggiorando la scalabilità con il crescere della rete. Per migliorare questo problema, è stato introdotto il TTL, ma ciò porta ad un altro problema: in questo modo non si riesce ad esplorare tutta le rete per ricercare le risorse che soddisfino una query, non garantendo che se una risposta esiste nella rete, essa sia riportata al peer che ne faccia richiesta. Per limitare i problemi dei sistemi decentralizzati puri, sono stati introdotti prima architetture che sfruttano il concetto di SuperPeer come quelle descritte in [22, 23], che tentano di limitare il diametro della rete non strutturata e quindi cercare di ridurre il numero di peer inondati dal flooding.

Successivamente i sistemi a RoungIndex come [19] cercano di evitare o limitare il flooding instradando i messaggi di query solo verso aree della rete in cui almeno un peer possiede una risorsa che risponda alla query. Essi inoltre mantengono i punti di forza delle reti non strutturate con un'alta tolleranza ai guasti ed un basso costo di mantenimento della rete e per finire supportano le range query multi attributo. Questi sistemi comunque mantengono, anche se attenuati, i problemi dei sistemi non strutturati decentralizzati, come la possibile irrangiugibilità di una risorsa, i problemi relativi al flooding (che nei sistemi RoutingIndex si presenta man mano che la rete aumenta di diametro, e si attenua nei sistemi a SuperPeer, ma non viene in realtà risolto) e di conseguenza la scalabilità ottenibile, che con l'aumentare del numero di nodi della rete diventa impossibile da garantire.

Tutti questi problemi hanno portato allo sviluppo di sistemi strutturati che, tramite l'uso di Overlay Networks predefinite, hanno una più elevata scalabilità ed un sistema di routing più efficiente.

2.2 Sistemi P2P con Overlay Network strutturata

Questi sistemi hanno la caratteristica di avere una struttura di rete logica predefinita in modo tale che ciascun nodo possa essere raggiunto in un numero limitato di

passi, tramite precise politiche di routing che sfruttano le caratteristiche della rete logica per inoltrare i messaggi. Se una risorsa è presente, un sistema strutturato garantisce la sua raggiungibilità. La maggior parte di essi utilizza una struttura chiamata **DHT**, come [4, 5, 6].

L'aspetto chiave delle DHT[6] è l'utilizzo di funzioni hash per garantire il bilanciamento del carico. I nodi ottengono degli **identificatori** appartenenti ad uno spazio uniformemente popolato. Inoltre, una funzione hash applicata alla descrizione di una risorsa restituisce la sua **chiave**, che appartiene allo stesso spazio degli identificatori dei nodi, permettendo di allocare la risorsa sulla DHT. Le DHT permettono di superare i limiti dei sistemi P2P non strutturati, poichè ogni nodo mantiene una tabella di routing di dimensione $O(\log N)$, e se una risorsa è presente nel sistema verrà sicuramente localizzata in $O(\log N)$ hop, dove N rappresenta il numero di nodi che fanno parte della rete. Le DHT si sono rilevate il miglior sistema per risolvere le **query esatte**, ma non sono in grado di risolvere efficientemente query per range di valori. Ciò dipende dall'uso di funzioni hash che mappano le risorse in modo casuale sulla rete e non consentono di conservare alcuna nozione di località.

Per superare questo problema, alcuni sistemi come MAAN [7] definiscono delle multi-DHT per poter risolvere anche range query multiattributo: in questo sistema, ogni attributo è mappato su un ring Chord tramite l'uso di una funzione di hashing che preserva la località, cioè risorse con valori vicini vengono mappati su chiavi vicine. Successivamente le range query vengono risolte utilizzando il concetto di selettività di un attributo: in questo modo, scegliendo l'attributo più selettivo, si può ridurre il costo relativo alla risoluzione di una range query su DHT.

Altri sistemi, come Squid [8], utilizzano una DHT mappando gli attributi tramite l'uso di una funzione di hashing che preserva la località che mappa uno spazio d-dimensionale in uno ad 1 dimensione, chiamata **Hillbert Space Filling Curve**[9]: in questo modo più attributi possono essere mappati su un'unica DHT.

Il difetto di questi ultimi sistemi è però il costo di inserimento delle informazioni nelle DHT e l'inefficienza nella risoluzione delle range query; infatti nonostante essi risolvano il problema della risoluzione delle range query multiattributo, le soluzioni proposte hanno una complessita superiore a $O(\log N)$, dove N rappresenta il numero di nodi che fanno parte della rete, tipica della risoluzione delle query esatte. Inoltre, in presenza di attributi dinamici, occorre reindicizzare la risorsa sulla rete.

Gli ultimi sistemi proposti utilizzano approcci alternativi alle DHT come quelli basati sulla tassellazione di Voronoi[10]. I diagrammi di Voronoi hanno la caratteristica di restringere l'insieme dei vicini di un nodo, nel caso di 2 attributi, ad un

[6]Distributed Hash Table

insieme statisticamente costante. Inoltre, sfruttando le caratteristiche dello Small World[11, 12], definiscono un protocollo di routing greedy poli-logaritmico. Le range query vengono risolte in modo naturale grazie al fatto che questi sistemi indicizzano oggetti invece di nodi fisici, rappresentando gli oggetti in base al loro valore degli attributi come punti su un piano multi-dimensionale, dove ogni dimensione rappresenta un attributo. Gli oggetti che sono match per una determinata range query sono raggiunti per mezzo del routing greedy. I vari metodi basati su Voronoi si differenziano per come vengono raccolti i match della range query: una volta raggiunto lo spazio descritto dalla range query SWAM-V [13] effettua un flooding per recuperare tutti i nodi, mentre Voronet[14] non esplicita il metodo utilizzato.

Per superare l'inefficienza della tecnica basata sul flooding, sistemi successivi come quelli descritti in [15, 16, 17] costruiscono uno spanning tree distribuito per minimizzare i messaggi mandati sulla rete. Nel capitolo 3 sono approfondite le caratteristiche e i vantaggi e svantaggi delle reti basate su diagrammi di Voronoi.

2.2.1 Chord

Chord [4] è stato il primo sistema con Overlay Network strutturata, ad usare una DHT come struttura dati per indicizzare le risorse della rete. Come nella maggior parte delle DHT, i dati in Chord sono composti da coppie (chiave, valore) e sono definite operazioni di ricerca, inserimento e cancellazione. Chord utilizza uno spazio delle chiavi unidimensionale ad anello con modulo 2^m, dove m è il numero di bit utilizzati per rappresentare le chiavi. Ogni nodo in Chord è associato ad un identificatore unico ID, ricavato tramite una funzione di hashing (tipicamente la SHA-1[7]) a cui sono passati parametri quali, ad esempio, l'indirizzo IP del nodo e porta del servizio DHT. Come i nodi, anche gli oggetti hanno associato il proprio ID, la chiave dell'oggetto. Una generica chiave k è assegnata al primo nodo il cui identificatore è uguale o segue k nello spazio degli indirizzi. Questo nodo è chiamato il successore di k ed è rappresentato come $successor(k)$.

La figura2.6 mostra una rete Chord con 8 nodi e uno spazio degli indirizzi di 26 elementi. Il nodo N20 ha come ID 20 e gestisce gli oggetti con chiave 10 e chiave 15.

Ogni nodo in Chord mantiene due insiemi di vicini, la **successor list** e la **finger table**. I nodi nella successor list sono quelli che immediatamente seguono il nodo sull'anello, mentre i nodi nella finger table sono distribuiti esponenzialmente nello spazio degli identificatori. La finger table può contenere al massimo m elementi. Per un nodo con ID n l'i-esimo elemento della finger table contiene il riferimento all'ID del primo nodo che supera n di almeno $2^i - 1$ sull'anello. In Chord, questo

[7]Secure Hash Algorithm 1: una funzione di hashing uniforme e consistente

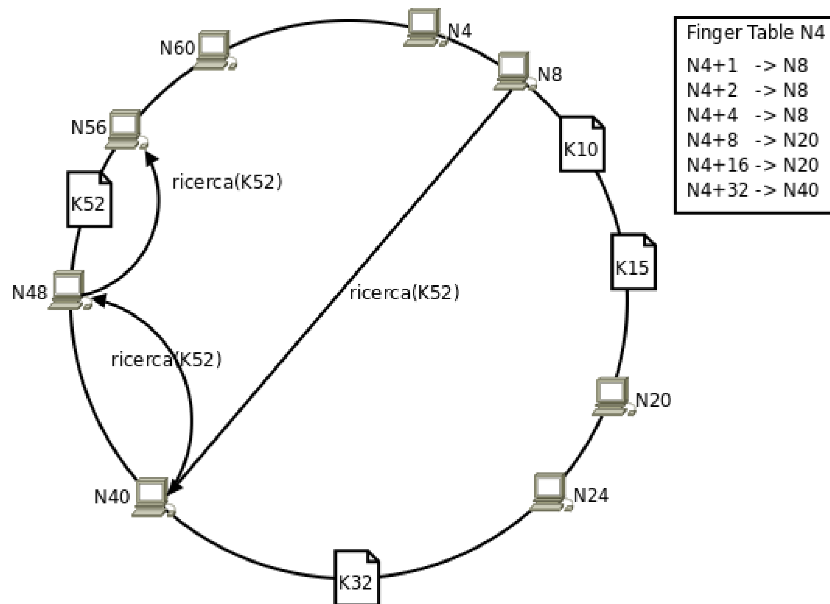


Figura 2.6: Esempio di rete Chord con chiavi di 6 bit

nodo è chiamato l'i-esimo finger del nodo n ed è rappresentato da $n.finger[i]$. Il primo finger è l'immediato successore di n (per $i = 1$). Ogni nodo quindi mantiene solo $O(\log(N))$ vicini in una rete Chord con N nodi. Ad esempio, in figura 2.6 i fingers di N4 sono N8, N20 e N40. Quando un nodo n vuole cercare un oggetto con chiave k deve inviare una richiesta al nodo x successore di k, $x = successor(k)$. L'algoritmo di routing funziona in questo modo: data una richiesta per la chiave k, il nodo cerca nella sua successor list il successore di k e vi inoltra la richiesta. Se il successore di k non è presente nella successor list, inoltra la richiesta al nodo j il cui ID è il maggiore tra quelli che precedono k nella finger table. Ripetendo questo processo, la richiesta si avvicina ad ogni passo al successore di k. Infine x riceve la richiesta, recupera i dati localmente ed invia la risposta al nodo n. Per esempio se N8 in figura 2.6 inizia una ricerca per la chiave K52, invia la richiesta al suo finger N40, quello più vicino a K52 nello spazio delle chiavi. N40 inoltra la richiesta a N48 che, a sua volta, la invia a N56. Dato che N56 è il nodo successore di K52, raccoglie il dato localmente ed invia il risultato a N8.

Quando invece un nuovo nodo n vuole entrare nella rete di Chord, esso prima di tutto calcola la propria chiave ID ed identifica il suo successore sulla rete, cercandolo tramite il normale routing di Chord a partire da un nodo a caso. Successivamente, crea la sua finger table, anche in modo lazy, cioè durante il suo funzionamento all'interno del sistema sfruttando i messaggi che gli vengono inoltrati dagli altri nodi. Per bilanciare il carico, il nodo n prende in carico dal successore le chiavi che sono minori o uguali al proprio ID. Quando un nodo n vuole lasciare

la rete Chord, passa le proprie chiavi al successore sull'anello che se ne prende carico, senza fare altre operazioni.

Ad intervalli regolari, ogni nodo di Chord aggiorna la propria successor list e la finger table per venire a conoscenza dei nuovi nodi e per eliminare quelli usciti, in modo da mantenere la consistenza della rete. Se il churn rate è più alto di questo intervallo, la consistenza non è garantita, in quanto i nodi potrebbero avere successor list non ancora aggiornate: infatti, in Chord la consistenza della rete ed il funzionamento del routing (anche se degradato rispetto al caso migliore, in cui tutte le finger table sono aggiornate) sono garantiti solo quando almeno le successor list sono aggiornate al momento in cui un nuovo nodo si inserisce.

Dato che i fingers sono distanziati esponenzialmente nella finger table, ogni hop dal nodo n al successivo copre almeno metà della distanza tra n e k. Il numero medio di hop per una ricerca è quindi $O(\log(N))$, dove N è il numero di nodi connessi alla rete. Anche la grandezza della finger table per ogni nodo come precedentemente detto è dell'ordine $O(\log(N))$, in quanto essa contiene $O(\log(N))$ nodi. Inoltre il consistent hashing permette un bilanciamento del carico tra i nodi. Per quanto riguarda l'affidabilità, deve essere garantita tramite replicazione delle chiavi su più nodi esplicitamente.

Di contro, in Chord abbiamo che l'uso di una DHT porta al supporto solo di exact match query monoattributo; la funzione di consistent hashing infatti distrugge la località delle informazioni e non permette il supporto di range query, inoltre permette la memorizzazione efficiente di un solo attributo e il routing sui valori di quell'attributo. Il costo di mantenimento della rete in presenza di entrata e uscita dei nodi è più alto rispetto ai metodi non strutturati e non è garantita la consistenza in caso di alto churn rate. Infine, Chord non è adatto a supportare attributi dinamici, in quanto il loro tasso di aggiornamento in molti casi è superiore al tempo necessario all'aggiornamento dell'informazione sull'anello.

2.2.2 Content Addressable Network

CAN [5] è un sistema che utilizza una DHT come struttura dati per indicizzare le risorse della rete. Come in Chord, gli attributi delle risorse sono composti da coppie (chiave/valore) e sono definite operazioni di ricerca, inserimento e cancellazione. Ogni nodo della rete CAN gestisce una parte, denominata **zona**, della DHT, così come le informazioni su un numero limitato di zone adiacenti. Le richieste di inserimento, ricerca, o eliminazione di una particolare chiave vengono instradate, attraverso zone intermedie, verso il nodo che gestisce la zona contenente la chiave. CAN utilizza una Overlay Network rappresentata da uno spazio virtuale cartesiano d-dimensionale, dove ad ogni dimensione corrisponde lo spazio delle chiavi di un attributo. La zona della DHT di cui un nodo è responsabile corrisponde a un intervallo di coordinate di questo spazio. Qualsiasi chiave K

è, pertanto, deterministicamente mappata su un punto P nello spazio per mezzo di una funzione hash uniforme, che genera una sequenza di bit, suddivisa in d sottosequenze, ognuna delle quali rappresenta la coordinata del punto in una determinata dimensione. Questa funzione distribuisce le chiavi fra i nodi in modo bilanciato e solitamente viene utilizzato l'algoritmo SHA-1 come in Chord. Anche i nodi sono mappati sullo stesso spazio delle chiavi per mezzo della stessa funzione hash. Le coppie (K, V) sono poi memorizzate dal nodo che è responsabile della zona in cui si trova il punto P.

Ad esempio, come si può vedere nella figura 2.7 (B), una chiave di coordinate $(0.1, 0.2)$ verrebbe conservata presso il nodo responsabile per la zona B. Nella stessa figura, confrontando (A) e (B), si può vedere la creazione della nuova zona F in seguito alla divisione a metà della zona appartenente al nodo E.

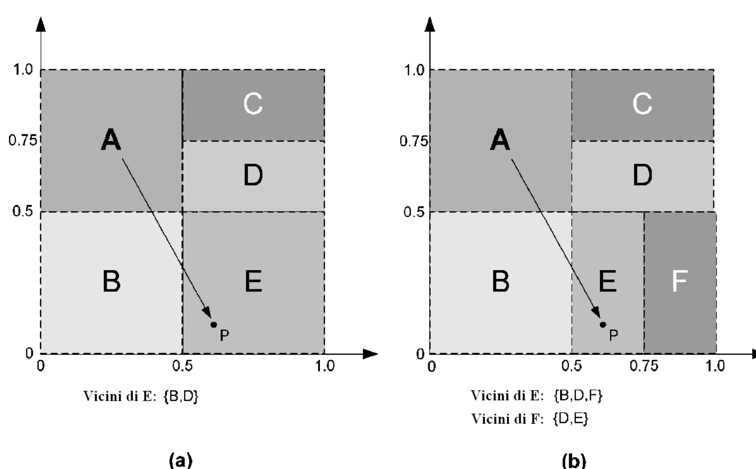


Figura 2.7: (a): Esempio di una rete CAN 2-d suddivisa in 5 zone; (b): Stessa rete dopo l'ingresso del nuovo nodo F

Per risolvere una query multi-attributo (o monoattributo) esatta con valori $V = (v_1, ..., v_d)$, ogni nodo applica la funzione hash per calcolare il punto P di coordinate $K = (H(v_1, ..., H(v_d))$ per poi recuperare il valore V dal nodo che gestisce P, se è presente nella sua DHT, oppure l'indirizzo IP e la porta del nodo . A meno che P ricada nella stessa zona del nodo richiedente, la richiesta deve essere instradata da nodo a nodo fino a raggiungere quello nella cui zona cade P. Ogni nodo di CAN ha una tabella di routing che contiene gli indirizzi IP dei nodi che gestiscono le zone adiacenti a lui e le loro coordinate, per consentire il routing tra punti arbitrari nello spazio. Il routing in CAN opera seguendo la retta che unisce, attraverso lo spazio cartesiano, il punto in cui è mappato il nodo sorgente a quello

del nodo di destinazione. Il nodo sorgente sceglie tra i punti delle zone adiacenti quello più vicino geometricamente al nodo di destinazione inviandogli la query.

Quando entra un nuovo nodo in CAN, esegue le seguenti operazioni:

1. identifica un nodo già esistente in CAN, utilizzando un meccanismo di bootstrap;

2. utilizzando un meccanismo di routing proprio, sceglie casualmente (o per mezzo di una funzione hash, usando il proprio indirizzo IP e la porta) un punto P e invia una richiesta di tipo JOIN al nodo che gestisce la zona in cui giace P. La zona è poi divisa a metà, di cui una è assegnata al nuovo nodo;

3. al nuovo nodo sono poi assegnate tutte le coppie (chiave ,valore) della tabella di hash del nodo che possedeva prima la sua zona e che ricadono in essa;

4. il nuovo nodo costruisce la sua tabella di routing con gli indirizzi IP dei suoi nuovi vicini; i vicini della zona suddivisa devono aggiornare le loro tabelle di routing in modo che rispecchino la nuova situazione;

Quando un nodo lascia volontariamente CAN, la zona che gestiva e le relative chiavi della tabella di hash sono assegnate ad uno dei vicini, scegliendo possibilmente quello con una zona della stessa grandezza, in modo da poterle fondere in un'unica zona più grande gestita dal vicino. In condizioni normali, un nodo invia messaggi di aggiornamento periodici a ciascuno dei suoi vicini, segnalando le coordinate della sua zona, la sua lista di vicini e le coordinate della loro zona. Se tale messaggio di aggiornamento non viene inviato per lungo tempo, i nodi adiacenti realizzano che vi è stato un fallimento e avviano un meccanismo di appropriazione della zona rimasta vuota, cercando sempre prima, se è possibile, di trovare un vicino che abbia la zona della stessa grandezza per poterle fondere.

Concludendo, in CAN si cerca di creare una Overlay Network a forma di toro d-dimensionale usando degli algoritmi di ingresso e uscita dei nodi che minimizzino il numero di vicini di una zona; in [5] viene dimostrato che il numero dei vicini di un nodo è indipendente dal numero dei nodi della rete, ma solo dal numero delle dimensioni (e quindi degli attributi), in modo che ogni nodo abbia una tabella di routing dell'ordine $O(d)$.

Per quanto riguarda la risoluzione di una query multiattributo, possiamo vedere che dato uno spazio d-dimensionale, applicando il normale routing di CAN, la lunghezza media di un path è dell'ordine $O(N^{\frac{1}{d}})$ lungo una dimensione, ottenendo un tempo di risoluzione totale della query dell'ordine $O(d * N^{\frac{1}{d}})$, con N numero totale dei nodi della rete; la complessità è superiore a quella ottenibile con Chord,

ma in CAN viene garantito il supporto alle query multiattributo, che non vi è in Chord. Inoltre il routing ha un discreto livello di fault tolerance, in quanto sono presenti più percorsi (non ottimali) per giungere al punto destinazione che passano dai vicini di un determinato nodo, quindi se un vicino fallisce, il nodo può sceglierne un altro; se tutti i vicini sono temporaneamente falliti, il nodo esegue un algoritmo ad anello che aumenta di raggio ad ogni passo, fino ad individuare un nodo di una zona più lontana. Per la fault tolerance sui valori delle risorse, occorre un meccanismo esplicito per garantirlo tramite la replicazione dei dati, facendo i modo che più nodi possano gestire una determinata zona. Concludendo CAN ha una scalabilità maggiore di quella dei sistemi non strutturati, così come Chord, per quanto riguarda il numero di messaggi mandati sulla rete per la risoluzione delle query.

Il problema maggiore in CAN è che, come in tutti i sistemi strutturati, ci possono essere degli intervalli di tempo in cui la rete è inconsistente e, nonostante la limitazione sul numero dei vicini, gli algoritmi di inserzione e cancellazione dei nodi sono piuttosto pesanti (a causa delle fusioni e divisioni delle zone). Questo rende CAN, come Chord, non adatto a gestire alti churn rate. Non riesce a gestire neanche range query a causa della perdita di località della funzione hash che tenta di distribuire uniformemente le chiavi nello spazio di indirizzi per bilanciare il carico. Infine, CAN non è adatto a memorizzare attributi dinamici, in quanto spesso il tempo di aggiornamento di questi è inferiore al tempo necessario per rimappare la chiave sulla DHT.

2.2.3 Multi-Attribute Addressable Network

Multi-Attribute Addressable Network (MAAN) [7] sfrutta Chord per ottenere un sistema che supporti la risoluzione di range query con più attributi. Una risorsa è identificata da più coppie attributo-valore. Per ogni attributo viene creata una funzione hash che ha il compito di mappare il valore dell'attributo stesso nell'intervallo $[0, 2^m - 1]$, dove m è la dimensione in bit dello spazio degli indirizzi.

Per inserire una risorsa nel sistema si memorizza, per ogni attributo, il suo valore al nodo opportuno, rappresentato dal successore sull'anello di Chord del valore stesso. In realtà le informazioni che si memorizzano, oltre al valore dell'attributo, sono diverse. Fra queste troviamo gli estremi che identificano il nodo proprietario della risorsa (tipicamente: indirizzo ip e porta) e, centrale per il funzionamento di MAAN, sono memorizzate tutte le altre coppie (attributo/valore) relative alla risorsa che si vuole registrare. Di fatto una risorsa è registrata un numero di volte pari al numero degli attributi.

Nella soluzione proposta da MAAN risolvere una range query significa risolvere tutte le sottoqueries di cui è composta, una per ogni attributo, ed effettuare

successivamente un'intersezione dei risultati. L'obiettivo è quello di riuscire a risolvere la query completa, sfruttando la DHT per risolvere solamente una delle sottoquery, quella che corrisponde all'attributo dominante. Il primo passo per la risoluzione di una richiesta è quindi la scelta dell'attributo dominante (l'attributo che identifica la query principale da risolvere). In generale l'attributo dominante cambia da query a query e la scelta cade su quello che garantisce la massima selettività possibile per permettere di ridurre il numero di nodi da interrogare. Si definisce **selettività** di un attributo il rapporto tra la dimensione dell'intervallo di ricerca specificato nella range query (calcolato sui valori degli estremi dell'intervallo tramite la funzione di hashing) e la dimensione del dominio dell'attributo stesso (che applicando la funzione di hash, coincide per tutti gli attributi alla grandezza dello spazio degli attributi, cioè $2^m - 1$). Questa scelta è centrale in un modello di prestazioni come quello delle DHT, in cui si cerca di minimizzare il numero di messaggi che transitano sulla rete. Dopodiché si interrogano in successione i nodi il cui ID è nell'intervallo dei valori dell'attributo dominante. Questo è reso possibile dal fatto che le funzioni hash in MAAN sono scelte in modo che, diversamente da Chord, preservino la località, come descritto in [7]: questo significa che dato un range di valori per un attributo A $[u, l]$ i suoi valori stanno tutti nell'intervallo di chiavi $[H(u), H(l)]$ dove H è la funzione hash scelta. Considerando che:

1. tutta la query è trasportata nel messaggio di richiesta
2. tutte le coppie attributo-valore della risorsa sono registrate

ogni nodo interrogato trova le risorse che soddisfano la query eseguendo un intersezione in locale, senza l'invio di ulteriori messaggi.

In conclusione l'architettura di MAAN ha una complessità del routing che è indipendente dal numero degli attributi, che garantisce scalabilità per le query multiattributo, la cui complessità è $O(\log(N) + N * S_{min})$, con N numero dei nodi della rete, che, in caso di S_{min} piccolo, diventa come quella di Choord, cioè logaritmica sul numero dei nodi. I possibili overhead si identificano nella quantità di memoria per memorizzare tutta la risorsa e, aspetto più rilevante, nell'alto costo di aggiornamento nel caso in cui il valore di un attributo cambi; questo rende ancora peggiore il supporto agli attributi dinamici rispetto a Chord. MAAN quindi mantiene una buona scalabilità sulla risoluzione delle query a patto che queste siano prevalenti sugli aggiornamenti dei valori degli attributi, in quanto la modifica degli attributi di un oggetto è molto costosa.

2.2.4 Scalable Wide-Area Resource Discovery

SWORD[27] è un'architettura complessa che ha come obiettivo la risoluzione di range query usando DHT. Oltre alle range query basate sugli attributi di un nodo, SWORD si distingue per aver integrato numerose funzionalità aggiuntive al processo di resource discovery. La possibilità data all'utente di determinare gruppi di nodi, con caratteristiche anche diverse, nella specifica di una query, l'utilizzo di attributi inter-nodo (ad esempio la latenza fra due nodi) e l'introduzione di un sistema di costi per una valutazione qualitativa delle soluzioni trovate, permettono a SWORD di essere considerato uno degli approcci più completi nel campo del resource discovery su DHT.

L'obiettivo di SWORD (e di un sistema di resource discovery in generale) è quello di permettere agli utenti di localizzare su una rete un sottoinsieme di risorse sulle quali eseguire le proprie applicazioni. Gli utenti possono esprimere i requisiti per un certo insieme di nodi, specificando due tipi di intervalli. Un intervallo più ampio che è il range richiesto, ed un altro, sottoinsieme del primo, che rappresenta il range desiderato. Inoltre è possibile per l'utente definire, a livello di query, gruppi creando di fatto dei veri e propri clusters virtuali di nodi. La selezione di nodi al di fuori dell'intervallo desiderato, ma comunque dentro quello richiesto, aggiungono un costo, detto **penalità**, alla risposta. La penalità rappresenta la distanza fra la richiesta di un utente e l'effettiva soluzione trovata.L'obiettivo finale di SWORD è quindi quello di trovare la risposta che soddisfa la query degli utenti con la minore penalità possibile.

In figura 2.8 è rappresentata l'architettura di SWORD.

Il processo di resource discovery è iniziato dall'utente (punto 1) il quale sottometteuna query in XML, inviandola ad un nodo qualsiasi facente parte di SWORD. La query è poi inoltrata al **distribuited query processor** (punto 2), componente che si occupa della risoluzione distribuita (punti 3 e 4). Come avviene l'effettiva risoluzione della query e come vengono scelti i nodi lo spiego successivamente. I nodi candidati trovati sono inviati all'**optimizer** (punto 5), il quale, dopo una fase di ottimizzazione, invia un sottoinsieme di essi all'utente (punto 6).

Una query in SWORD è rappresentata in formato XML ed è divisa logicamente in tre parti. La prima parte definisce i vincoli sulle risorse da utilizzare per la risoluzione della query, parametrizzando, ad esempio, il numero massimo di nodi da contattare nella fase di risoluzione. Nella seconda parte sono definiti gli intervalli per gli attributi intra-nodo e inter-nodo. Inoltre vengono definiti i gruppi, ognuno dei quali può avere vincoli diversi a seconda delle esigenze. Infine nell'ultima parte sono definiti i vincoli inter-nodo fra nodi che appartengono a gruppi diversi. Ad esempio si può rendere obbligatoria l'esistenza di un nodo in ogni gruppo tale che la sua latenza fra lui ed un nodo di un altro gruppo non superi una soglia specificata.

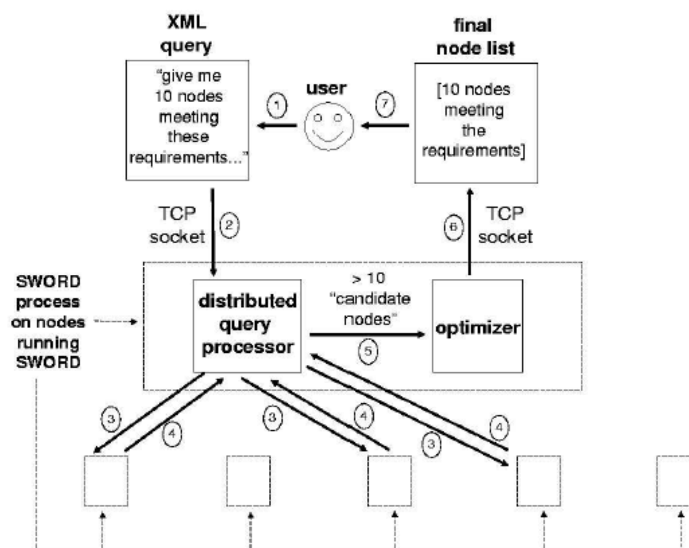


Figura 2.8: Architettura di SWORD

Per risolvere il problema delle range query multi attributo descritte come ho detto sopra, SWORD ha implementato e studiato quattro soluzioni differenti:

1. **SingleQuery** Nella fase di pubblicazione si inviano tutte le informazioni a *m* insiemi di servers, dove *m* è il numero di attributi con cui si descrivono le risorse. Ognuno di questi insiemi di server si occupa della gestione di un solo attributo, pur memorizzando l'intera descrizione della risorsa. Nella fase di risoluzione di query, la richiesta è inviata ad un nodo che gestisce una qualsiasi chiave di un qualsiasi attributo all'interno del range della query. Questo nodo poi inoltra la richiesta ai nodi successivi nello spazio degli indirizzi della DHT. Questo approccio è simile a quello specificato in MAAN [7].

2. **MultiQuery** Esiste un insieme di nodi che si occupa esclusivamente della gestione di un solo attributo. La fase di pubblicazione di una risorsa consiste in *n* messaggi, ognuno dei quali contiene una sola coppia (attributo, valore). Rispetto al metodo SingleQuery la dimensione del messaggio di aggiornamento è più piccola, ma nella risoluzione della query sono inviati *n* messaggi, uno per ogni insieme di server. I risultati ottenuti vengono poi intersecati localmente al nodo che ha inviato la query.

3. **Fixed** Questo è sostanzialmente un approccio centralizzato in cui si utilizza un numero variabile di server. La risorsa è inviata ad uno solo degli *n*

server presenti, scelto a caso all'avvio della query dal nodo, in modo da distribuire uniformemente il carico fra i server. Nella fase di risoluzione della query sono interrogati i rimanenti $n-1$ servers, che inviano i risultati come risposta.

4. **Index** Questo approccio è un ibrido fra la soluzione distribuita e quella centralizzata. Esistono dei nodi speciali, detti **index node**, che hanno la funzione di mappare un range di valori dello spazio delle chiavi con il nodo della DHT che lo gestisce. Periodicamente ogni nodo della DHT informa uno degli index node su qual'è il range che gestisce in quel momento. Per la fase di aggiornamento possono essere utilizzati gli stessi metodi delle soluzioni SingleQuery e MultiQuery. La query è inviata agli index node,i quali si occupano di inoltrarla ai nodi della DHT che gestiscono i range richiesti.

La chiave per associare le informazioni ai nodi presenti nella DHT è costruita tramite una funzione hash che preservi la località, come in MAAN [7]; questo permette di utilizzare per esempio l'approccio con attributo dominante nel metodo di risoluzione SingleQuery.

La seconda fase nel processo di risoluzione della query consiste nell'ottenere le informazioni riguardo gli attributi inter-nodo. Questo tipo di attributi sono memorizzati sul nodo che effettua la misurazione e non sono esplicitamente pubblicati. Una possibile ottimizzazione suggerita, è quella di raccogliere le misurazioni inter-nodo solo su alcuni nodi detti **nodi rappresentativi**, i quali memorizzano i valori degli attributi vicini. In questo modo, fra tutti i nodi candidati, possono essere contattati solo i nodi rappresentativi. Questo, unito alla prima fase di scelta effettuata sugli attributi intra-nodo, dovrebbe garantire un numero di comunicazioni non troppo elevate.

2.2.5 Voronet: Voronoi Overlay Network

Diagrammi di Voronoi

I diagrammi di Voronoi [28] prendono il nome dal matematico ucraino Georgy Fedoseevich Voronoi, che li definì e li studiò nel 1908, generalizzandoli al caso n-dimensionale. Questa tipologia di diagrammi era già stata utilizzata da Cartesio nel 1644 e, successivamente, ripresa dal matematico tedesco Dirichlet nel 1850, quando li applicò per i suoi studi sulle forme quadratiche, limitandosi a considerare i casi bi-dimensionale e tri-dimensionale[10].

Indichiamo con $dist(p,q)$ la distanza Euclidea tra i punti p e q. Nel piano abbiamo che

$$dist(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Sia $P = \{p_1, p_2, ..., p_n\}$ un insieme di n punti distinti nel piano detti **siti**. Definiamo il **diagramma di Voronoi** di P come la suddivisione del piano in n **celle**, una per ogni sito in P, con la proprietà che un punto q giace nella cella corrispondente ad un sito p_i se e solo se $dist(q, p_i) \leq dist(q, p_j)$ per ogni $p_j \in P$ con $j \neq i$. Indichiamo con $Vor(P)$ il diagramma di Voronoi di P, e con $V(p_i)$ la cella corrispondente al sito p_i, detta anche cella di p_i.

Il **lato** del diagramma di Voronoi appartenente a due regioni $V(p_i)$ e $V(p_j)$ adiacenti è il luogo dei punti equidistanti da p_i e p_j che giace sull'asse del segmento che connette i siti p_i e p_j.

Il **vertice** del diagramma di Voronoi appartenente a tre regioni $V(p_i)$, $V(p_j)$ e $V(p_k)$ è il punto equidistante dai tre siti p_i, p_j e p_k e coincide con il centro del cerchio passante per quei tre punti.

Consideriamo due punti p e q del piano e definiamo la bisettrice di p e q come la bisettrice perpendicolare al segmento $\overline{pq}$. Tale bisettrice divide il piano in due semipiani. Definiamo il semipiano aperto che contiene p come $h(p, q)$ e il semipiano aperto che contiene q come $h(q, p)$. Avremo che $r \in h(p, q)$ se e solo se $dist(r, p) < dist(r, q)$. Da ciò otteniamo la seguente osservazione.

Osservazione. $V(p_i) = \bigcap_{1 \leq j \leq n, j \neq i} h(p_i, p_j)$

Infatti $V(p_i)$ è data dall'intersezione di $n - 1$ semipiani e, quindi, ogni regione poligonale convessa di $Vor(P)$ è limitata da $n - 1$ vertici e $n - 1$ lati [29, 30].

Triangolazione di Delaunay

La definizione della triangolazione di Delaunay si basa sui diagrammi di Voronoi attraverso il principio della dualità. Due siti p_i e p_j si dicono contigui se le due regioni $V(p_i)$ e $V(p_j)$ hanno un lato in comune e la triangolazione di Delaunay si ottiene connettendo tali siti contigui come si può vedere in figura2.9.

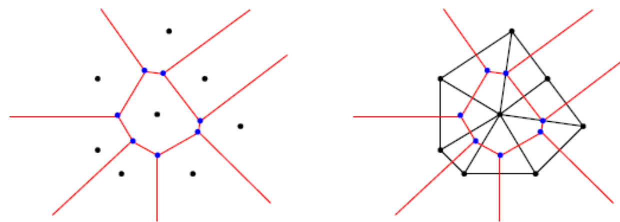


Figura 2.9: Esempio di diagramma di Voronoi e Triangolazione di Delaunay

Proprietà. Il cerchio circoscritto ad un generico triangolo di Delaunay avente come vertici p_i, p_j e p_k non contiene nessun altro punto appartenente alla triangolazione.

Si può però verificare un caso degenere, ovvero, possono esistere quattro o più punti concentrici; in tal caso vi sono due o più vertici di Voronoi coincidenti e la triangolazione di Delaunay non è univocamente determinata.

Small World

Negli anni '60, Stanley Milgram, eseguì una serie di esperimenti, il cui obbiettivo fondamentale era dimostrare che una qualunque persona nel pianeta è separata da ogni altra da un numero esiguo di relazioni. Milgram preparò una lettera che fu consegnata ad una persona nel Nebraska che aveva il compito di farla pervenire ad un'altra nel Massachusettes. Del destinatario, non venne fornito nessun indirizzo, ma era noto il nome, la professione e la città dove viveva (Boston). La lettera sarebbe stata consegnata solo ad una persona direttamente conosciuta che poteva avere qualche punto di contatto con il ricevente. Dopo una serie di tentativi, Milgram ottenne che il numero medio di passi per raggiungere il destinatario variava tra 5 e 6.

Watts e Strogatz [31] hanno ripreso l'idea di Milgram con lo scopo di definire un routing efficiente in reti P2P. In [31] presentano una rete caratterizzata da contatti locali e remoti (*long range*), e dimostrano che i messaggi possono essere trasferiti da un nodo ad un altro velocemente, in quanto i peer risultano separati da cammini molto brevi. L'obbiettivo di Watts e Strogatz era quello di individuare una topologia di reti sulle quali fosse possibile implementare algoritmi distribuiti in grado di individuare cammini brevi.

Nel modello dello small-world proposto da Kleinberg [11, 12] ogni peer viene identificato come un punto in una griglia NxN ed ognuno di essi conosce sia i propri vicini direttamente connessi in tutte le direzioni, sia un certo numero di vicini remoti, come si vede in figura 2.10.

In questo ambiente, Kleinberg mostra come un semplice algoritmo greedy, utilizzando esclusivamente informazioni locali, possa trovare cammini tra due punti qualsiasi sfruttando solo $O(\log N)^2$ link.

La rete VoroNet riprende il modello di Kleinberg con l'obiettivo di realizzare un routing polilogaritmico nel numero dei peer partecipanti.

Voronet: caratteristiche

Voronet [14] è una rete P2P che si basa sui diagrammi di Voronoi e differisce dalle altre soluzioni strutturate in quanto indicizza oggetti piuttosto che nodi fisici. Il suo obbiettivo è quello di fornire un supporto naturale per le range query, ciò è reso possibile perchè non sfrutta funzioni hash né per distribuire il carico tra i peer, né per assegnare gli identificatori. Piuttosto, VoroNet si basa su uno spazio **d-dimensionale**, dove ogni dimensione rappresenta un attributo. Ogni nodo fisico,

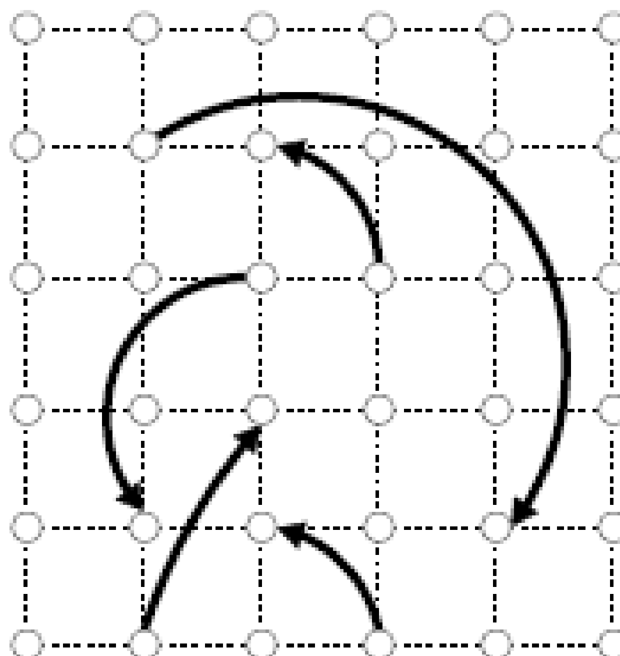


Figura 2.10: Esempio di diagramma secondo lo Small World di Kleinberg: le frecce nere rappresentano i vicini remoti

partecipante alla rete, pubblica un certo numero di oggetti che vengono rappresentati come coordinate nello spazio d-dimensionale e tale nodo viene identificato come punto individuato da queste coordinate. Lo spazio degli attributi viene mappato in un diagramma di Voronoi in d dimensioni.

Un altro obbiettivo fondamentale di VoroNet è quello di fornire un routing poli-logaritmico nel numero dei peer partecipanti. Per questa ragione il sistema si ispira al modello dello small-world proposto da Kleinberg [11, 12].

Per semplicità di trattazione, consideriamo il caso che **d** sia uguale a 2 e che ogni peer fisico pubblichi un solo oggetto.

Ogni oggetto O mantiene una **visione** del sistema, ovvero, conosce la posizione di un insieme di vicini, che vengono classificati in tre sottoinsiemi:

- **Voronoi neighbours** $VN(O)$, che sono gli oggetti le cui regioni condividono un lato della regione di O;
- **Long Range neighbours** $LR_N(O)$, che sono i vicini remoti definiti nella rete per rispettare le caratteristiche del piccolo mondo di Kleinberg;
- **Close neighbours** $CN(O)$, che sono quei vicini che giacciono su una circonferenza di raggio d_{min} con centro le coordinate di O.

È importante sottolineare che sia i Voronoi neighbours che i Close neighbours formano connessioni di natura simmetrica, ma ciò non vale per i Long range neighbours. Questo potrebbe risultare un problema se l'oggetto **t**, che appartiene a LR_N(**O**), lascia la rete. Infatti **t** non è in grado di contattare O per notificare la modifica della topologia e l'identità del nuovo LR_N. Per superare questo problema, O diventa un vicino di **t**, e prende il nome di **Back Long Range Neighbours** BLR_N. Quest'ultimo non verrà utilizzato per il routing, ma solo per mantenere consistente la topologia della rete. Tutti i vicini definiti in VoroNet sono visibili in figura 2.11. I tre insiemi di vicini, descritti sopra, sono necessari per garantire un **routing poli-logaritmico**.

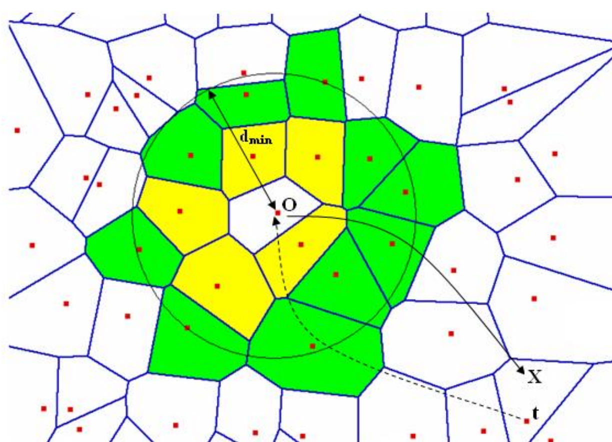


Figura 2.11: Esempio di rete di Voronoi, diversi colori indicano diversi insiemi di vicini

Come possiamo vedere in figura 2.11, O è un oggetto della rete. Le regioni di Voronoi colorate di giallo, rappresentano i Voronoi Neighbours di O, mentre le regioni in verde sono i Close Neighbours di O. La freccia uni-direzionale diretta verso il punto X, rappresenta il long link target, mentre la freccia tratteggiata è back long link di **t**.

È interessante sottolineare che i diagrammi di Voronoi hanno una caratteristica importante, ovvero, data una regione R, il numero medio dei confinanti è pari a 6. Infatti, in [14] si dimostra che la dimensione dell'insieme dei VN è dell'ordine di $O(1)$, ma ciò vale anche per i Close, i Long Range ed i Back Long Range neighbours.

In conclusione, possiamo dire che la dimensione delle strutture dati memorizzate da ogni peer, mediamente è costante. Questo risultato assume importanza quando si hanno operazioni di inserimento e cancellazione da parte dei peer, in quanto provocano modifiche alla struttura della rete di overlay. Mantenere consistente la topologia di VoroNet non è un'operazione costosa, proprio perché la

visione del sistema da parte di ogni peer è limitata ad un numero costante di vicini. Come conseguenza otteniamo che un peer, per informare gli altri di avvenuti cambiamenti, invia un numero limitato di messaggi.

VoroNet: Routing

VoroNet sfrutta un algoritmo di **routing greedy** che opera in questo modo:

- l'oggetto O riceve un messaggio M, il cui destinatario è il punto P;
- O sceglie il vicino N appartenente a $VN(O)$, $CN(O)$ o $LRN(O)$ che minimizza la distanza Euclidea con le coordinate di P;
- O inoltra ad N il messaggio.

P è un punto nello spazio d-dimensionale e può corrispondere ad un oggetto X, se le coordinate di P coincidono con quelle di X, altrimenti rappresenta un punto che giace all'interno di una regione di Voronoi. Nel primo caso, il messaggio M raggiungerà l'oggetto X, nel secondo caso verrà individuato l'oggetto che gestisce la regione dove P giace. In entrambe i casi, viene garantito che il numero di passi necessari per raggiungere P è **poli-logaritmico in** N, dove N è il numero totale dei peer della rete.

VoroNet: Inserimento

Supponiamo che un nuovo oggetto O voglia entrare a far parte della rete. Il primo compito sarà il calcolo delle coordinate del punto P che O va ad occupare all'interno dello spazio degli attributi. Ad esempio, se l'ascissa rappresenta la capacità della RAM in MB e l'ordinata rappresenta la velocità della CPU in GHz, un peer con 512 MB di RAM e 3 GHz di CPU corrisponderà al punto $(512, 3)$. Supponiamo che O conosca un oggetto X già connesso. A partire da quest'ultimo, verranno eseguite le seguenti operazioni:

1. si applica l'algoritmo di routing greedy, descritto nel paragrafo precedente, per raggiungere l'oggetto O' che soddisfa la relazione $O \in R(O')$.
2. O' si preoccupa di calcolare sia la regione di Voronoi di O che $VN(O)$, ma anche di informare i suoi vicini del nuovo inserimento. Inoltre, determina i $BLR_N(O)$.
3. Infine, O può calcolare i propri $CN(O)$ e sceglie i $LR_N(O)$.

Al termine dell'algoritmo, l'oggetto O ha piena conoscenza della grandezza della sua regione e dei vicini direttamente connessi a lui.

VoroNet: Cancellazione

Se un oggetto X decide di lasciare la rete, eseguirà la seguenti operazioni:

1. X calcola il nuovo diagramma di Voronoi, escludendo la sua regione, ed invia a tutti gli $Y \in VN(X)$ i confini aggiornati.
2. X invia un messaggio a tutti i nodi appartenenti a $CN(X)$ affinché possano aggiornare la loro vista del sistema.
3. Ed infine, assegna i suoi $BLR_N(X)$ ai nodi vicini più appropriati.

Terminata l'esecuzione delle operazioni sopra descritte, il nodo X è disconnesso dalla rete. Infatti i suoi vicini lo hanno cancellato dalla loro vista e nessuno di loro può più raggiungerlo.

In conclusione possiamo dire che VoroNet mostra delle performance interessanti sia dal punto di vista del mantenimento della rete di overlay, sia per la gestione delle range query che per il routing.

2.2.6 SWAM-V

SWAM-V (Small World Access Methods based on Voronoi diagram) [13] è un sistema P2P che nasce con l'obiettivo di fornire delle tecniche per risolvere in maniera efficiente **query esatte** e **range query**. Ogni nodo che fa parte della rete indicizza e memorizza i propri dati (**chiavi - key**) ed ognuno di essi viene descritto tramite un vettore *d-dimensionale* (o da una tupla formata da d attributi). Ad esempio, tali attributi possono essere le caratteristiche di un file musicale (nome artista, titolo, ecc.) se lo scopo della rete è la condivisione dei file mp3.

Per semplicità di trattazione supponiamo che ogni nodo memorizzi una e una sola tupla. SWAM-V costruisce un diagramma di Voronoi *d-dimensionale* sul quale viene mappato lo spazio delle chiavi. La triangolazione di Delaunay, che rappresenta il duale dei diagrammi di Voronoi, definisce la topologia base di SWAM-V e descrive le connessioni tra nodi vicini. Inoltre, vengono aggiunti dei *random link* in accordo al grafo definito dallo Small World di Kleinberg. In figura 2.12 vediamo come data una triangolazione di Delaunay, possiamo aggiungere a questa una serie di Random Link per ottenere infine la Overlay Network di SWAM-V.

In questo modo viene garantito che l'oggetto (gli oggetti) definito (definiti) da una query, se esiste (se esistono), venga raggiunto (vengono raggiunti) in $\log N$ hop, dove N è il numero di nodi della rete.

Una chiave in SWAM-V viene rappresentata come un vettore a d dimensioni. Definiamo la norma L_p, con $p \in Z^+$, la funzione che misura la "distanza" tra due

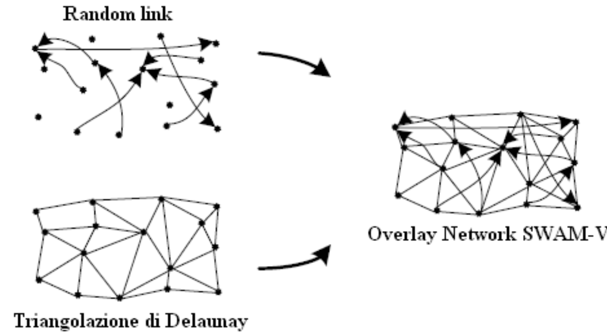


Figura 2.12: Overlay Network di Swam-V

chiavi $\overrightarrow{k_1}$ e $\overrightarrow{k_2}$ come $L_p(\overrightarrow{k_1} - \overrightarrow{k_2})$, dove $L_p(\overrightarrow{x}) = (\sum_{i=1}^{d} |x_i|^p)^{\frac{1}{p}}$. Inoltre, definiamo come $A(n)$ l'insieme dei vicini del nodo n.

Query esatte. Per risolvere una query esatta viene applicato l'algoritmo di **routing greedy**. Quando un nodo n_k, che pubblica la chiave $\overrightarrow{k}$, riceve una query $\overrightarrow{q}$, verifica se $L_p(\overrightarrow{k} - \overrightarrow{q}) < min_{\forall n_{k_i} \in A(n_k)} L_p(\overrightarrow{k_i} - \overrightarrow{q})$. In caso affermativo, $\overrightarrow{q}$ appartiene alla regione di Voronoi di n_k. A questo punto si possono verificare due possibilità, se $\overrightarrow{q} = \overrightarrow{k}$ il risultato è il nodo n_k, altrimenti non abbiamo soluzione per $\overrightarrow{q}$. Se $\overrightarrow{q}$ non appartiene alla regione di Voronoi di n_k, quest'ultimo inoltra la query al vicino n_{k_m} tale che $L_p(\overrightarrow{k_m} - \overrightarrow{q}) = min_{\forall n_{k_i} \in A(n_k)} L_p(\overrightarrow{k_i} - \overrightarrow{q})$.

Range query. Una range query viene definita da una chiave $\overrightarrow{q}$ e da un range **r** e viene eseguita in due fasi:

1. viene applicato l'algoritmo di routing greedy per raggiungere il nodo n_k nella cui cella giace la query $\overrightarrow{q}$.

2. A partire da n_k, ogni peer n che riceve la query per la prima volta, la inoltra a tutti i suoi vicini $m_{k'} \in A(n)$ se e solo se $L_p(\overrightarrow{k'} - \overrightarrow{q}) = r$.

In conclusione, SWAM-V è un sistema P2P che presenta molte similitudini con VoroNet sia per la topologia della rete, sia per l'uso del routing greedy per inoltrare dei messaggi da una sorgente a una destinazione, ma anche per gli obiettivi con cui nascono i due sistemi. Differentemente da VoroNet, SWAM-V fornisce una metodologia per la risoluzione di range query che impiega un routing di tipo **flooding** per recuperare tutti i possibili match, che si rivela efficace, ma non efficiente dal punto di vista dei messaggi spediti sulla rete.

2.2.7 VoRaQue: Voronoi Range Query

VoRaQue [15, 16] riprende il concetto presentato da VoroNet, ovvero sfruttare le caratteristiche dei diagrammi di Voronoi per fornire supporto a range query multi-attributo. Per poter fare questo, esso sfrutta i principi dello Small World [11, 12] per ottenere un routing greedy di tipo poli-logaritmico sulla dimensione della rete. Come Voronet, per semplicità di trattazione considera il numero di dimensioni $d = 2$ e che ogni nodo fisico pubblichi un solo oggetto.

L'obiettivo fondamentale di VoRaQue è quello di migliorare il supporto alle range query sfruttando un algoritmo diverso dal flooding usato da SWAM-V per visitare tutti i nodi che fanno parte dell'**Area di Interesse** della range query, cioè il poligono che rappresenta sul piano tutti i punti che rientrano nei limiti della query.

L'esecuzione di una query si svolge in due fasi:

- nella prima fase viene utilizzato un algoritmo di routing greedy per raggiungere un nodo che faccia parte dell'Area di Interesse, definito **nodo radice**.
- nella seconda fase viene eseguito un algoritmo di **compass routing** a partire dal nodo radice che permetta di costruire in modo distribuito un albero di copertura dell'Area di Interesse senza usare messaggi ridondanti

Compass Routing

Il **compass routing** [32] è un algoritmo di instradamento che viene applicato sui grafi geometrici.

Introduciamo il concetto che sta alla base del compass routing mediante un semplice esempio. Supponiamo che un viaggiatore arrivi alla città di Pisa e che voglia visitare la famosa Torre Pendente. Supponiamo che il turista, sprovvisto di una cartina, si trovi ad un incrocio dal quale può vedere la torre, con diverse strade $S_1, ..., S_N$ che può imboccare per raggiungere la sua meta. Il viaggiatore sceglie in modo naturale la via che in linea d'aria si avvicina di più alla direzione della torre.

Da un punto di vista matematico, possiamo modellare la mappa di una città tramite un grafo geometrico, dove gli incroci vengono rappresentati con dei vertici, mentre le strade con dei segmenti. Definiamo in modo più formale le sue proprietà.

Dato un grafo, supponiamo che da un vertice iniziale s si voglia raggiungere il vertice t e che tutte le informazioni disponibili ad ogni passo siano le coordinate di t, la posizione corrente p e la direzione dei lati incidenti sul vertice corrente p. A partire da s, ad ogni passo, possiamo scegliere la direzione da percorrere,

attraversando il lato incidente sulla posizione corrente p che forma l'angolo più piccolo con il segmento che connette p con t.

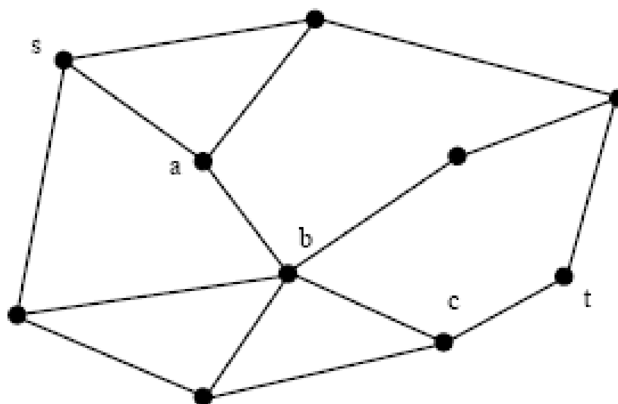


Figura 2.13: Cammino risultante dall' applicazione del compass routing da s a t

L'obiettivo primario del compass routing non è quello di trovare il cammino più breve per connettere due vertici, ma garantire il raggiungimento della destinazione. Questo non è sempre vero, osserviamo il grafo in figura 2.14.

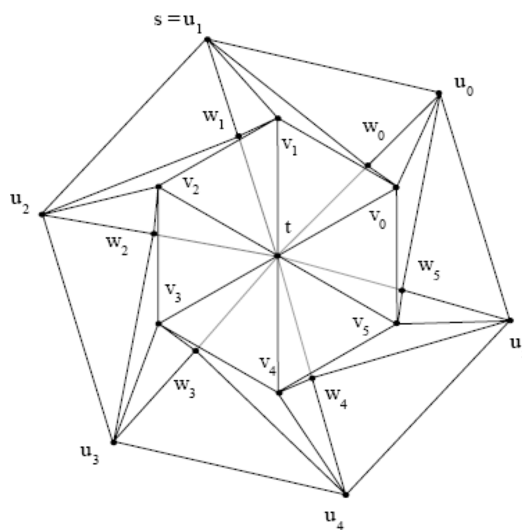


Figura 2.14: Esempio di compass routing che non raggiunge il punto t da nessun punto u_i

In questo caso vogliamo identificare un tragitto da u_1 a t usando il compass routing, ma il risultato è un ciclo infinito sull'insieme di vertici $\{u_1, w_i : i = 0, ..., 5\}$. Diremo che un grafo geometrico G supporta il compass routing se per ogni cop-

pia dei suoi vertici s e t, tale algoritmo di instradamento, partendo da s, restituisce sempre un cammino da s a t.

Teorema 1 *Sia P_n un'insieme di n punti del piano, allora la triangolazione di Delaunay di P_n, $D(P_n)$, supporta il compass routing.*

Dimostrazione: Supponiamo di voler raggiungere il vertice t a partire s. Mostreremo che se il compass routing sceglie di attraversare il lato sv di $D(P_n)$ allora la distanza da v a t è strettamente minore della distanza tra s e t. Sia $\overline{st}$ il segmento che unisce s a t, e supponiamo che intersechi il triangolo con vertici $\triangle sxy$. Sia C il cerchio sul quale giacciono s, x, y. Sia c il centro di C e s' l'immagine speculare di s rispetto alla linea che unisce c e t (vedere figura 2.15).

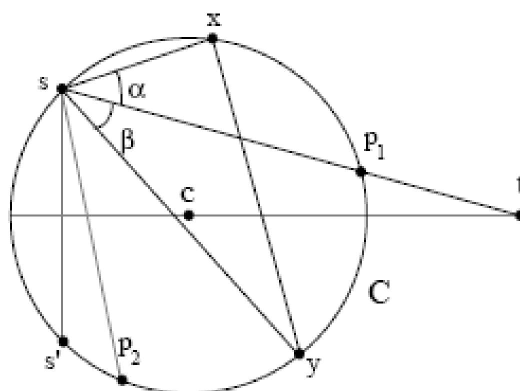


Figura 2.15: Routing sulla triangolazione di Delaunay

Siano α e β gli angoli formati tra $\angle xst$ e $\angle tsy$ rispettivamente. Si possono verificare due casi:

1. $\alpha < \beta$. In questo caso il compass routing sceglierà il lato sx e dalla figura 3.13 è possibile vedere che la distanza tra x e t è minore rispetto a quella tra s e t.

2. $\beta \leq \alpha$. Sia P_1 il punto di intersezione tra C e il segmento che unisce s e t, e P_2 il punto che giace su C tale che la distanza tra P_1 e P_2 coincida con la distanza che separa s e P_1. Dalla figura 3.13 possiamo vedere che P_2 giace sul semicerchio C' che unisce s a s' in senso orario. Poiché $\beta \leq \alpha$ y deve giacere su C' e quindi la sua distanza con t è più piccola rispetto alla distanza da s a t.

Questa tecnica di instradamento permette di costruire lo spanning tree a partire dalla Triangolazione di Delaunay. Le decisioni locali avvengono secondo la definizione di Compass Routing: il nodo A, data la radice R, calcola il nodo B come

suo genitore nell'albero, perché B é il vicino con l'angolo più piccolo rispetto ad R, come mostrato nella figura 2.16(a).

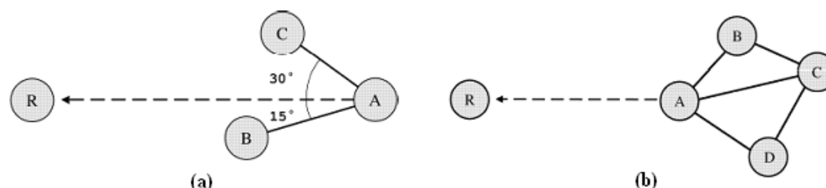


Figura 2.16: Esempio di compass routing che non raggiunge il punto t da nessun punto u_i

In particolar modo, il Compass Routing viene utilizzato per costruire l'albero per il multicast in maniera distribuita. Ad esempio, un nodo A, è il genitore del nodo C, perchè l'angolo $\angle RCA$ è più piccolo rispetto agli angoli $\angle RCD$ e $\angle RCB$. Nello stesso modo, B e D non sono i genitori di C perché $\angle RCA < \angle RCB$ e $\angle RCA < \angle RCD$, (vedere figura 2.16(b)). Se ogni nodo effettua il calcolo sopra descritto, si ottiene lo spanning tree con radice R [33].

Non è scopo di questa tesi illustrare l'algoritmo implementato per il compass da VoRaQue. Per maggiori dettagli vedere [15].

In [17] viene identificato un problema nella costruzione dell'albero di copertura: infatti l'algoritmo di compass routing utilizzato in [15, 16] necessita ad ogni passo di avere informazioni sui nodi vicini dei vicini rispetto al nodo considerato. Quindi affinché un nodo possa decidere a quali vicini inviare un messaggio, il nodo stesso dovrà richiedere ad ogni vicino la lista dei nodi che lo circondano. Questa necessità, nonostante la caratteristica di ogni nodo di avere statisticamente in media 6 vicini, implica che vengano inviati 6 messaggi di richiesta informazioni e 6 di risposta per ogni messaggio necessario per la creazione dell'albero di copertura.

Per evitare che l'efficienza del routing fosse fortemente penalizzata da questa necessità, è stata fatta un'importante ottimizzazione che riduce al minimo la ridondanza dei messaggi. Ogni nodo viene dotato di una cache in cui salva le informazioni ricavate dai propri vicini, in modo tale che per la query successiva che passa per il nodo, sia possibile individuare direttamente i figli nell'albero di copertura. Quest'ottimizzazione non è più sfruttabile nel caso in cui ci sia una modifica nella rete che cambia l'assetto dei nodi, per cui un nodo non può essere più sicuro dell'affidabilità dei dati ed è costretto ad invalidare l'intera cache.

In [17] il problema viene superato sfruttando un algoritmo a due fasi:

1. la prima consiste nell'**ordinare** opportunamente i nodi vicini.
2. la seconda consiste nella valutazione degli angoli formati dai vicini con la radice dell'albero di copertura

Questo permette al nodo padre di identificare con un algoritmo locale la lista dei nodi figli nell'albero di copertura, senza dover inviare messaggi ai vicini.

Anche in questo caso non è scopo di questa tesi illustrare l'algoritmo implementato, per maggiori dettagli vedere [17].

In conclusione, possiamo dire che VoRaQue, opportunamente modificato dal lavoro in [17], migliora le già interessanti performance raggiunte da Voronet[14]. Infatti, l'algoritmo greedy ha ancora la proprietà di essere poli-logaritmico rispetto al numero di nodi della rete. Invece, viene migliorata la complessità di risoluzione delle range query, sostituendo il flooding di SWAM-V[13] con un più efficiente algoritmo di compass routing, che permette di ottenere una complessità in numero di messaggi lineare rispetto al numero dei nodi che fanno parte dell'Area di Interesse della query, evitando messaggi ridondanti.

2.2.8 Conclusioni sui metodi strutturati

Possiamo concludere che i sistemi strutturati, a differenza di quelli non strutturati, sono adatti a supportare sistemi di Information Service evoluti, con supporto in alcuni casi anche alle range query multiattributo.

Tutti i sistemi descritti sono decentralizzati. La presenza di una Overlay Network strutturata permette un routing mirato delle query verso i peer che possiedono le risorse, permettendo di scalare efficientemente all'aumentare della dimensione della rete e del numero delle query rispetto al flooding utilizzato dalla maggior parte dei sistemi non strutturati. Inoltre garantiscono di ottenere tutti i match ad una query.

I primi sistemi presentati basati sulle DHT, come [5, 4, 6], permettono una efficiente risoluzione delle query esatte monoattributo e l'utilizzo di una funzione di hashing uniforme porta ad un bilanciamento del carico tra i nodi. Per contro, essi non permettono la risoluzione di range query perché la funzione di hashing unifome distrugge la località dell'informazione. I sistemi successivi, sempre basati su DHT, come [7, 8, 27] permettono la risoluzione di range query multiattributo ma con grosse limitazioni sulle prestazioni ottenibili: la funzione che mantiene la località non permette un bilanciamento del carico, il costo di aggiornamento delle informazioni e dell'inserimento o uscita di nuovi nodi in presenza di un elevato churn rate è elevato e la complessità di risoluzione delle range query multiattributo non è particolarmente efficiente.

I problemi più grossi dei sistemi strutturati basati su DHT riguardano innanzitutto il mantenimento della consistenza della struttura della rete: in particolare questo problema si presenta in caso di churn rate elevati o in caso di attributi dinamici. Un alto churn rate può portare al partizionamento della rete o ad inconsistenze che possono portare ad errori nel routing o nell'inserimento di nuovi nodi. Gli attributi dinamici causano una reindicizzazione troppo frequente, che

porta alla perdita della scalabilità ottenuta tramite un miglior supporto alle query. Un altro problema riguarda la bassa resistenza ai guasti: in caso di fallimenti di nodi, la struttura di rete può diventare inconsistente o partizionarsi. Per ovviare a questo problema sono stati introdotti diversi meccanismi di replicazione dei dati, che portano però un maggior costo dell'inserimento e dell'uscita dei nodi in termini di messaggi scambiati sulla rete.

In conclusione, tutti i sistemi strutturati hanno un costo di mantenimento della struttura della rete che non è presente nei metodi non strutturati: i metodi basati su diagrammi di Voronoi tentano di limitare il costo di mantenimento abbassando il numero di vicini di un nodo ad un numero costante, ma al costo di supportare un numero ridotto di attributi. Come si vedrà nel capitolo 3, i diagrammi di Voronoi multidimensionali con un numero di dimensioni maggiore di 2 tendono ad avere un numero di vicini che cresce esponenzialmente all'aumentare degli attributi. Infatti, VoRaQue per supportare query multiattributo usa un diagramma multidimensionale. Inoltre, in presenza di oggetti con attributi identici, si viene nuovamente a perdere il numero di vicini costante. Infine, il bilanciamento del carico non è possibile in quanto ogni nodo gestisce i propri oggetti. A loro favore c'è una miglior efficienza nel risolvere le range query rispetto alle DHT ed una migliore fault tollerance.

Capitolo 3

Hierarchical VoRaQue: l’architettura

3.1 Motivazioni

Questa tesi ha come scopo l’estensione di VoRaQue, l’architettura introdotta in [15, 16], per il supporto di query multiattributo. VoRaQue utilizza un approccio geometrico per la risoluzione di range query multi-attributo sfruttando le proprietà delle reti di Voronoi.

In una rete di Voronoi ogni peer pubblica oggetti, i quali sono caratterizzati da un certo numero di attributi; come in un database relazionale, l’insieme dei valori degli attributi di un oggetto rappresenta l’oggetto stesso, quindi oggetti con gli stessi valori degli attributi vengono considerati identici e vengono quindi rappresentati dallo stesso sito nel diagramma di Voronoi.

Gli oggetti con M attributi vengono mappati in una rete di Voronoi in uno spazio M-dimensionale, ed ogni oggetto occupa in questo spazio il sito corrispondente ai valori dei propri attributi o a trasformazioni continue e monotone definite su di essi. VoRaQue crea connessioni tra gli oggetti pubblicati cosicché oggetti con valori degli attributi ”simili” stiano vicini nello spazio di Voronoi multidimensionale, dove per vicinanza si intende la distanza Euclidea tra i punti. In questo modo si crea un supporto naturale per le range query. Le reti di Voronoi inoltre non utilizzano funzioni di hashing che distribuiscono oggetti sullo spazio degli identificatori come le overlay network basate sulle DHT, ma ogni peer fisico è responsabile degli oggetti da lui stesso pubblicati, come è stato descritto nel capitolo precedente. Questo approccio porta una serie di vantaggi e svantaggi:

1. La **resistenza ai fallimenti** è maggiore rispetto ad una rete che mappa gli oggetti su peer diversi da quelli che li pubblicano. Infatti, in seguito a fallimento di un peer non si perdono le informazioni sugli oggetti pubblicati

dagli altri peer. I vicini del peer hanno la possibilità di verificare che il peer non esiste più, per esempio perché lo selezionano come peer durante il routing (metodo reattivo) oppure inviandogli messaggi di keep alive. Successivamente i vicini del peer riconfigurano la rete togliendo dal diagramma di Voronoi i siti corrispondenti agli oggetti pubblicati dal peer fallito, ripristinando uno stato consistente della rete.

2. La **replicazione dei dati** non è strettamente necessaria perché se un peer fallisce solo le informazioni riguardanti i suoi oggetti vengono perse e questo è corretto in quanto in seguito al fallimento i suoi oggetti non sono più accessibili. Per quanto riguarda gli oggetti pubblicati dagli altri peer, questi rimangono correttamente accessibili perché gestiti dai peer stessi che li hanno generati. In conclusione, in una rete di Voronoi la replicazione non è necessaria per mantenere la consistenza.

3. Indipendentemente dalla distribuzione statistica delle query immesse dagli utenti (che potrebbero essere dirette principalmente verso peer più "popolari" rispetto ad altri), possiamo supporre che maggiore è il numero di oggetti gestiti da un peer, maggiore è il carico a cui tale peer viene sottoposto (in termini di query ricevute, tempo di elaborazione, messaggi scambiati, ecc...). Partendo da questa premessa, è possibile che nelle reti di Voronoi la caratteristica che ogni peer gestisca i propri oggetti possa portare ad uno sbilanciamento del carico: i peer che gestiscono più oggetti e che quindi gestiscono più punti nello spazio della rete di Voronoi avranno in media un maggior carico. Consideriamo un'applicazione di tipo file sharing, in cui ogni peer pubblica un file usando ad esempio attributi come nome, autore, etc... ogni peer gestirà più siti del diagramma di Voronoi, uno per ogni file con valori degli attributi diversi: questo fenomeno viene definito **multisiting**. Consideriamo ad esempio 4 peer fisici, ognuno dei quali registra alcuni file con attributi il NOME e la DIMENSIONE in Mbyte. Gli oggetti pubblicati dai peer sono i seguenti:

$$
\begin{aligned}
N1 : \{O1 &= [NOME = A.DAT, DIM = 1.0],\\
O2 &= [NOME = B.DAT, DIM = 1.5],\\
O3 &= [NOME = C.DAT, DIM = 2.0]\}
\end{aligned}
$$

$$
\begin{aligned}
N2 : \{O4 &= [NOME = D.DAT, DIM = 5.0],\\
O5 &= [NOME = E.DAT, DIM = 10.0],\\
O1 &= [NOME = A.DAT, DIM = 1.0]\}
\end{aligned}
$$

$$\begin{aligned} N3 : \{O1 &= [NOME = A.DAT, DIM = 1.0],\\ O2 &= [NOME = B.DAT, DIM = 1.5],\\ O6 &= [NOME = F.DAT, DIM = 8.0]\} \end{aligned}$$

$$\begin{aligned} N4 : \{O1 &= [NOME = A.DAT, DIM = 1.0],\\ O7 &= [NOME = G.DAT, DIM = 7.0],\\ O8 &= [NOME = H.DAT, DIM = 6.0]\} \end{aligned}$$

Nell'esempio è mostrato come alcuni file (in questo esempio i file A.DAT e B.DAT) possono essere pubblicati da più peer che ne possiedono una copia. É quindi possibile come mostrato in figura 3.1 che ad un sito del diagramma di Voronoi sia associata una lista di peer che gestiscono quell'oggetto, andando a formare quello che da ora in poi sarà definito un **cluster**. I nomi dei file sono mappati sull'asse delle ordinate, la dimensione dei file sull'asse delle ascisse. La funzione di mapping che trasforma la stringa del nome del file in un numero non è rilevante, tranne per il fatto che mantiene l'ordine alfabetico crescente dei nomi.

É importante notare che in questa tesi siamo interessati alla ricerca di risorse in ambiente distribuito. In questo caso ogni peer pubblica un solo oggetto che descrive le caratteristiche del peer stesso quali quantità di memoria, tipo di CPU, quantità di spazio disco, etc... e quindi non siamo interessati al problema del multi-siting. In questo caso di studio non ci sono problemi di bilanciamento del carico, ma si può verificare il fenomeno del clustering perché un insieme di peer può essere associato allo stesso sito del diagramma di Voronoi se sono caratterizzati dagli stessi attributi.

Nelle overlay network basate su DHT, sia gli identificatori dei peer che gli oggetti sono mappati su un unico spazio di indirizzi tramite una funzione di hashing che garantisce la distribuzione uniforme di questi oggetti sui vari peer; l'uniformità garantisce il bilanciamento del carico di cui le reti di Voronoi sono naturalmente sprovviste. La funzione di hashing uniforme distrugge però la località degli oggetti: per esempio dato un range di valori [0...100], interi, ed un insieme di indirizzi ad m-bits, i valori 0 e 1 sono vicini fra loro nel range, ma la funzione di hashing potrebbe mapparli in indirizzi che si trovano su peer lontani. Quindi le DHT sono adatte alla risoluzione di query esatte, mentre non possono essere considerate un supporto naturale per la risoluzione di range query se non viene garantita la località in qualche modo: in [7] ad esempio viene utilizzato un Order Preserving Hashing per ogni attributo, ma questo approccio aumenta sia la complessità di risoluzione delle range query che il costo di inserzione e di modifica

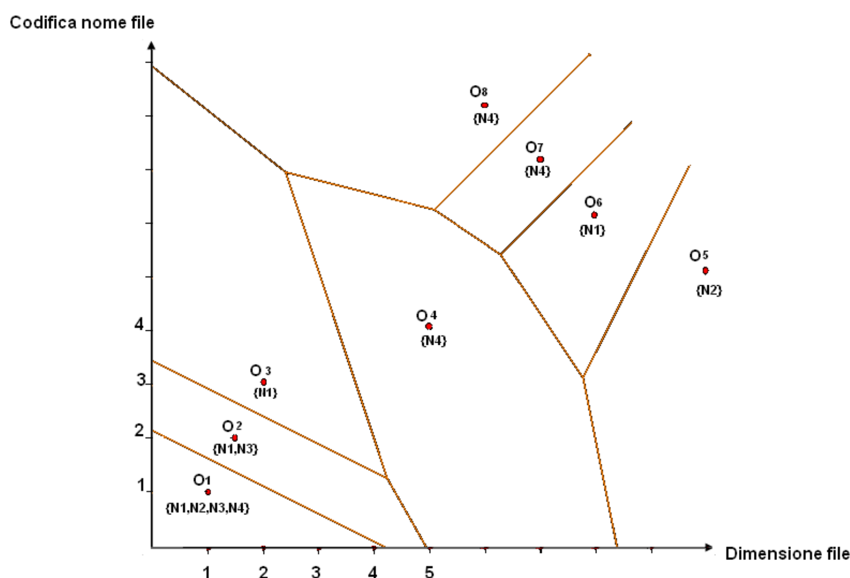


Figura 3.1: Esempio di multisiting e clustering in una rete di Voronoi

del valore degli attributi ed annulla il bilanciamento del carico garantito da una funzione di hashing uniforme.

Le reti di Voronoi hanno anche un'altra proprietà importante: in un sistema di Voronoi bidimensionale l'insieme dei VN (e così anche dei CN e degli LRN) è stato dimostrato essere limitato ad un ordine statisticamente costante di vicini, in media 6 [15, 16, 14]. Quindi anche le strutture dati necessarie a contenere i tre insiemi di vicini hanno complessità spaziale $O(1)$. Quando un peer entra o esce da una rete di Voronoi, esso deve comunicare i cambiamenti (e quindi mandare messaggi sulla rete) soltanto ai suoi vicini. Essendo questi in numero statisticamente costante il costo di riconfigurazione e mantenimento della consistenza della rete risulta molto basso rispetto ai più diffusi metodi basati su DHT come MAAN [7]. Anche gli algoritmi di routing beneficiano nello stesso modo del numero costante di vicini, come analizzato in [15, 16].

Con il crescere degli attributi e quindi delle dimensioni della rete di Voronoi il numero dei vicini non è più limitato ad un valore costante come dimostrato in vari lavori di generalizzazione delle reti di Voronoi bidimensionali come[14]. Quindi le considerazioni relative alle dimensioni delle strutture dati ed alla complessità degli algoritmi di risoluzione delle query e di ingresso/uscita dei peer dalla rete precedentemente fatti non valgono più nel caso di un numero di dimensioni maggiore di due.

Tenendo in considerazione tutte le caratteristiche descritte, questa tesi sceglie

una soluzione che sfrutta una rete di Voronoi bidimensionale come base per un'architettura che risolva range query su un numero di attributi $M > 2$. Nella nostra soluzione gli M attributi sono partizionati a gruppi di due ed ad ogni gruppo viene associato uno o più diagrammi di Voronoi bidimensionali: in questo modo si ottiene una rete **multi-livello**, dove ogni livello è costituito dai diagrammi di Voronoi di una determinata coppia di attributi. La struttura ottenuta forma un **albero di reti di Voronoi**, la cui radice è costituita dal livello superiore. Gli altri livelli sono costruiti di volta in volta laddove in un sito si forma un cluster di peer: se questo è sufficientemente grande si genera una nuova rete con i peer del cluster come componenti, usando i due successivi attributi non mappati sul precedente livello. Le reti dei livelli successivi al primo vengono quindi a dipendere dal punto in cui si genera il cluster, e sono quindi variabili in numero (se ne può generare fino ad una per ogni sito di un qualsiasi livello escluso il massimo).

Un cluster è sufficientemente grande se la sua dimensione supera una soglia S definita a priori: S può essere scelta in base a studi statistici sulla distribuzione dei peer sui punti della rete od al costo di generazione di un nuovo livello dell'albero, in termini di messaggi necessari.

La scelta degli attributi da associare alla rete di Voronoi del primo livello è quindi molto importante per 2 motivi:

1. Come vedremo in seguito, i cluster creano dei problemi di efficienza per la rete di Voronoi, quindi dobbiamo scegliere la coppia di attributi a cui statisticamente vengono solitamente assegnati valori il più possibile diversi fra gli oggetti pubblicati.

2. Definiamo l'**Area di Interesse** (AOI) della query come la zona del piano individuata dai vincoli definiti dalla query stessa. Diversamente da una rete di Voronoi multidimensionale, l'AOI così definita dalla query sugli attributi del primo livello non racchiude sempre tutti e soli i peer che sono match per la range query. Questo perché si perde la località dell'informazione sugli altri $M - 2$ attributi, e si va ad interrogare anche peer che soddisfano sicuramente la query sui due attributi scelti, ma che possibilmente non la soddisfano sugli altri attributi. Questo significa che maggiore è la dimensione della AOI, più numerosi saranno in proporzione i peer che devono essere interrogati per verificare se sono match per la query sugli altri attributi.

Definiamo **selettività** di un attributo il rapporto tra la dimensione dell'intervallo di ricerca specificato nella range query e la dimensione del dominio dell'attributo stesso. Dalle osservazioni fatte al punto 2 precedente, si deduce che i due attributi mappati sulla rete di Voronoi del primo livello devono avere la selettività più alta possibile, in rapporto alle query in cui vengono usati: una selettività maggiore corrisponde ad un rapporto più piccolo.

In un diagramma di Voronoi bidimensionale, la selettività di un attributo si ottiene normalizzando ad 1 il dominio di ogni attributo nella query. Ovviamente il numero di punti contenuti in una AOI dipende anche dalla distribuzione degli oggetti pubblicati dai peer, da cui consegue che un'AOI più grande non necessariamente possiede più punti. Se la distribuzione degli oggetti non è nota a priori, è comunque possibile utilizzare questa euristica per la scelta degli attributi da mappare sui diversi livelli della rete.

Consideriamo ad esempio, una rete di Voronoi costruita da peer che pubblicano oggetti con i tre attributi CPU (che rappresenta la velocità della CPU di un peer fisico espressa in GHertz, ipotizzando uno spazio dell'attributo [0...5]), MEM (che rappresenta la quantità di memoria RAM di un peer fisico espressa in Mbyte, ipotizzando uno spazio dell'attributo [0...64]) e spazio di DISCO (che rappresenta la quantità di memoria non volatile di un peer fisico espressa in GByte, ipotizzando uno spazio dell'attributo [0...4096]). Supponiamo che gli attributi mappati sulla rete di Voronoi di primo livello siano i primi due e prendiamo in considerazione le seguenti query:

$$A = \{3 \leq CPU \leq 4, 1.5 \leq MEM \leq 2.0, 100 \leq DISCO \leq 300.0\}$$

$$B = \{2 \leq CPU, 1.5 \leq MEM \leq 5.0, 0 \leq DISCO \leq 100.0\}$$

Passando ad analizzare le due query di esempio, si può notare che per gli attributi mappati sul primo livello della rete di Voronoi, la query A genera una AOI minima, mentre per la query B il primo attributo, essendo limitato superiormente solo dell'estremo superiore del dominio, genera una AOI più grande rispetto al terzo attributo DISCO, che ha una selettività maggiore.

D'altra parte, la modifica dinamica dell'ordine degli attributi dei vari livelli della rete di Voronoi in base alla selettività delle query non è realizzabile perché la rete di Voronoi dovrebbe essere completamente ricostruita, in quanto cambiando un attributo i punti verrebbero a trovarsi in una diversa collocazione spaziale. L'unica soluzione in questo caso sarebbe far uscire tutti i peer dalla rete e ripubblicare tutti i loro oggetti sulla nuova rete di Voronoi, soluzione molto costosa in quanto dipendente dalla dimensione della rete. Inoltre le query in corso dovrebbero essere bloccate durante la fase di ricostruzione della rete.

Per questi motivi, diversamente da molti metodi di risoluzione di range query basati su DHT come MAAN [7], che si basano appunto sulla selettività della query e scelgono dinamicamente l'attributo più selettivo per ridurre il numero dei peer da interrogare e quindi anche il numero di messaggi scambiati sulla rete, in Hierarchical Voraque vengono scelti gli attributi da mappare sul primo livello della rete di Voronoi mediante uno studio statistico di un certo pool di query.

3.2 Reti gerarchiche basate su SuperPeer

In questo paragrafo discutiamo la necessità di definire un insieme di SuperPeer che consentano di gestire opportunamente la rete di Voronoi gerarchica, Partiamo dall'analisi dell'algoritmo di routing necessario per risolvere range query multiattributo in reti gerarciche:

1. si prendono in considerazione gli attributi della query che sono mappati sulla rete di Voronoi di primo livello e si identifica l'AOI corrispondente.

2. se un peer non fa parte dell'AOI, esegue un algoritmo di **routing greedy** per raggiungere un punto della AOI, altrimenti passa al prossimo punto.

3. una volta raggiunto un sito dell'AOI, si esegue l'algoritmo di compass routing[32] per costruire un albero multicast per raggiungere tutti i peer della AOI.

4. ogni peer che fa parte dell'albero costruito dal compass routing soddisfa la query soltanto per gli attributi che fanno parte del primo livello della rete di Voronoi: occorre quindi che ognuno di essi verifichi se esso soddisfa la query sugli altri M-2 attributi. Se il peer fa parte di un cluster di dimensione maggiore di S, si propaga la query al livello inferiore della rete, altrimenti il peer controlla immediatamente il matching sugli altri attributi.

Il funzionamento degli algoritmi di routing greedy e di compass routing è presentato in [15, 16].

É importante notare come il passo 4 non sia eseguito in una rete di Voronoi in cui tutti gli attributi sono mappati nello stesso livello. In questo caso, infatti, la AOI contiene esattamente tutti e soli i punti che soddisfano la query e non sono necessari ulteriori raffinamenti.

Consideriamo ora le modifiche da apportare all'algoritmo di routing a causa della presenza di cluster di peer. Le considerazioni seguenti porteranno a motivare la presenza di SuperPeer nella rete.

Si nota che la formazione di cluster di peer accade più frequentemente su reti di Voronoi bidimensionali gerarchiche in quanto molti oggetti si differenziano su alcuni attributi, ma ne lasciano altri identici, come quelli mappati sul primo livello della rete. Un'altra causa potrebbe essere la **distribuzione** dei valori degli attributi degli oggetti: essi possono per esempio avere una distribuzione uniforme (e non generare cluster) oppure power-law [34].

In caso di una power-law, si avrà ad esempio un numero altissimo di punti su cui è mappato un singolo oggetto, e pochi punti costituiti da cluster su cui è mappato un numero altissimo di oggetti. I cluster introducono una serie di problemi:

1. l'assunzione statistica sul numero limitato dei vicini viene a cadere: un peer che possiede per vicino un sito corrispondente ad un cluster di peer, deve memorizzare come suoi vicini di Voronoi tutti i peer che fanno parte del cluster.

2. si presenta il problema di quale peer scegliere tra quelli facenti parte di un cluster per decidere il prossimo hop di routing, di tipo greedy o compass.

Analizzando il routing, se il sito scelto ad un determinato passo è associato in realtà ad un cluster di peer, si può osservare come il modo con cui uno qualunque di questi peer viene scelto non influisce sul routing perché dipende solo dalla conformazione dell'area di Voronoi. Il problema è che poiché il numero dei peer che costituiscono gli insiemi dei vicini non è più limitato superiormente, la scelta del sito successivo non risulta più $O(1)$ ma dipende dalla dimensione del cluster e nel caso peggiore risulta $O(N)$, con N numero di peer della rete.

Questa tesi affronta il problema della riduzione del numero di vicini di un sito nel caso di presenza di cluster di peer. L'architettura scelta per risolvere il problema è di tipo **ibrido**: per ogni cluster, una parte dei peer viene eletta SuperPeer, gli altri entrano a far parte dei peer gestiti dai SuperPeer utilizzando una nuova rete di Voronoi come Overlay Network. La struttura generale di questa architettura verrà descritta nel paragrafo successivo.

L'architettura scelta è inoltre a SuperPeer multipli. Eleggere un unico peer come SuperPeer per un cluster risolve completamente il problema dei vicini, che si riducono ad uno per ogni area di Voronoi, ma costituisce un singolo punto di fallimento per il cluster e, quindi, un collo di bottiglia per la risoluzione delle query. Il numero di SuperPeer scelti deve essere comunque tale da limitare superiormente il numero di vicini.

Si noti anche come la presenza dei cluster possa anche portare ad un miglioramento nell'efficienza del routing greedy nel caso di risoluzione di più query diverse. Supponiamo ad esempio che due messaggi di query abbiano raggiunto due punti diversi nell'esecuzione del routing greedy, e che scelgano lo stesso sito per l'hop successivo. In questo caso, poiché le due query sono diverse, è possibile inoltrarle in parallelo su due peer diversi del cluster, aumentando l'efficienza del sistema.

La presenza di cluster presenta anche un altro problema relativo al bilanciamento del carico delle query sui peer del cluster. Si può notare che una soluzione basata sul passaggio delle informazioni sul carico di lavoro dei peer di un cluster a tutti i peer dei siti vicini, i quali possono essere a loro volta dei cluster, è improponibile in termini sia di messaggi scambiati, sia della loro frequenza, in quanto l'informazione è altamente dinamica. La nostra soluzione consiste in un algoritmo basato sulla scelta casuale di un SuperPeer del cluster. Si noti anche che ogni

query deve essere quindi passata anche a tutti gli altri peer del cluster per verificare se essi la soddisfano sugli altri M-2 attributi. Questo implica un aumento dei messaggi da scambiare all'interno dei peer del cluster, che possono essere diretti a peer che potenzialmente potrebbero non essere soluzione della query.

3.3 Hierarchical Voraque: struttura generale

Consideriamo un insieme di peer che pubblicano risorse definite da un numero M di attributi; per semplificare consideriamo gli attributi di tipo numerico e che ogni peer pubblichi un unico oggetto, quindi una sola tupla di M coppie (Attributo, Valore), e che M sia pari.

I principali obiettivi che ci siamo posti nella definizione di Hierarchical Voraque sono la definizione di un'architettura che:

- risolva range query **multiattributo**, superando il problema delle reti di Voronoi multidimensionali, in cui il numero dei siti vicini di un sito cresce in modo esponenziale, come indicato in Voronet [14].

- scali in presenza di **cluster** di peer, la cui dimensione dipende dalla distribuzione del valore degli attributi delle risorse: più risorse hanno il valore degli attributi identico, più un cluster cresce di dimensione, per cui non è possibile limitare a priori superiormente la sua dimensione. Questo porta nuovamente ad avere un numero di vicini non limitato. Il problema verrà affrontato limitando il numero di vicini a $O(\log N)$, con N numero di peer dell'intera rete, nel caso peggiore.

- migliori la complessità della pubblicazione di una risorsa in una rete di Voronoi multidimensionale il cui migliore algoritmo di costruzione ha complessità $O(N^{\frac{d}{2}})$ [35] con N numero di peer della rete e d il numero delle dimensioni totali.

Questi obiettivi vengono raggiunti estendendo VoRaQue [15, 16]. Hierarchical Voraque divide l'insieme degli M attributi in due gruppi, come descritto nel paragrafo 3.1: due attributi vengono scelti in base alla loro selettività per essere mappati su una rete di Voronoi bidimensionale, che rappresenta il livello più alto dell'architettura. Gli altri attributi vengono considerati secondari e non rappresentati su questo piano. Nel caso in cui un sito della rete di Voronoi sia gestito da un unico peer, il secondo livello non è presente in quanto non necessario: il peer gestisce da solo tutte le query che riceve verificando se esiste un match su tutti i propri attributi. Nel caso in cui più peer pubblichino risorse con attributi primari identici, ma con possibili diversi attributi secondari, si forma un **cluster**. In questo

caso più peer vengono associati allo stesso sito p_i del diagramma di Voronoi. Per limitare il numero di vicini, per ogni cluster vengono eletti un certo numero di SuperPeer d'ora in poi indicati come SP_{p_i}, con p_i il sito dove si forma il cluster. Gli altri peer del cluster non hanno invece visibilità della rete di Voronoi di primo livello, ma sono collegati ai propri SP. La Triangolazione di Delaunay definita tra gruppi di SP definisce l'Overlay Network. Ogni peer n_i a livello L_i che non è stato eletto SuperPeer mantiene la lista dei SuperPeer del livello L_i. Questa lista viene creata al momento in cui n_i viene inserito nella rete e viene mantenuta in modo lazy, cioè ha un tempo di validità T dopo cui viene rinfrescata. Il valore di T dipende dalla frequenza con cui l'insieme dei SP viene modificato.

Il metodo con cui viene eletto un SP è ininfluente per la nostra tesi, ma è comunque possibile utilizzare per l'elezione le loro caratteristiche quali potenza di calcolo, quantità di memoria, o altri parametri. Il numero dei SP è variabile in base alla dimensione del cluster in modo da poter scalare con esso. Affinché questo numero sia il più possibile ridotto, ma sufficiente a mantenere un livello accettabile di servizio, il numero di SP è $\log(K)$, dove K rappresenta la dimensione del cluster. I SP sono **completamente connessi** fra loro, attraverso connessioni dedicate a:

- lo scambio dei messaggi necessario al bilanciamento del carico, se implementato
- lo scambio della struttura della regione di Voronoi del livello superiore qualora sia modificata, con i relativi nuovi vicini
- lo scambio del numero dei peer del livello inferiore che devono gestire, in caso di elezione di un nuovo SP o dell'entrata nel cluster di un nuovo peer

Ogni SP deve quindi tenere una tabella con i riferimenti agli altri SP, la cui dimensione è $\log(K)$, mentre il numero delle connessioni effettive è $\log^2(K)$.

É stato inoltre deciso di effettuare un'ottimizzazione che sfrutta il fatto che punti con attributi simili possiedono aree di Voronoi la cui unione è approssimabile con l'area di Voronoi che sarebbe presente nel piano se esistesse solo uno dei punti. Quindi definiamo una distanza r (raggio di assorbimento) tale che se un peer n deve essere inserito all'interno della rete in un punto p_i e $dist(p_i, q_i) < r$ dove q_i è un sito già presente nella rete, allora n entra a far parte dei peer associati a q_i, in modo da aumentare la probabilità e le dimensioni dei cluster in reti particolarmente dense, con regioni di Voronoi estremamente piccole.

Il seguente teorema definisce un limite superiore al numero di vicini di un peer in questa architettura.

Teorema 2 *Data K la dimensione del cluster più grande associato ai siti vicini di un certo peer ad un qualsiasi livello, il numero totale dei peer vicini è* $O(\log(K))$.

Dimostrazione: Consideriamo due casi:

- se non esistono cluster nei siti vicini, allora esiste un solo peer per ogni sito ed essendo i siti vicini statisticamente in numero costante come dimostrato in [14], abbiamo che i peer vicini sono $O(1)$, che corrisponde alla dimensione del cluster più grande.

- se sono presenti uno o più cluster associati ai siti vicini, mettiamoci nel caso in cui tutti i siti vicini hanno un cluster di dimensione K: nessuno può avere un cluster di dimensione maggiore perché per ipotesi K è il massimo, quindi questo è il caso pessimo. In Hierarchical Voraque, abbiamo fissato il numero di SP di un cluster al logaritmo della dimensione del cluster. Essendo i siti vicini statisticamente in numero costante otteniamo:

$$J * \log K = O(\log K)$$

dove J rappresenta il numero dei siti vicini. Da questo teorema otteniamo che se $K \approx N$, il numero totale dei vicini è $\log N$.

Analizziamo adesso come sono gestiti i peer di un cluster associato ad un sito p_i per ottenere una struttura efficiente. Per ottenere una soluzione scalabile, si è deciso di costruire una nuova rete di Voronoi per ogni coppia degli attributi secondari. Supponiamo di avere un cluster di dimensione K: tra gli $M - 2$ attributi secondari rimasti, si sceglie nuovamente la coppia che statisticamente risulta maggiormente selettiva. Tutti i peer del cluster si inseriscono, in base al valore di questi due attributi, nella nuova rete di Voronoi: $\log K$ di questi vengono eletti SP e restano visibili nella rete degli attributi primari, nel sito p_i. Gli altri invece non saranno visibili a quel livello. Per ammortizzare il costo di creazione della nuova rete, questa viene generata solo se la dimensione del cluster supera la soglia S definita come nel paragrafo 3.1. I peer che hanno i valori degli attributi primari che ricadono nel sito p_i ed entrano a far parte del cluster, vengono inseriti nella nuova rete di Voronoi nel punto corrispondente al valore della coppia degli attributi secondari mappati in questa rete. Essi quindi, una volta inseriti nella nuova rete, non sono più visibili nella rete degli attributi primari a meno che non vengano eletti SuperPeer.

Per ogni sito p_j della nuova rete in cui si viene a formare un nuovo cluster con dimensione superiore a S, viene generata una nuova ulteriore rete di Voronoi bidimensionale, usando una nuova coppia di attributi secondari. Se invece la dimensione del cluster non supera S, i peer vengono definiti **companions** e vengono trattati nello stesso modo dei SP di un cluster con dimensione maggiore di S. In sostanza, esiste una connessione completa tra di loro ed in più sono collegati ai loro vicini di Voronoi della rete di quel livello.

Generalizzando, la nostra architettura genera una serie di **livelli** gerarchici, uno per ogni coppia di attributi: questi livelli vengono numerati da L_0 a L_{Max} dove

$L_{Max} = M/2$ è il numero massimo di livelli possibili. Si ottiene quindi un'architettura strutturata ad albero: la radice è formata dal livello L_0, che è costituito da un'unica rete, su cui sono mappati gli attributi primari, il livello successivo da tutte le reti costruite sui due successivi attributi secondari i cui peer fanno parte di un cluster di un certo sito p_i della rete di livello superiore con dimensione maggiore di S e così via fino al livello L_{Max}. Una rete di Voronoi ad un determinato livello dell'albero può essere quindi identificata univocamente con un'**etichetta** costituita dal valore degli attributi dei livelli superiori, che risultano uguali per tutti i peer che ne fanno parte, ma diversi da quelli di ogni altra rete dello stesso livello dell'albero. Per quanto riguarda i cluster di dimensione superiore alla soglia S che si formano in reti al livello L_{Max}, non avendo ulteriori attributi secondari su cui costruire un nuovo livello di Voronoi, vengono trattati eleggendo sempre un numero $\log K$ di essi come SP e collegando gli altri peer del cluster direttamente a questi senza formare una nuova rete di Voronoi, come in [22]. Inoltre, i peer del cluster per definizione soddisfano tutti le query le cui AOI comprendono il sito associato al cluster: in questo caso se ne può scegliere un qualsiasi numero k in modo casuale, se la query richiede un numero di risposte k limitato.

Preso un qualunque cluster formatosi in un sito p_i di una rete di Voronoi di livello L_i, con una certa etichetta E, definiamo sottoalbero S_{p_i} l'insieme delle reti di Voronoi dei livelli inferiori in cui sono inseriti tutti i peer del cluster. Il numero totale dei peer di questo sottoalbero è dato dalla somma di tutti i peer della rete di livello L_{i+1} radicata in p_i e dal numero dei peer dei sottoalberi radicati nei siti di L_{i+1} che generano un cluster di dimensione maggiore di S, dove ogni peer che è stato eletto SuperPeer viene contato una sola volta in questa somma. Nel seguito questo numero sarà indicato con $Num(p_i)$, dove p_i è la radice dell'albero: la notazione è univoca perché ogni punto genera un diverso sottoalbero.

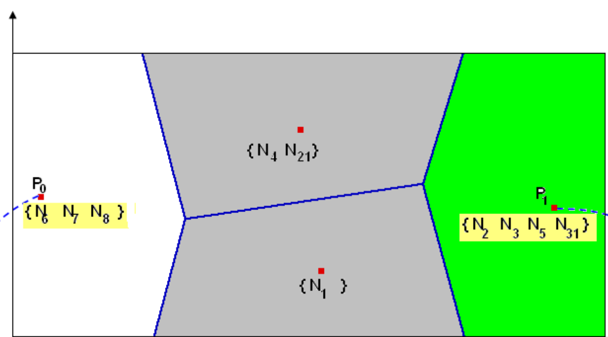
Tenuto conto della corrispondenza biunivoca tra cluster e sottoalberi, scegliamo come numero SP_{p_i} il valore $\log Num(p_i)$.

Nella figura 3.2 vediamo una rete costruita secondo la nostra architettura, con M=6 e con 42 peer inseriti. Le linee grigie tratteggiate separano le reti dei vari livelli: i livelli presenti sono uguali a $M/2$, e vanno da L_0 a L_2. Il valore della costante di soglia S è uguale a 4. Possiamo notare nella figura l'esistenza di due cluster formatisi nei siti P_0 e P_1, il primo con $Num(P_0) = 12$ e il secondo con $Num(P_1) = 25$. I peer marcati in giallo rappresentano i SP dei cluster radicati nei rispettivi siti. Le frecce tratteggiate azzurre uniscono un sito alla rete di livello inferiore radicata in quel sito e in cui sono inseriti i peer del cluster.

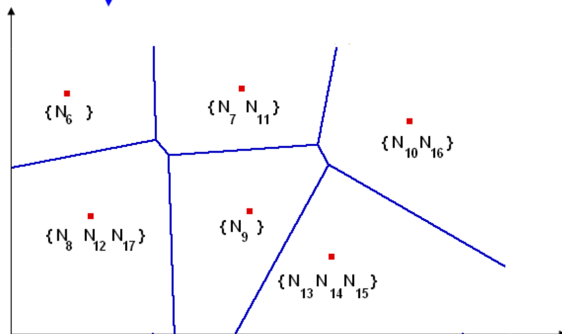
Nell'esempio sono state colorate di verde le aree di Voronoi in cui è stato mappato il peer N_2 ed in grigio le aree, nei vari livelli, costituite dai suoi vicini, per mostrare come il numero di questi sia basso.

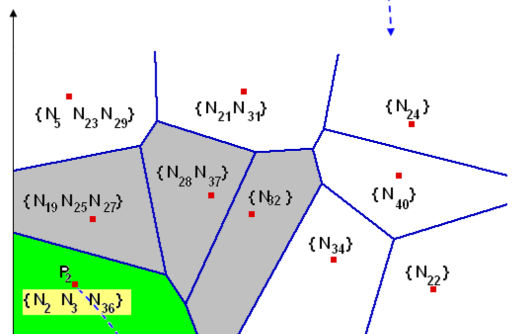
La seguente osservazione mostra come sia sempre possibile eleggere un numero di SuperPeer pari al logaritmo del cluster.

$L_0 : \{ A_0 \ A_1 \}$



$L_1 : \{ A_2 \ A_3 \}$





$L_2 : \{ A_4 \ A_5 \}$

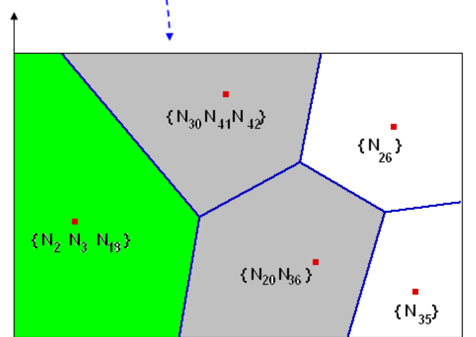


Figura 3.2: Esempio di rete di Voronoi multilivello a SuperPeer multipli

Osservazione 1 *Data una qualunque rete di Voronoi ad un certo livello L_i di etichetta E radicata nel sito p_i di livello L_{i-1}, per cui $Num(p_i) > 1$, è sempre possibile eleggere un numero di SP uguale a* $\log(Num(p_i))$ *scelti tra i peer visibili di ogni sito della rete di livello L_i di etichetta E.*

Nei prossimi paragrafi vengono illustrate le operazioni principali eseguibili sulla nostra architettura: **risoluzione delle range query multiattributo** e **inserzione di nuovi peer nella rete**. Vedremo come questi algoritmi siano un'estensione di quelli sviluppati in [15, 16], essendo l'architettura costituita da un insieme di reti di Voronoi bidimensionali. Nel primo sottoparagrafo vengono illustrate, brevemente ed ad alto livello, le strutture dati che vengono usate dai vari peer per mantenere la visione della rete globale.

3.4 Hierarchical Voraque: le operazioni

3.4.1 Le strutture dati

In questa prima sezione, analizziamo le strutture dati definite da ogni peer per mantenere la propria visione locale della rete. Essendo la nostra architettura ibrida, cioè formata da un certo insieme di livelli di reti di Voronoi bidimensionali, ognuna delle quali legata ad una determinata rete di livello superiore da un insieme di SP, vengono prima indicate le strutture necessarie a mantenere la visione di un peer su un determinato livello: queste strutture sono una stretta estensione di quelle descritte in [15, 16]. Successivamente verrà descritta la struttura dati necessaria a rappresentare i diversi livelli.

Dato un peer n_i visibile in una rete di un certo livello L_i di Hierarchical Voraque in un certo sito p_i, esso deve mantenere per il livello L_i le seguenti strutture:

1. Lista dei VN, dei CN, dei LRN e dei BLRN. In questa architettura è possibile che un sito sia costituito da un cluster di peer, quindi dato il peer n_i, ogni sito suo vicino di Voronoi può essere associato ad un insieme di peer. Quindi queste liste sono sostituite da Hash Table, dove ad ogni sito Q chiave della Hash Table, è associata la lista dei peer associati a quel sito, cioè i SP_Q se il cluster è di dimensione maggiore di S, oppure la lista dei companions definita nel paragrafo 3.3, se la dimensione del cluster è minore di S.

2. Le caratteristiche geometriche dell'area di Voronoi associata ad n_i, cioè i vertici, i lati e le coordinate di p_i.

3. La lista dei SP del livello L_i mantenuta in modo lazy, come discusso nel paragrafo 3.3.

4. la lista dei **companions**, cioè gli altri peer che fanno parte del cluster associato al sito p_i, per poter comunicare direttamente con loro. La struttura è presente solo quando il cluster è di dimensione minore di S. In caso contrario la lista dei companions viene rimpiazzata dalla lista dei SP_{p_i}.

5. Se il peer è stato eletto SP per questa rete di livello L_i, significa che esso è visibile anche nel livello superiore L_{i-1} in un sito p_j costituito da un cluster di dimensione maggiore di S. In questo caso esso deve mantenere anche le seguenti strutture dati:

   (a) La lista dei SP_{p_j}, aggiornata ogni volta che un nuovo SP ne entra a far parte. In questo modo si costruisce una ulteriore overlay network per ogni livello, costituita da una rete completamente connessa di soli SP: le sue funzioni sono quelle descritte nel paragrafo 3.3, altre verranno descritte nelle prossime sezioni.

   (b) Il numero dei peer del cluster $Num(p_j)$: questo numero viene aggiornato ogni volta che un nuovo peer entra a far parte del cluster e quindi si inserisce in una delle reti del sottoalbero corrispondente e comunicato agli altri SP_{p_j}. $Num(p_j)$ viene utilizzato per determinare quando deve essere eletto un nuovo SP, in modo da mantenere i SP in numero $\log Num(p_j)$.

Queste strutture devono essere mantenute per ogni livello in cui un certo peer risulti visibile. Un peer pubblica un unico oggetto con M attributi, quindi non può essere presente in più di una rete di Voronoi per livello perché ogni rete di livello successivo al primo è costituita da una coppia diversa del valore degli attributi del livello superiore. Il peer quindi mantiene una ulteriore struttura costituita da un vettore di livelli, da 0 a $L_{Max} = M/2$. In ogni posizione del vettore vengono memorizzate le strutture relative alla visione che ha il peer per quel livello. Se il peer non è visibile in quel livello, l'entry corrispondente risulta vuota. Inoltre, ogni peer memorizza la costante S di soglia, il raggio dei CN ed il **livello di inserimento**: quest'ultimo corrisponde al livello della rete più bassa in cui il peer è stato inserito. Ad esempio in figura 3.2 il cui di inserimento del peer N_2 corrisponde alla rete in basso a destra di secondo livello.

3.4.2 Risoluzione di Range Query multiattributo

In questa sezione descriveremo l'algoritmo di risoluzione delle range query multiattributo. Per semplificare, possiamo limitarci a query che specificano una coppia di valori Min e Max per ogni attributo, dove tutti gli attributi sono definiti. Se non è specificato alcun vincolo per un certo attributo, è possibile specificare

per quell'attributo un range che comprende tutti i valori compresi tra Min e Max. Ricordiamo che il peer n che immette la query può essere visibile ad uno o più livelli della rete. In particolare n non è necessariamente visibile al livello L_0.

Rispetto ad una rete bidimensionale costituita da due soli attributi, dove tutti i peer che fanno parte dell'AOI sono risposta alla query, in Hierarchical Voraque abbiamo invece che i valori di ogni coppia di attributi specificati nella range query, di indice $[A_{2i}, A_{2i+1}]$, definiscono una AOI **locale** ed i peer che fanno parte di questa AOI sono match soltanto per quella coppia di attributi: la soluzione della range query multiattributo corrisponde all'intersezione degli insiemi dei peer che fanno parte delle varie AOI locali.

Per sfruttare la selettività degli attributi, invece di calcolare i match appartenenti a tutte le AOI locali delle reti di Voronoi di tutti i livelli dell'albero di Hierarchical Voraque in modo parallelo e poi calcolare l'intersezione dei risultati, la risoluzione della query inizia invece dalla visita dei peer della AOI del livello L_0 e, per ognuno di questi che fa parte di un cluster di dimensione maggiore di S, l'algoritmo continua propagando la query al livello inferiore, dove si ripete la ricerca dei peer dell'AOI locale a quel livello. In questo modo, non vengono visitati i peer che fanno parte di AOI di reti appartenenti a cluster associati a siti p_i che non fanno parte dell'AOI di almeno un livello superiore: questi peer infatti sicuramente non costituirebbero match per tutta la query. Il funzionamento è molto simile a quello descritto in MAAN [7], ma nel nostro caso non possiamo scegliere di iniziare dal livello con la migliore selettività, ma sempre dal livello che corrisponde alla radice dell'albero, cioè L_0. Ricordiamo che nel paragrafo 3.1 sono indicati alcuni criteri per determinare come scegliere il mapping tra gli attributi e i livelli per ottenere un livello di selettività il più possibile vicino all'ottimo.

La figura 3.3 mostra la differenza tra l'algoritmo definito in Hierarchical Voraque e quello che calcola tutte le AOI locali in parallelo: nella figura i rettangoli rappresentano le AOI locali alle reti dei due livelli L_i e L_{i+1}. La rete in basso a sinistra contiene il cluster dei peer mappati nel sito P, mentre quella a destra il cluster dei peer mappati nel sito Q, come indicano le frecce nere tratteggiate. Nel nostro caso, la rete in basso a destra non viene visitata perché Q non fa parte dell'AOI della rete di livello superiore, mentre l'algoritmo parallelo visiterebbe i peer della sua AOI comunque.

L'algoritmo di risoluzione di una range query immessa da un peer n associato al sito p_n della rete di livello L_0 è costituito dai seguenti passi:

1. n determina se fa parte dell'AOI di livello L_0, cioè quella per gli attributi $[A_1, A_2]$ e, se il controllo è negativo, spedisce la query ad un peer associato al sito p_i più vicino al centro dell'AOI o, a scelta dell'utente, ad un peer del sito p_i più vicino ad un qualsiasi sito appartenente all'AOI, eseguendo un algoritmo di **routing greedy**.

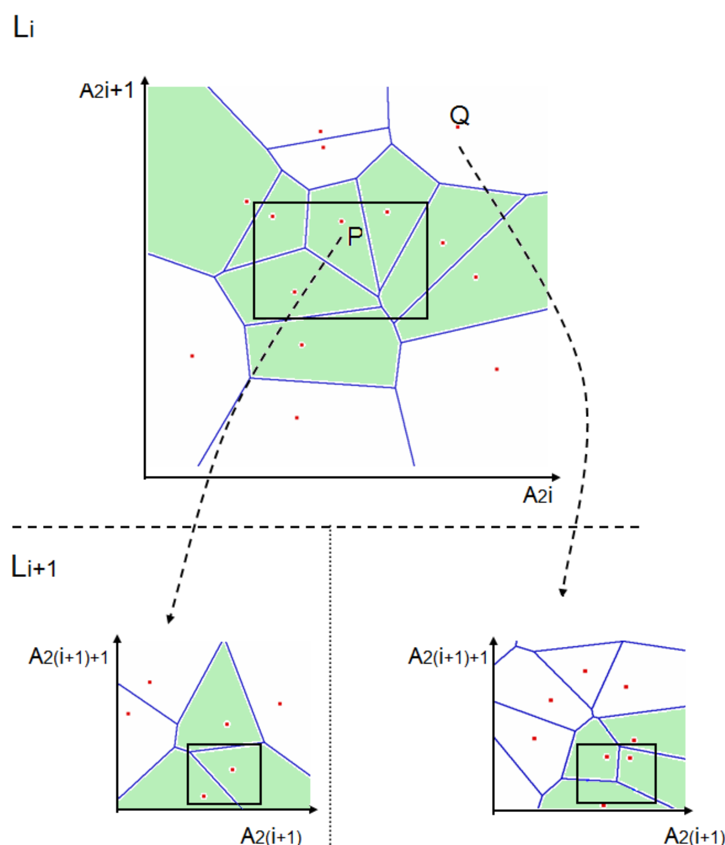


Figura 3.3: Esempio di AOI locali: differenza tra Hierarchical Voraque e l'algoritmo parallelo; la rete in basso a destra non viene visitata da Hierarchical Voraque

2. Se n fa parte dell'AOI, o il routing greedy ha raggiunto un peer n_i appartenente all'AOI, n_i o n viene selezionato come radice di un albero di multicast calcolato per mezzo dell'algoritmo di compass routing [32] e lo inizia. Il peer che inizia il compass viene chiamato n_{root}.

3. Se la AOI è interna alla regione di Voronoi associata ad n o ad n_i, viene inviata una risposta negativa al peer che ha immesso la query. Altrimenti si passa al punto 4.

4. Ogni peer n_j che riceve il messaggio di tipo compass routing (compreso n_{root}) effettua entrambi i seguenti passi:

   (a) Controlla se è SuperPeer di un cluster di dimensione maggiore di S: in caso affermativo, propaga la query al livello inferiore, reiniziando l'al-

goritmo dal punto 1 sulla coppia di attributi del nuovo livello. Gli altri SP del cluster non ricevono la query a questo livello, perché compaiono comunque a livello inferiore e verranno eventualmente raggiunti dalla ricerca nella rete di Voronoi a livello inferiore se rientrano nella corrispondente AOI locale. Se è un cluster di dimensione minore di S, propaga la query ai propri *Companions* che controllano se sono match, inviando una risposta ad n affermativa o negativa.

(b) Controlla se è risposta alla query per tutti gli attributi. In caso affermativo, si aggiunge alla lista dei peer che sono risposta alla query. Dopodiché, porta avanti il protocollo di compass routing propagando la query nello stesso livello ai propri figli nell'albero di multicast, che abbiano almeno un punto che interseca l'AOI.

Si deve tenere conto però che n può non essere visibile a livello L_0: in questo caso il peer deve prima propagare la query fino ad un peer n_i del livello L_0, in modo che n_i possa accedere ai punti dell'AOI corrispondente. La propagazione della query fino al livello L_0 avviene propagandola tramite i SP. Ricordiamo infatti che ogni peer facente parte di una rete di un qualsiasi livello L_i possiede un riferimento ai propri SP della rete di livello L_i. Questi a sua volta fanno hanno un riferimento ai SP della rete di livello L_{i-1} in cui sono visibili e così via fino al livello L_0. n_i successivamente procede con l'algoritmo sopra descritto.

Nella figura 3.4 vediamo due casi diversi possibili durante l'esecuzione dell'algoritmo di compass routing [32]: nel grafico a sinistra la regione di Voronoi asscociata al sito P interseca la AOI rappresentata dal rettangolo nero, ma non ne fa parte, mentre nel grafico a destra vediamo il caso opposto. In entrambi i grafici si assume $K < S$, con $S = 4$ e K dimensione del cluster associato al sito P; le frecce nere rappresentano una parte dell'albero di multicast e N_1 è il peer scelto dal peer Q come rappresentante del sito P per ricevere il messaggio di compass. Nel grafo a sinistra N_1 si accorge di non far parte dell'AOI e non propaga il messaggio di query ai propri *Companions* [N_2,...,N_K]; nel grafo a destra invece si accorge di far parte dell'AOI e, come vediamo nella nuvoletta, propaga il messaggio ai suoi *Companions*. In entrambi i casi N_1 propaga la query ai suoi figli nell'albero di multicast, come vediamo rappresentato dalla freccia nera tratteggiata del grafo a destra.

Infine nella figura 3.5 vediamo il caso in cui il peer N_1, come in figura 3.4, è scelto dal peer Q come rappresentante del sito P: in questo caso però il cluster è rappresentato da un numero di peer maggiore o uguale a S e K di questi sono SP: il peer N_1, dopo essersi accorto di essere interno all'AOI, propaga il messaggio di query al livello sottostante. Come possiamo vedere dalla figura, nel livello inferiore il peer ripete l'algoritmo di risoluzione locale della query con i due nuovi

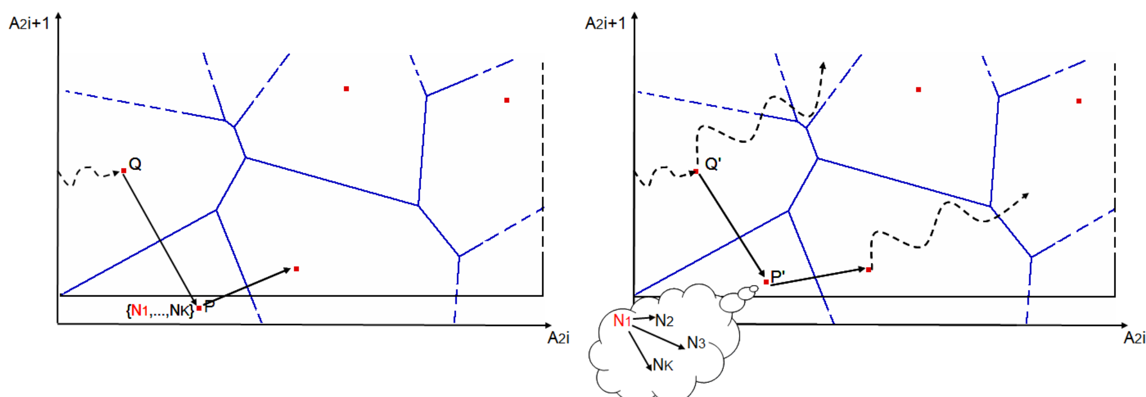


Figura 3.4: Esecuzione compass routing: casi possibili quando al sito *P* è associato un cluster di dimensione K minore di S

attributi $[A_{2i+1}, A_{2(i+1)+1}]$. In rosso le frecce della fase greedy, in nero i siti che compongono l'albero di multicast costruito dal compass routing.

Infine è importante notare come sia possibile effettuare un insieme di ulteriori ottimizzazioni, ad esempio usare un campo TTL per limitare la profondità degli algoritmi di routing compass e greedy in caso di AOI troppo grandi con un tempo di risposta troppo elevato.

L'algoritmo per la risoluzione delle query esatte (dove gli attributi sono specificati con dei valori unici) è praticamente identico a quello descritto, con la differenza che una volta raggiunto il punto di risoluzione della query, se questo cade all'interno della regione di Voronoi del sito ad esso più vicino ma non corrisponde o non rientra nel raggio di assorbimento, il peer corrispondente genera immediatamente un messaggio di risposta negativo.

Infine il peer *n* che ha immesso la query può terminare di attendere risposte alla query dopo un certo periodo di tempo, oppure dopo aver ricevuto un numero sufficiente di risposte o, nel caso di una query esatta, aver ricevuto risposta negativa o positiva.

3.4.3 Inserzione di un nuovo peer

Questo paragrafo descrive l'algoritmo di inserzione di un nuovo oggetto *O* definito da una tupla di *M* valori di attributi, per semplicità numerici; supponiamo che ogni peer pubblichi solo un oggetto. Per semplificare la trattazione, si considera che **ogni inserzione inizi al termine della precedente**; ci si può estendere al caso reale di più inserzioni contemporanee, inserendo nell'algoritmo opportuni

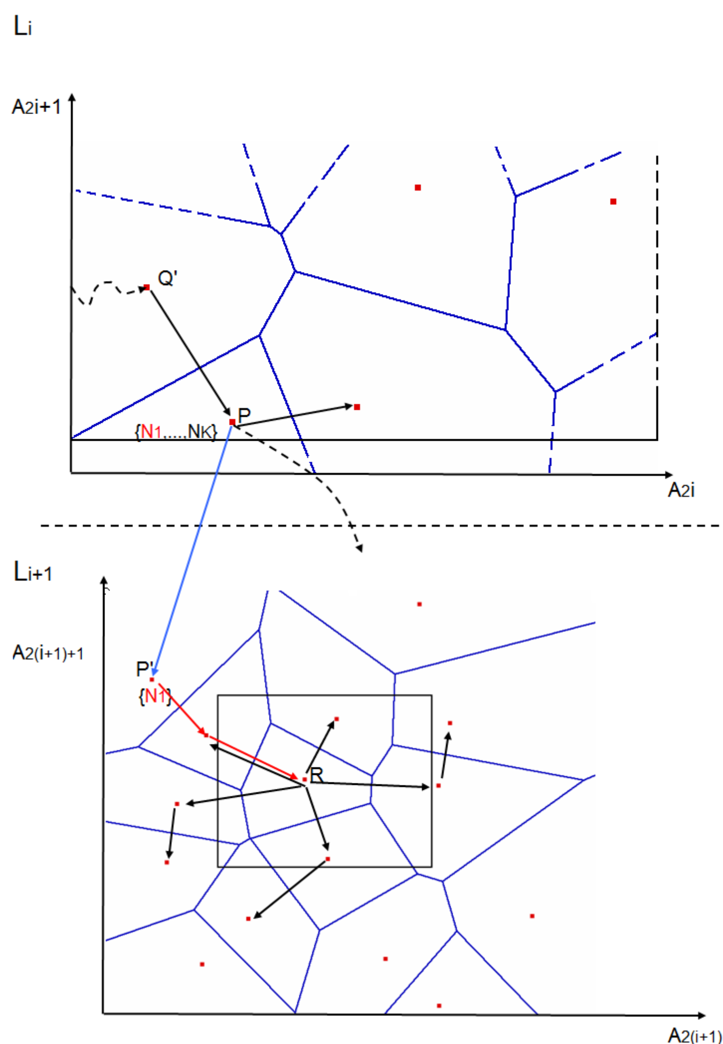


Figura 3.5: Esecuzione compass routing: caso in cui il sito P costituisce un cluster di dimensione maggiore o uguale a S

protocolli di sincronizzazione tra i peer per evitare inconsistenze nella visione della rete.

Ricordiamo che il livello di inserzione di un peer n_i è il livello più basso in cui viene inserito e che ogni peer deve conoscere almeno i SP del proprio livello di inserimento. Questa conoscenza è sufficiente perché i propri SP conoscono ricorsivamente i SP del livello superiore. Se un peer viene eletto SP, la sua conoscenza viene aumentata con i SP del livello superiore.

Ricerca del punto di inserzione

Supponiamo che il peer n_i si inserisca nella rete nel punto p_i.

Per come è costituita una rete Hierarchical Voraque, l'inserzione deve iniziare da un peer visibile nella rete di livello L_0: questo peer, che deve essere conosciuto a priori, viene detto **peer di bootstrap**, da ora in poi chiamato n_{boot}. Il peer n_i inizia inviando un messaggio di tipo Insert a questo peer; il messaggio è costituito dai seguenti campi:

- *Livello*: indica il livello di Hierarchical Voraque a cui è destinato il messaggio. É necessario perché il peer destinatario può essere visibile in più di un livello.
- ID_{n_i}: indicatore univoco del peer n_i, occorre per poter inviare ad n_i il messaggio di inizializzazione necessario a renderlo visibile nella rete, come indicato più avanti.
- *Valore attributi*: la tupla dei valori degli M attributi dell'oggetto che si vuole pubblicare.
- ID_{Greedy}: identificatore univoco del peer che inizia l'algoritmo greedy per un livello L_i diverso da L_0. Si può osservare che questo è sempre un SP del livello L_i. Viene utilizzato dal peer n_i per ottenere la lista dei SP e dal peer gestore per l'elezione, come verrà descritto più avanti.

Lo pseudocodice presentato nell'algoritmo 1 riassume i passi eseguiti da n_i in questa fase dell'algoritmo di inserzione.

Algorithm 1: Procedura di inizio dell'algoritmo di inserzione

Input: la tupla T dei valori degli M attributi ed il peer n_i

```
1 begin
2     ID_boot = SearchNboot();
3     Acquisizione del proprio indice ID_{n_i} calcolato in modo univoco da
      ID_boot;
4     InsertMSG = CreateInsertMSG(0, ID_{n_i}, T, null);
5     Invio InsertMSG al peer ID_boot;
6     Attesa messaggio di inizializzazione InitMSG;
7     ElaborateMSG(InitMSG);
8 end
```

Considerata la rete di Voronoi di livello L_0, ma si può estendere in maniera simile alla rete di un qualsiasi livello L_i di Hierarchical Voraque, il problema si

riduce alla ricerca del punto p_i di pubblicazione dell'oggetto O, determinato dal valore degli attributi (A_0,A_1) dell'oggetto stesso. Si procede analogamente per gli attributi (A_{2i},A_{2i+1}) per un qualsiasi livello L_i.

Ogni livello è costituito da una o più reti di Voronoi bidimensionali, quindi possiamo estendere l'algoritmo di inserzione descritto in Voraque [15, 16]. Il primo passo di questo algoritmo consiste nel raggiungere il sito p_j nella cui regione di Voronoi cade il punto p_i: in Voraque, viene usato un algoritmo di **routing greedy** per risolvere questo problema. Descrivo i passi di questo algoritmo, insieme alle modifiche introdotte in Hierarchical Voraque, per un peer n_j che riceve, dopo un certo numero di passi, il messaggio di Insert propagato da n_{boot}; ricordiamo che stiamo considerando il livello L_0:

1. n_j controlla il livello del messaggio ed estrae dalla tupla T dei valori degli attributi quelli del livello in questione, in questo caso (A_0,A_1), per ottenere le coordinate del punto p_i.

2. n_j controlla se p_i cade nella sua stessa area di Voronoi: in questo caso, il routing greedy termina e n_j diventa il **peer gestore** (NG_{p_i}) per il passo successivo dell'algoritmo di inserzione.

3. se il punto 2 è falso, n_j calcola la distanza euclidea tra ogni sito facente parte degli insiemi dei suoi VN, CN e LRN, e il punto p_i.

4. n_j sceglie il sito p_k con distanza euclidea minima e gli inoltra il messaggio di Insert. In questo caso possono verificarsi due diversi scenari:

   - al sito p_k è associato un cluster di peer.
   - al sito p_k è associato un solo peer.

   Nel primo caso, è ininfluente quale peer scegliere tra quelli visibili associati al cluster per ottenere un algoritmo di routing corretto, per cui n_j ne sceglie uno in maniera casuale: in questo modo si cerca di bilanciare statisticamente il carico tra i peer visibili del cluster evitando inoltre di implementare un algoritmo di bilanciamento del carico esplicito tra i SP_{p_k}. Nel secondo caso, il peer scelto è unico.

La figura 3.6 mostra un esempio in cui due peer mappati rispettivamente in P_1 e P_2 stanno eseguendo in parallelo due routing greedy, relativi a due diversi inserimenti con target rispettivamente P e Q. Il primo routing è rappresentato dal percorso con frecce blu, l'altro con frecce verdi. Si può osservare come, in caso di inserimenti simultanei che passano attraverso lo stesso sito (in questo caso P_3), la presenza di un cluster associato a P_3 permette di ottenere una maggiore

scalabilità: i due peer infatti scelgono rispettivamente i peer colorati di blu e di verde, ottenendo anche una migliore distribuzione del carico fra i peer del cluster associato a P_3.

Li

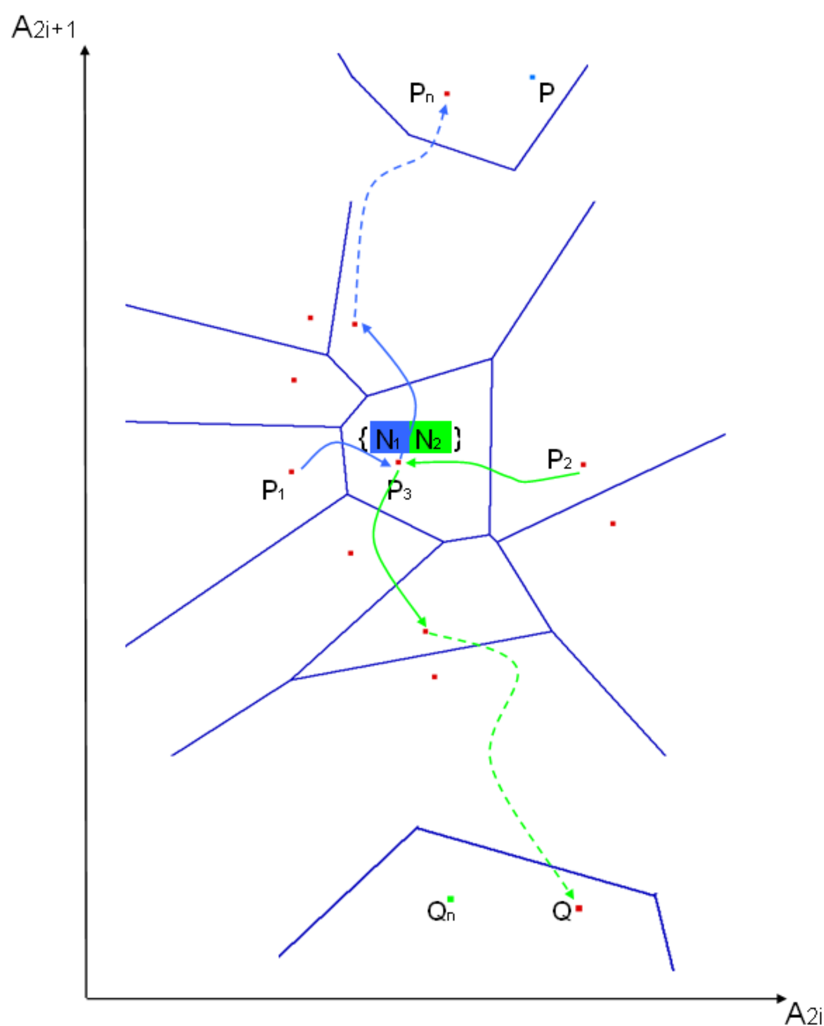


Figura 3.6: Esempio esecuzione di due algoritmi di routing in cui due punti devono inoltrare il messaggio ad uno stesso punto: caso di ripartizione del carico

La figura 3.7 mostra un esempio di inserimento: il punto P è il target. Il routing inizia da N_{boot} che inoltra il messaggio di inserimento fino a quando non viene ricevuto dal peer N_1 mappato in P_{N_1}. La freccia tratteggiata rappresenta il cammino del routing; nella figura si vede che N_1 calcola come sito più vicino al target P

il sito P_2 appertenente ai suoi Close Neighbours, costituito da un cluster: in questo caso, il peer N_6 colorato su sfondo grigio viene scelto in modo casuale per proseguire l'algoritmo fino al peer gestore sito in P_1.

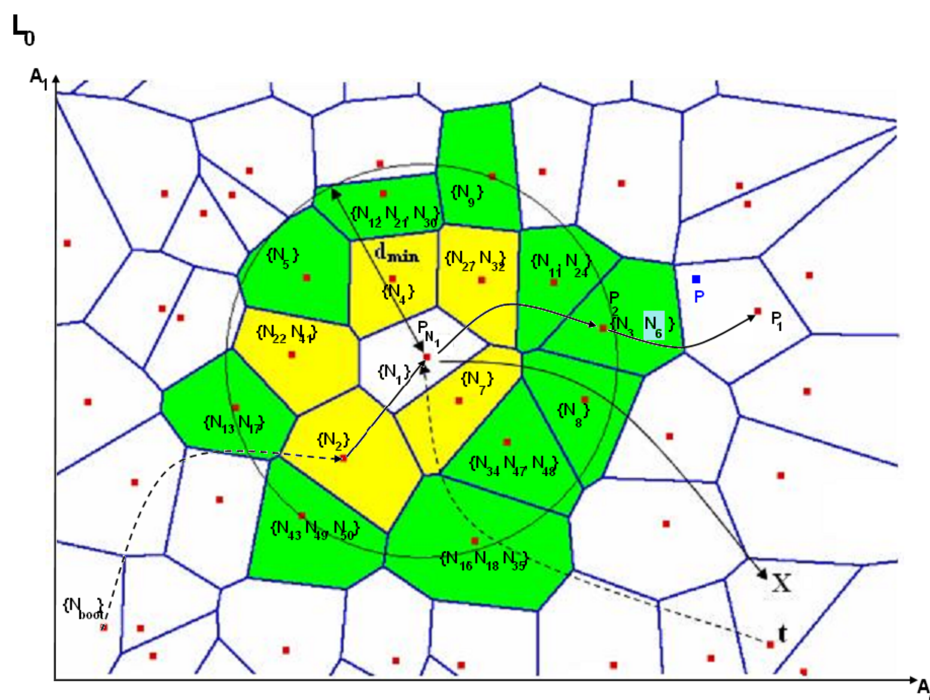


Figura 3.7: Esempio di esecuzione dell' algoritmo di routing da parte del peer N_1; in giallo ed in verde i suoi VN e CN

Il routing greedy termina quando il messaggio di Insert raggiunge il peer NG_{p_i}, mappato in un sito p_{NG}.

Inserzione nel punto target

A questo punto inizia il secondo passo consistente nell'inserzione del peer nel punto p_i considerato. In [15, 16] questo passo è diverso da Hierarchical Voraque, in quanto non esistono cluster e quindi il nuovo peer genera sempre una nuova area di Voronoi. In Hierarchical Voraque invece, definito un **raggio di assorbimento**, possono accadere due casi:

1. Il punto p_i non cade nel raggio di assorbimento di p_{NG}: dobbiamo generare una nuova regione di Voronoi, come in Voraque.

2. Il punto p_i cade nel raggio di assorbimento di p_{NG}: in questo caso le coordinate del peer vengono modificate come se il target fosse esattamente p_{NG} ed il peer entra a far parte o crea un nuovo cluster di peer in p_{NG}.

Prendiamo in considerazione il primo caso, il più complesso: la generazione di una nuova area di Voronoi. Premettiamo che ogni peer, compreso il peer NG_{p_i}, ha solo una visione parziale della rete, e possiede un proprio grafo di Voronoi costruito per mezzo dell'algoritmo di Fortune [35] a partire dai siti che fanno parte dei suoi vicini VN e CN. Non avendo il nuovo peer n_i che si vuole inserire fino a questo momento nessuna conoscenza sulla topologia della rete, i suoi insiemi di vicini VN e CN sono calcolati da NG_{p_i}. Occorre però notare che anche il NG_{p_i} possiede una visione parziale della rete, per cui NG_{p_i} deve ispezionare la rete per raggiungere i vicini di n_i a esso non visibili. Il peer NG_{p_i} ricava il grafo di Voronoi aggiornato mediante l'inserimento del nuovo sito p_i. A partire da questo grafo modificato ricava quali tra i vicini sono anche vicini di p_i ed invia ai peer visibili in questi siti un messaggio di modifica del loro grafo. I peer vicini applicano ricorsivamente questo algoritmo per ricercare a loro volta quali tra i loro vicini sono vicini di p_i ed inviano questa lista a NG_{p_i}. Poiché l'algoritmo di inserimento è molto complesso, lo spieghiamo mediante un esempio, rimandando al capitolo 4 per una sua specifica dettagliata mediante pseudocodice.

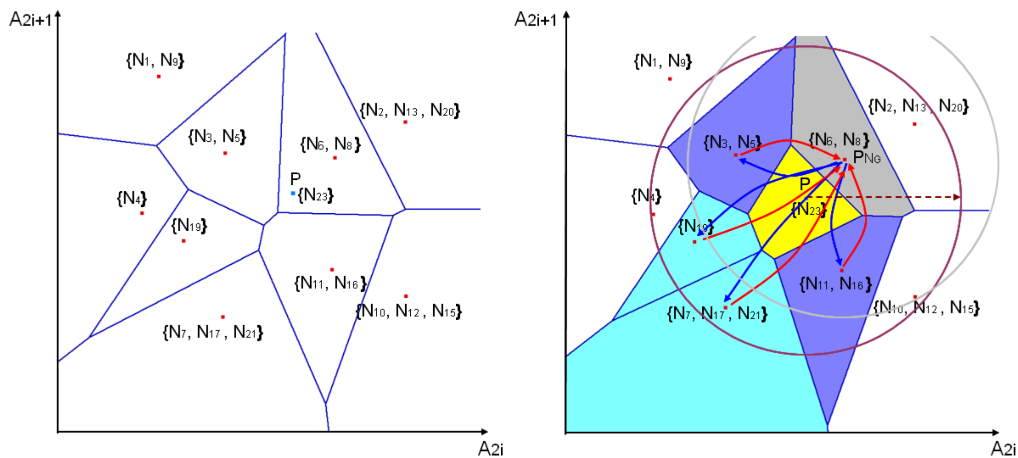


Figura 3.8: Inserimento di un nuovo peer: in blu i vicini contattati direttamente da P_{NG}, in azzurro i peer scoperti grazie ai vicini

Nella figura 3.8 possiamo vedere alcune fasi dell'esecuzione dell'algoritmo di costruzione dell'area di Voronoi per il peer N_{23} che deve essere inserito nel punto

P. N_6 inizia l'algoritmo di ispezione della rete in quanto NG del punto P, che cade nella sua area di Voronoi (vedere grafo a sinistra). N_{23} è il nuovo peer da inserire. Alla prima iterazione, viene calcolata la lista $VN_{N_{23}} = (N_6, N_8, N_3, N_5, N_{11}, N_{16})$ dove gli ultimi quattro vengono scoperti direttamente da N_6 perché sono nella sua lista di vicini e contattati tramite messaggi di modifica (le frecce blu). Grazie a questi messaggi i nodi contattati aggiornano il proprio grafo di Voronoi inserendoci il sito P. Le frecce rosse rappresentano le risposte dei peer contattati, e da esse il peer N_6 aggiunge alla lista $VN_{N_{23}}$ i nuovi peer delle aree colorate di azzurro, cioè i peer N_{19}, N_7, N_{17} e N_{21} ottenendo la nuova lista $VN_{N_{23}} = (N_6, N_8, N_3, N_5, N_{11}, N_{16}, N_{19}, N_7, N_{17})$. All'iterazione successiva vengono contattati i nuovi peer associati alle aree colorate di azzurro scoperti al termine dell'iterazione precedente che rispondono con liste di peer che sono già presenti nella lista $VN_{N_{23}}$, quindi l'algoritmo termina e l'insieme dei $VN_{N_{23}}$ viene inviato da N_6 a N_{23}.

Prendiamo adesso in considerazione il secondo caso: il punto p_i cade nel raggio di assorbimento di p_{NG} ed il nuovo peer entra a far parte o crea un nuovo cluster di peer in p_{NG}. Supponiamo di essere ad un determinato livello L_i, non obbligatoriamente L_0, e che S sia la soglia al di sopra della quale viene creata una nuova rete di Voronoi di livello L_{i+1}. Osserviamo innanzitutto che il diagramma di Voronoi, dal punto di vista geometrico, in questo caso non varia. Consideriamo le operazioni eseguite da NG_{p_i} per questo caso, per l'inserzione del peer n_i. Per prima cosa viene incrementata la dimensione del cluster di p_{NG} in seguito all'entrata del nuovo peer n_i. A seconda della nuova dimensione del cluster di p_{NG}, ci si può ricondurre ai seguenti tre casi:

- Se $Num(p_i) < S$ non viene creato un nuovo livello ma il nuovo peer n_i entra a far parte dei *Companions* di NG_{p_i}.
- Se $Num(p_i) = S$ si deve creare una nuova rete di Voronoi di livello inferiore, in cui vengono mappati tutti i peer del cluster sito in p_{NG}.
- Se $Num(p_i) > S$ si propaga il messaggio al livello sottostante.

Analizziamo i diversi casi più in dettaglio:

1. $Num(p_i) < S$, oppure siamo in una rete di livello L_{Max}. In questo caso non viene creato una nuova rete di livello L_{i+1}, ma n_i entra a far parte dei *Companions* di NG_{p_i}, cambiando il valore dei suoi attributi (A_{2i},A_{2i+1}) in modo da farli coincidere con quelli del sito p_{NG}. n_i ed NG_{p_i} hanno quindi la stessa visione della rete di livello L_i: NG_{p_i} invia a n_i i suoi 4 insiemi dei vicini di Voronoi, più l'insieme dei *Companions* in cui è compreso anche sé stesso affinché n_i li memorizzi. Successivamente comunica a tutti i

peer vicini di Voronoi e agli altri *Companions* l'esistenza del nuovo peer n_i affinché anch'essi aggiornino la loro visione della rete.

In figura 3.9 possiamo vedere un esempio, con valore di soglia $S = 4$: deve essere inserito il peer N_6, con target P' che cade nel raggio di assorbimento di P. N_1 viene scelto come peer gestore per l'inserimento: esso, quando riceve il messaggio di Insert per il nuovo peer, si accorge che la soglia non viene superata e non genera il nuovo livello di rete. Per semplificare, sono stati inseriti nel grafo soltanto i $VN_{NG_{P'}}$, e sono indicati sotto ad ognuno di essi una parte dei peer del loro insieme dei VN. Nel grafo a destra si vede, dopo l'esecuzione dell'algoritmo, come sia i vicini che i *Companions* di N_1 siano stati contattati tramite opportuni messaggi (le frecce blu) e abbiano aggiornato la loro visione della rete.

Li

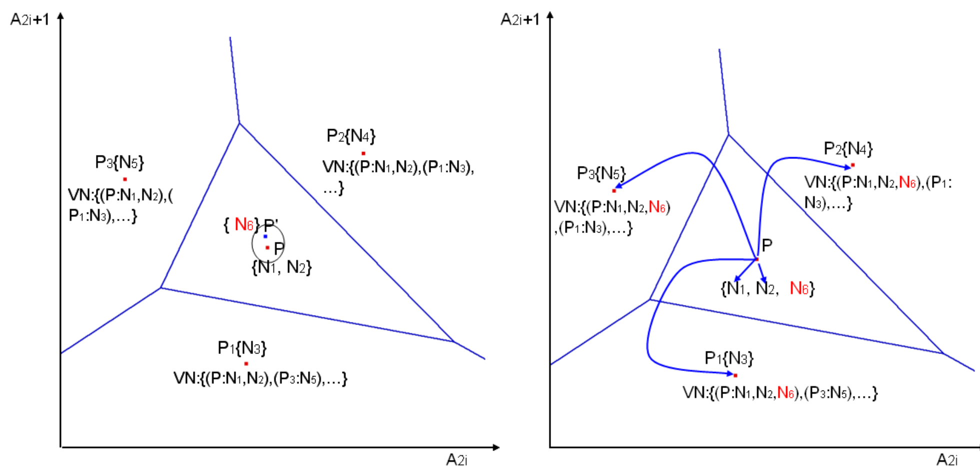


Figura 3.9: Inserimento di un nuovo peer: caso in cui il target cade nel raggio di assorbimento di P e $Num_P < S$

2. $Num(p_i) = S$. In questo caso si deve creare una nuova rete di Voronoi di livello inferiore, su cui NG_{p_i} mappa i peer del cluster di p_i in base al valore dei due attributi $(A_{2(i+1)}, A_{2(i+1)+1})$. Se siamo al livello L_{Max} invece, NG_{p_i} ripete l'algoritmo descritto nel punto precedente.

   Il peer NG_{p_i} funziona da peer di bootstrap per inserire tutti gli altri peer del cluster di p_i nella nuova rete, compreso il nuovo peer n_i. NG_{p_i} esegue quindi i seguenti passi:

- aggiorna le proprie strutture dati per tener traccia del nuovo livello e si inserisce nel punto target determinato dal valore dei suoi attributi del nuovo livello.
- inserisce in una apposita lista i peer del cluster, escludendo sé stesso poiché si è già inserito come primo peer della nuova rete.
- NG_{p_i} crea, per ogni peer della lista, un nuovo messaggio di tipo Insert. Questi messaggi hanno la particolarità di avere come valore di livello L_{i+1}, quindi funzionano come messaggi necessari a inserire i peer del cluster nella loro giusta posizione nel nuovo livello della rete. Successivamente NG_{p_i}, funzionando da peer di bootstrap, esegue nuovamente dall'inizio l'intero algoritmo di inserzione per ogni messaggio di Insert della lista e attende le risposte di inserimento dei peer.

Si può notare come i peer del cluster di p_i restino visibili nella rete di livello superiore, escluso il nuovo peer n_i. Dopo aver creato il nuovo livello, soltanto un numero corrispondente a $log(Num(p_i))$ punti del cluster dovrebbero essere eletti SP_{p_i} e quindi visibili in quella rete: esattamente dovrebbero essere $log(S)$. In questo caso, ne ho già visibili al livello superiore esattamente $S - 1 > log(S)$: per minimizzare il numero di messaggi da mandare nella rete di livello superiore, in Hierarchical Voraque tutti i peer già visibili del cluster nella rete di livello superiore vengono automaticamente eletti SP_{p_i}. Se S è sufficientemente piccola, allora $S - 1 \approx \log S$ e quindi il numero dei SuperPeer non è troppo elevato. Ottengo in questo modo due vantaggi: non si deve inoltrare nessun messaggio ai vicini di Voronoi del punto p_i nella rete di livello superiore perché non ci sono cambiamenti nei peer visibili a quel livello; in caso di nuovi inserimenti o di cancellazioni nel cluster, non devo eleggere subito nuovi SP_{p_i} o revocarne alcuni, soprattutto se ho un alto churn rate: la perdita di efficienza in un cluster di dimensione così piccola sarebbe superiore a quella che ottengo lasciando che il numero dei vicini visibili al livello superiore sia leggermente superiore al caso ottimo. Il nuovo peer n_i non viene eletto SP_{p_i} in quanto non necessario.

In figura 3.10 possiamo vedere un esempio, con valore di soglia $S = 4$: deve essere inserito il peer N_7, con target P' che cade nel raggio di assorbimento di P; N_2 viene scelto come peer gestore per l'inserimento. Nel grafo a sinistra abbiamo la situazione precedente all'inserzione, in cui il cluster ha dimensione 3. N_2, dopo aver ricevuto il messaggio di Insert, si accorge che ha un numero di $Companions = 3$ e che con il nuovo peer diventano 4, cioè uguale al valore di soglia S: esso quindi procede a generare un nuovo livello. Nel grafo a destra, è stata colorata di blu la posizione di N_2 nella nuova rete e di rosso quella del nuovo peer N_7. Inoltre si può osservare

come i peer N_2, N_1 e N_6 rimangono come SuperPeer nella rete di livello L_i e questo consente di evitare l'inoltro di ulteriori messaggi ai vicini.

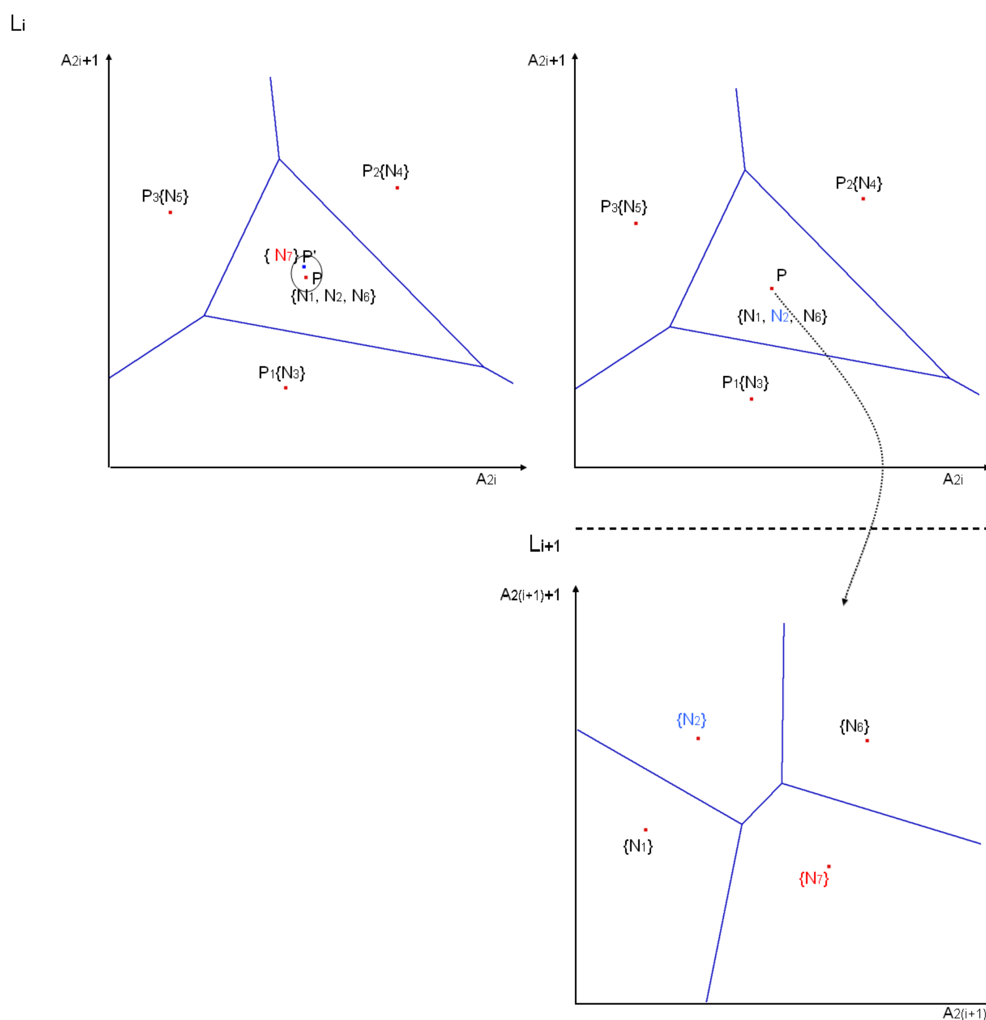


Figura 3.10: Inserimento di un nuovo peer: caso in cui il target cade nel raggio di assorbimento di P e $Num_P = S$

3. $Num(p_i) > S$. In questo caso esiste già la rete di livello inferiore. NG_{p_i} propaga il messaggio di Insert nella rete di livello inferiore, aggiornando il livello a L_{i+1} e iniziando un nuovo algoritmo di inserzione ripartendo dal routing greedy nella nuova rete. Il caso è ricorsivo: questo passo dell'algoritmo di inserzione termina quando o si individua un cluster di dimensione $\leq S$, oppure si genera una nuova area di Voronoi. In figura 3.11 e 3.12 possiamo vedere un esempio, con valore di soglia $S = 4$: si può vedere in

3.11 che deve essere inserito il peer N_{26}, con target P' che cade nel raggio di assorbimento di P; N_3, colorato di rosso viene scelto come peer Gestore per l'inserimento. La freccia verde indica il percorso dell'algoritmo greedy fino a P; il livello successivo è disegnato solo in modo incompleto: sono mostrate solo le posizioni dei SP. Si suppone che il cluster abbia dimensione compresa tra 16 e 23, per non ricadere in una elezione successivamente all'inserimento di N_{26}. In 3.12 si vede che N_3, dopo aver ricevuto il messaggio di Insert, si accorge che esiste il livello sottostante L_{i+1} e che lui ne è un SP: provvede quindi a inoltrare il messaggio a questo livello. In verde è indicato il nuovo cammino di routing greedy generato da N_3 nel nuovo livello. Se siamo al livello L_{Max} invece, NG_{p_i} ripete l'algoritmo descritto nel punto 1.

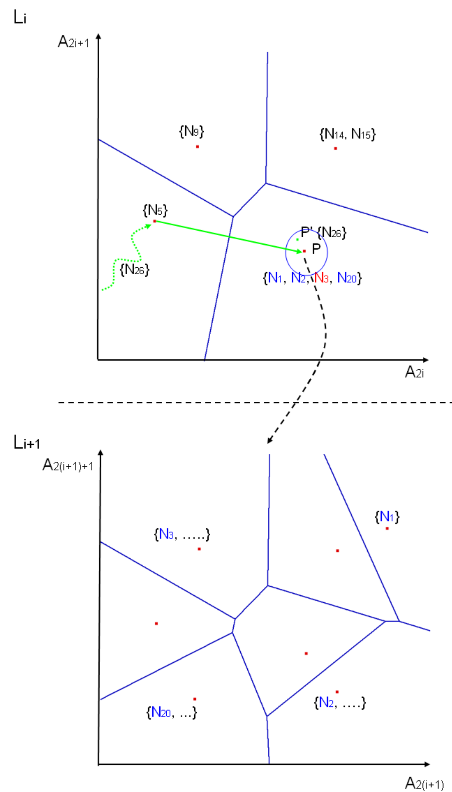


Figura 3.11: Inserimento di un nuovo peer: caso in cui il target cade nel raggio di assorbimento di P e $Num_P > S$

Alla fine di questa fase, il peer n_i è stato inserito in Hierarchical Voraque in un punto p_i ad un livello L_i, definito come **livello di inserimento** nel paragrafo 3.4.1 (d'ora in poi InsertLevel). n_i non ha però terminato la sua inizializzazione: per

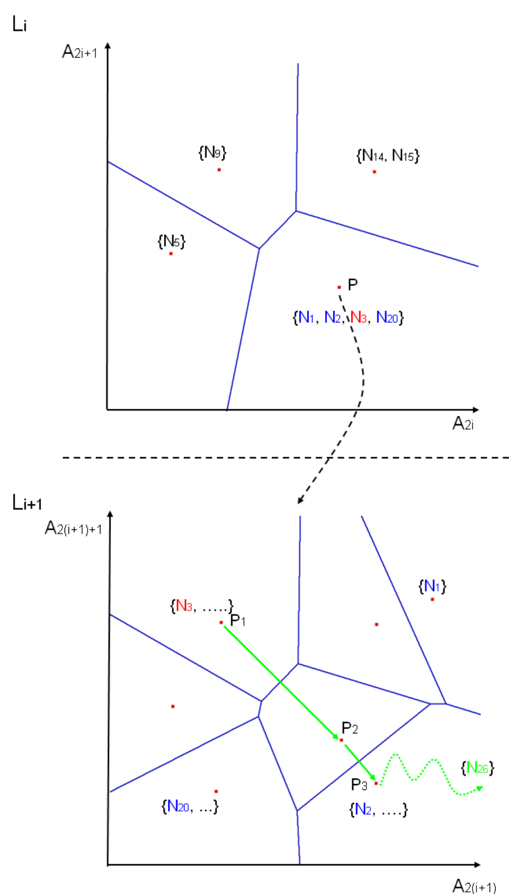


Figura 3.12: Inserimento di un nuovo peer: prosecuzione della figura 3.11

possedere la visione completa del livello, gli occorre la lista dei SP del livello di inserimento, a meno che il livello di inserimento non risulti uaguale a L_0. L'ultimo campo del messaggio di tipo Insert è utilizzato appunto per questo: per ogni livello escluso L_0, il peer di bootstrap è sicuramente un SP. n_i lo contatta prima di terminare l'inizializzazione per avere la lista dei SP, dopodiché risponde al NG_{p_i} che ha terminato l'inizializzazione.

3.4.4 Elezione di SuperPeer

Ultima fase dell'algoritmo di inserzione è quella relativa all'**elezione** di nuovi SuperPeer: come discusso nel paragrafo 3.1, in Hierarchical Voraque il numero dei SuperPeer di un qualsiasi cluster associato ad un sito P è $\log Num(P)$ (se si implementa l'ottimizzazione inserita nel caso di creazione di una nuova rete è $Num(SP_P) \geq \log(Num(P))$. Il nuovo peer n, se non inserito al livello L_0, è

sicuramente entrato a far parte di un cluster con dimensione maggiore di S per ogni livello superiore a quello di inserzione e ne ha incrementato la dimensione di 1. Per ognuno di questi livelli occorre quindi controllare se la condizione sul numero dei SuperPeer non sia violata e, in caso affermativo, eleggere un nuovo SuperPeer.

Le figure 3.13 e 3.14 mostrano un esempio che spiega il perché dobbiamo effettuare il controllo ad ogni livello: nella figura 3.13 vediamo l'inserimento del nuovo peer $N32$ nella rete in basso a destra del livello L_2. Le frecce verdi rappresentano il routing greedy eseguito per giungere al sito P_0, dove viene selezionato il nodo N_3 per propagare l'inserzione al livello L_2. N_3 diventa il nodo $n_{greedy_{L_1}}$ e anche $n_{greedy_{L_2}}$ perché è SuperPeer anche della rete in basso a destra di livello L_2. Come possiamo vedere, esso incrementa il valore $Num(P_0)$ di 1 e anche il valore $Num(P_2)$ di 1 in seguito all'entrata di N_{23} nei rispettivi cluster, che raggiungono la dimensione rispettivamente di 32 e 20, mentre la rete in basso a sinistra non varia dimensione. Nelle reti di livello 3 non vengono mostrati tutti i peer, ma solo i SuperPeer. Nella figura 3.14 vediamo cosa succede quando N_3 deve controllare se effettuare un'elezione nella rete in basso a destra: $Num(SP_{P_2}) = 4$ e $\log(Num(P_2)) = 4, \ldots$ quindi la condizione di elezione non è violata e non elegge un nuovo SuperPeer, come si può vedere dal fatto che non sono variati di numero i peer mappati nel punto P_2. Quando N_3 controlla se deve effettuare un'elezione nella rete al livello L_1, scopre che la condizione viene violata ed elegge un nuovo SuperPeer, in questo caso il nodo N_{18} colorato di nero.

Questo succede perché la dimensione del cluster di P_0 è data dalla somma delle dimensioni dei cluster associati a P_2 e P_1, non soltanto dalla dimensione del cluster associato a P_2.

Affinché esista un peer eleggibile in un certo livello, è necessario che tutti i cluster associati ai siti di quel livello abbiano già un numero di SuperPeer che rispetti la condizione precedente; ciò significa che l'algoritmo di controllo ed elezione deve essere fatto **bottom-up**, partendo dal livello di inserzione del nuovo peer n.

Ricordiamo che il messaggio di Insert relativo ad un nuovo peer n viene propagato, al livello L_i, da un SuperPeer che chiamiamo n_{greedy_i} fino al peer NG_{p_i} gestore del punto corrispondente alle coordinate degli attributi di n al livello L_i e che contiene un campo con l'identificatore di n_{greedy_i}. NG_{p_i} riceve un messaggio di notifica della avvenuta inserzione da n e propaga questo messaggio a n_{greedy_i}. n_{greedy_i} è il peer che esegue il controllo sul numero dei peer del cluster del sito p_{i-1} in cui n_{greedy_i} è mappato al livello L_{i-1} e decide se occorre eleggere un nuovo SuperPeer per il livello L_i. In caso affermativo, procede alla elezione. In ogni caso, n_{greedy_i} propaga il messaggio di notifica di avvenuta inserzione a $n_{greedy_{i-1}}$ nella rete di livello superiore, il quale procede nuovamente al controllo e alla eventuale elezione. L'algoritmo termina quando si giunge al peer n_{greedy_1}, in quanto nella rete al livello L_0 non dobbiamo procedere a nessuna elezione essendo il primo livello.

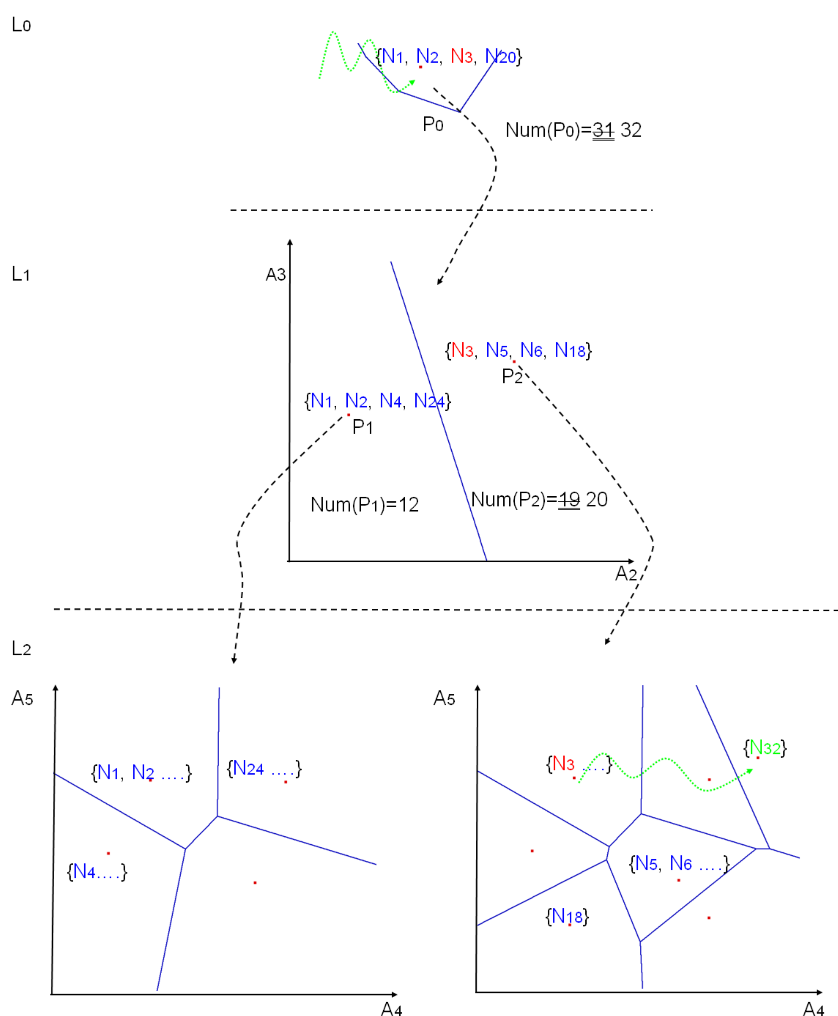


Figura 3.13: Caso in cui non è necessario eleggere un nuovo SuperPeer al livello di inserimento ma lo è ad un livello superiore: prima parte

La procedura di elezione di un nuovo SP in una rete al livello L_i, radicata in un sito p_{i-1} di una rete di livello L_{i-1} diverso da L_0, risulta necessaria in uno dei due seguenti casi:

- un nuovo peer è entrato nel cluster radicato nel sito p_{i-1} di livello L_{i-1} .
- un SP è uscito dalla rete e quel peer era necessario per mantenere il numero totale di SP a $\log Num(P)$.

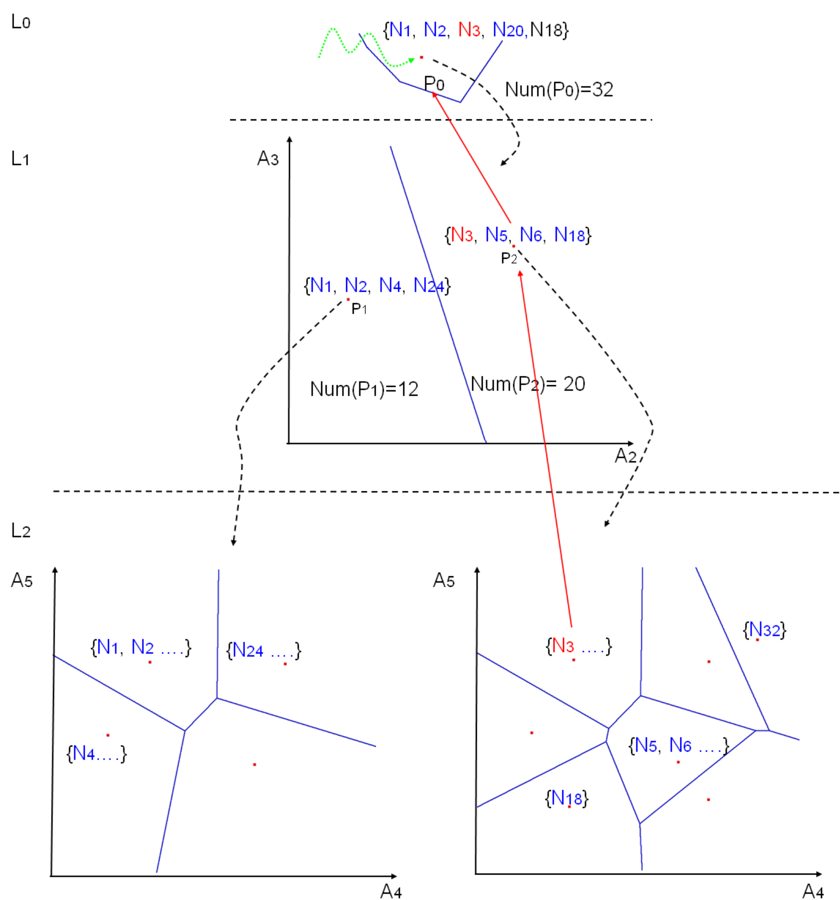


Figura 3.14: Caso in cui non è necessario eleggere un nuovo SuperPeer al livello di inserimento ma lo è ad un livello superiore

La politica di elezione in Hierarchical Voraque è molto semplice: il SuperPeer che procede all'elezione sceglie un **qualsiasi** peer visibile nella rete tra l'insieme dei peer del cluster suddetto. In alternativa, il SuperPeer può applicare una politica che tenga conto soltanto dello stato interno dei peer.

L'algoritmo si svolge in due fasi:

- ricerca di un peer da eleggere, la cui esistenza è garantita dall'Osservazione 1.

- elezione del peer

Ricerca del peer da eleggere

Descriviamo la procedura di ricerca eseguita da un peer n_i di una rete di livello L_i con etichetta E, radicata in un sito p_{i-1} di una rete di livello L_{i-1} diverso da L_0 tale che n_i faccia parte dei $SP_{p_{i-1}}$. Se l'elezione capita a causa dell'entrata di un nuovo peer n, se n è appena stato inserito nella rete E oppure è stato eletto SuperPeer da una delle reti di livello inferiore radicate in un sito della rete E, allora n è entrato a far parte dei peer visibili della rete E e n_i lo elegge direttamente SuperPeer: in questo modo non occorre eseguire un algoritmo di ricerca del peer da eleggere, risparmiando messaggi per questa fase dell'elezione. Si noti che l'identità del nuovo peer n da eleggere può essere comunicata ad n_i mediante il messaggio di notifica di avvenuta inserzione.

Nel caso in cui n non è stato inserito nella rete E, oppure non è stato eletto nessun SuperPeer da una delle reti di livello inferiore radicate in un sito della rete E oppure un peer è uscito dalla rete E ed è stata violata la condizione sul numero dei SP, allora n_i deve ricercare un peer n_j da eleggere eseguendo un **algoritmo di routing distribuito**. L'algoritmo sfrutta l'enunciato dell'Osservazione 1: la garanzia dell'esistenza di un peer visibile che non faccia parte dell'insieme dei SuperPeer. Da questo si può dedurre che, se esistono uno o più peer eleggibili, almeno uno di essi è un vicino di Voronoi di uno dei SuperPeer (non è sempre vero l'opposto). Il peer n_i esegue quindi la seguente procedura:

- Verifica se, tra tutti i suoi vicini di Voronoi ci sia un peer che non sia tra i $SP_{p_{i-1}}$. In questo caso lo elegge immediatamente SuperPeer. La figura 3.15 mostra un esempio di questo caso: il peer N_1 è un SuperPeer della rete di livello L_{i+1}. Supponendo che sia entrato il 32-esimo peer nel cluster associato al sito P', il peer N_1 ricerca se tra i suoi vicini (in verde le aree dei VN e CN, in blu gli LRN) c'è un peer che non è stato eletto SuperPeer: in questo caso N_1 può scegliere tra N_5, N_8 o N_9.

  Si noti che se un cluster ha dimensione minore di S, si controlla tutto l'insieme dei peer che lo compongono.

- Se tutti i vicini sono già SuperPeer, si invia un **messaggio in broadcast** a tutti i SuperPeer per richiedere se tra i loro vicini di Voronoi esiste un peer che possa essere eletto. Tutti i SuperPeer che ricevono il messaggio controllano se nel loro insieme dei vicini esiste almeno un peer eleggibile, il suo identificatore viene inviato a n_i, altrimenti rispondono con un messaggio vuoto.

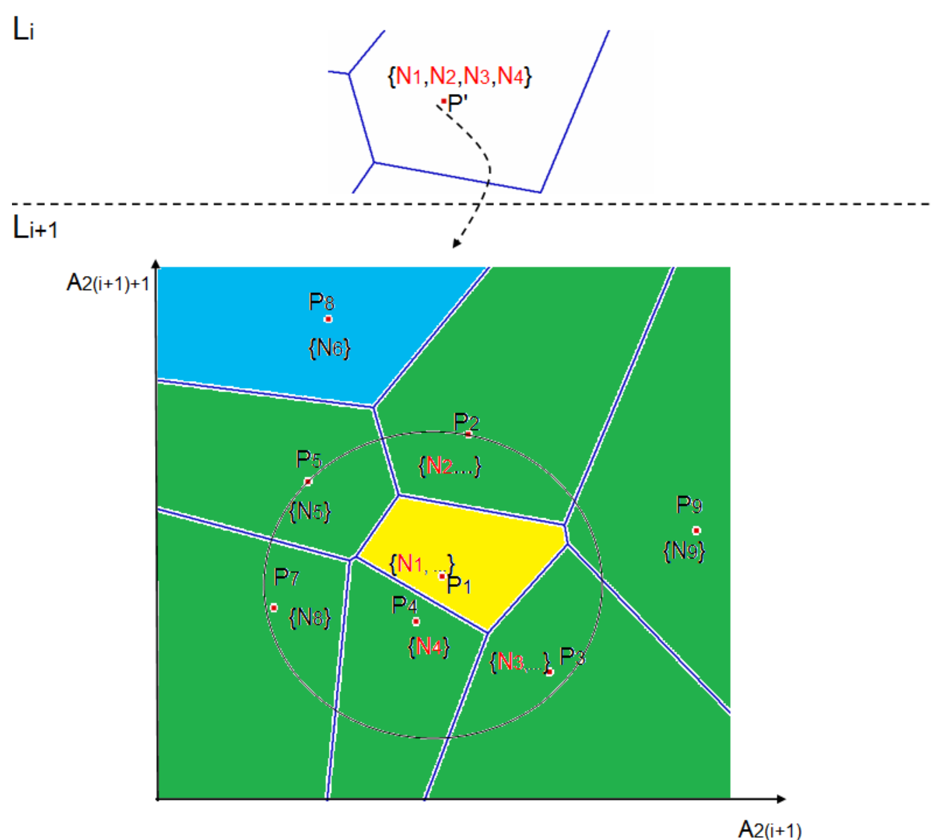


Figura 3.15: Ricerca di un SuperPeer: caso in cui un vicino di Voronoi è eleggibile

- Appena n_i riceve un messaggio di risposta non vuoto contenente un peer n_k, la cui esistenza è garantita dall'Osservazione 1, lo elegge SuperPeer e ignora i successivi messaggi di risposta eventualmente inviati dagli altri SuperPeer.

In figura 3.16 possiamo vedere un caso in cui il peer N_1 deve eleggere un nuovo SuperPeer, ma non ha vicini da poter eleggere: infatti tutti i suoi vicini sono già SuperPeer (peer segnati di rosso). Questa figura rappresenta la prima parte della ricerca del peer: N_1 invia a tutti i suoi vicini, che sono tutti SuperPeer, il messaggio di broadcast, rappresentato dalle frecce nere. La figura 3.17 mostra la seconda parte della ricerca: il peer N_5, uno dei peer contattati da N_1, ricerca fra i suoi VN (rappresentati dalle aree colorate in verde) un peer che non sia SuperPeer: nell'esempio trova il peer N_9 e invia a N_1 un messaggio di risposta.

Elezione del peer

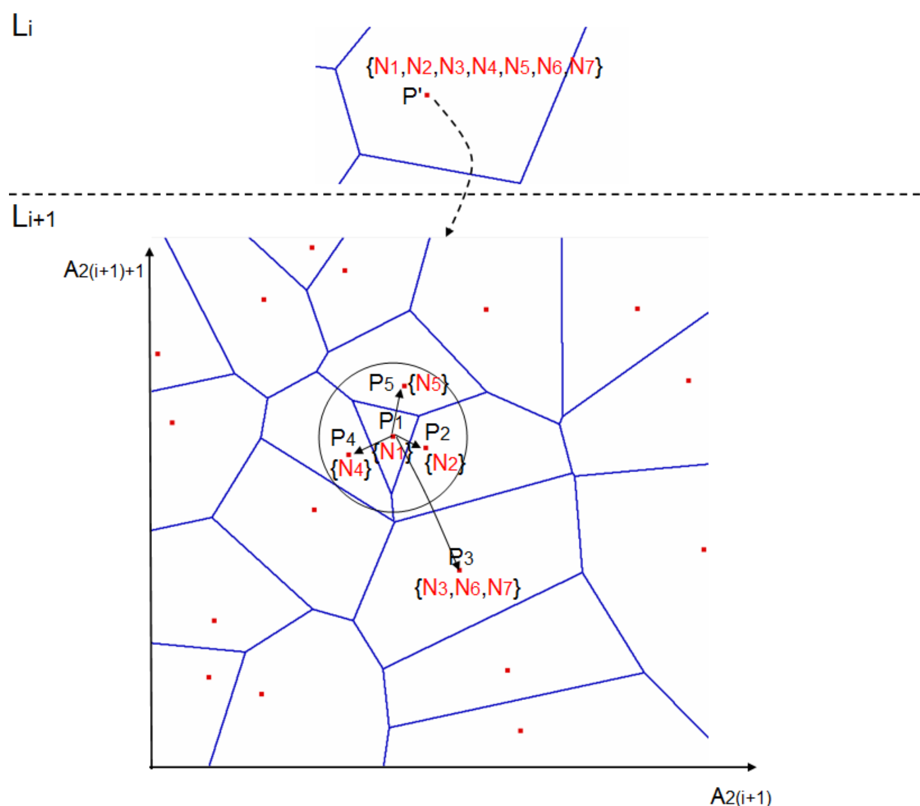


Figura 3.16: Ricerca di un SuperPeer: caso in cui un vicino di Voronoi non è eleggibile

Descriviamo adesso la seconda fase dell'algoritmo. In questa fase n_i si deve occupare delle seguenti operazioni necessarie per realizzare l'elezione del peer n individuato nella fase precedente:

- inserire il nuovo peer da eleggere nella Overlay Network dei SuperPeer. Questo viene fatto eseguendo i seguenti passi:

  1. n_i inserisce il nuovo peer nella sua lista dei SuperPeer della rete E di livello L_i.
  2. n_i invia un messaggio di notifica agli altri SuperPeer, affinché aggiornino la loro lista dei SuperPeer aggiungendoci il peer n ed attende una risposta dai SuperPeer notificati.

- notificare l'esistenza del nuovo SuperPeer a tutti i peer appartenenti ai propri insiemi di vicini di Voronoi di livello L_{i-1} in cui risulta visibile l'esistenza di esso affinché possano aggiornare la loro vista ed attende la risposta. Questo

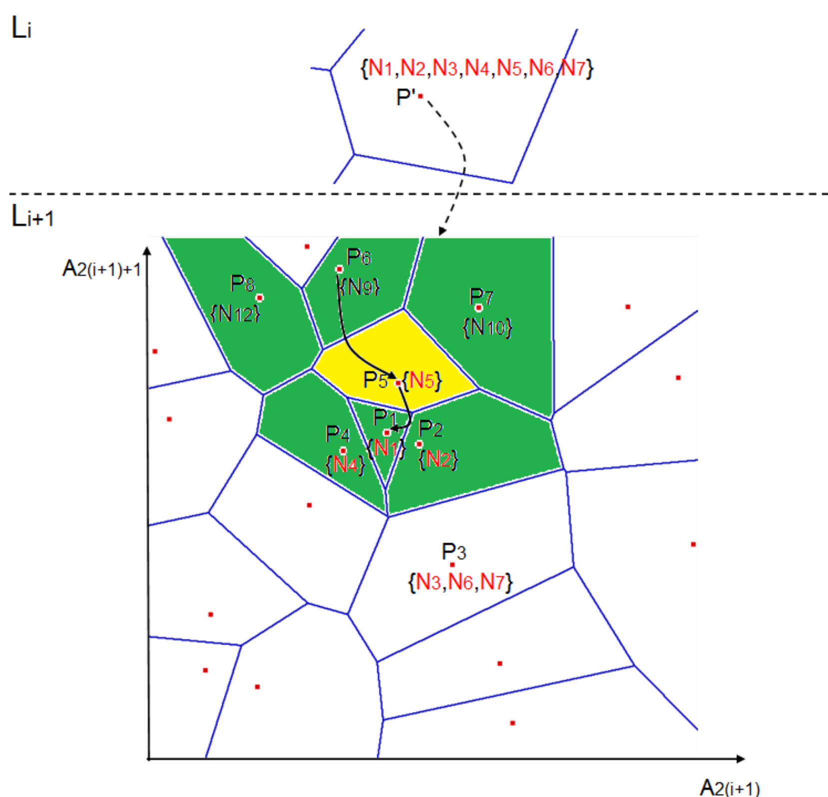


Figura 3.17: Ricerca di un SuperPeer: caso in cui un vicino di Voronoi non è eleggibile. Operazioni eseguite da un SuperPeer contattato che ripete la procedura di ricerca

passo è necessario perché ognuno di questi peer deve essere a conoscenza che un nuovo SuperPeer si è aggiunto tra i nodi visibili del cluster associato al sito p_{i-1}.

- inviare al nuovo SuperPeer n un messaggio di elezione contenente i propri insiemi dei vicini di Voronoi del livello L_{i-1} e le coordinate del punto p_{i-1}, la lista dei SuperPeer ed il numero dei peer $Num(P)$ del livello L_i aggiornati ed infine la propria lista dei SP della rete di livello L_{i-1} in cui è contenuto il sito p_{i-1}.

Il peer n, ricevuto il messaggio di elezione, esegue i seguenti passi:

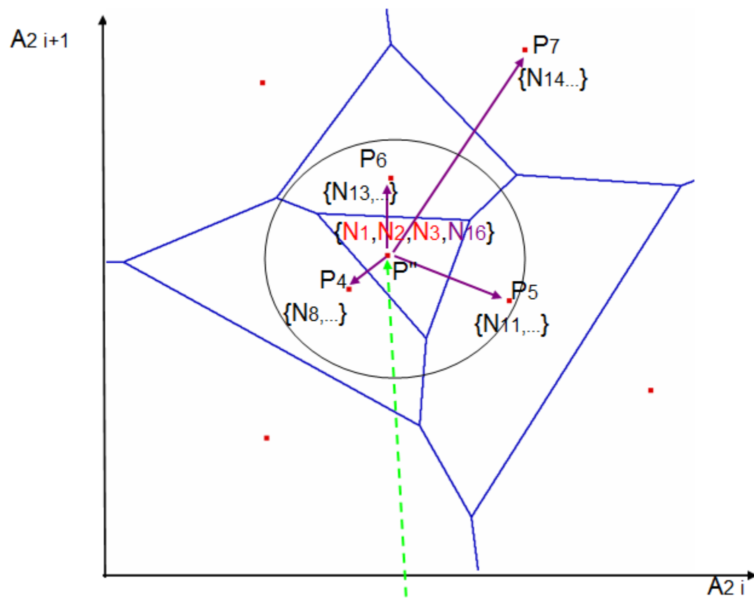
1. ricava dal messaggio il numero di livello L_{i-1} e crea una entry nella tabella dei livelli di indice $i-1$.

2. ricava dal messaggio le liste degli insiemi dei vicini di Voronoi del sito p_{i-1} e la lista dei SP della rete di livello L_{i-1} in cui è contenuto il sito p_{i-1} e le copia nelle liste corrispondenti nella entry appena creata.

3. genera il proprio grafo di Voronoi di livello L_{i-1} usando l'algoritmo di Fortune [35], contenente le aree di Voronoi del sito p_{i-1} e dei siti corrispondenti all'insieme dei propri VN.

4. ricava dal messaggio la lista dei SuperPeer e del numero dei peer $Num(p_{i-1})$ e li copia nelle corrispondenti strutture dati nella entry della tabella dei livelli di indice i

La figura 3.18 mostra le operazioni che il SuperPeer N_1 deve eseguire per eleggere il peer N_{16}. Il nuovo peer viene colorato di viola, i SuperPeer vecchi sono colorati in rosso. Le frecce nere nella rete a livello L_{i+1} rappresentano i messaggi di notifica che N_1 invia agli altri SuperPeer affinché essi aggiungano il nuovo peer N_{16}. Le frecce viola rappresentano i messaggi di notifica che N_1 manda a tutti i peer che fanno parte dei suoi insiemi di vicini di Voronoi del livello L_i affinché essi aggiornino la loro vista. La freccia rossa tratteggiata rappresenta il messaggio inviato a N_{16} per comunicargli la lista dei SuperPeer, il punto P affinché possa costruire il suo grafo di Voronoi del livello L_{i+1} e il numero dei peer del livello L_i: in questo caso il messaggio conterrebbe P, la lista dei SuperPeer [N_1, N_2, e N_3] e $Num(P) = 16$.



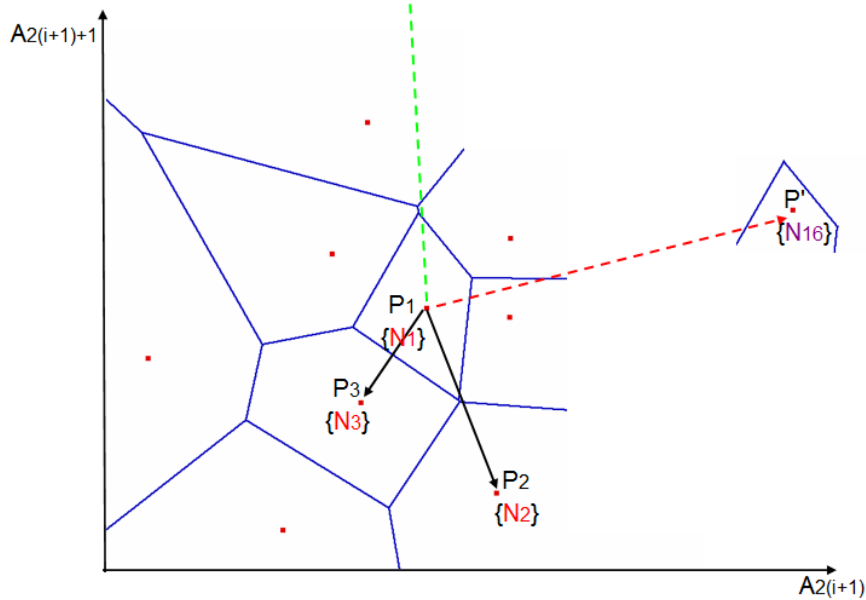


Figura 3.18: Elezione di un SuperPeer: messaggi e notifiche inviate da N_1 per aggiornare le viste dei peer coinvolti nell'elezione

3.4.5 Studio della complessità degli algoritmi

Il calcolo della complessità degli algoritmi di Hierarchical Voraque si presenta apparentemente di una certa difficoltà, dato che sembrerebbe dipendere da come sono distribuiti i peer nei vari livelli che lo compongono. In realtà può essere dimostrato che non sempre è così. L'analisi mostra la complessità degli algoritmi di Inserzione di un peer e di risoluzione di una range query multiattributo in due casi estremi, che abbiamo utilizzato per effettuare i nostri test nel capitolo 5:

- Uso di una distribuzione uniforme random
- Uso di una ditribuzione che sbilanci la distribuzione dei valori degli attributi sui peer della rete, come la power-law[34].

La complessità viene studiata in numero di messaggi scambiati sulla rete e, in alcuni casi, in numero di peer attraversati (Hop) rispetto al numero totale di peer della rete, in quanto possiamo supporre che il tempo di calcolo locale ai nodi sia trascurabile rispetto al tempo di trasmissione e ai ritardi della rete fisica.

Iniziamo facendo una particolare osservazione nel caso in cui l'intera struttura di Hierarchical VoRaQue appaia come un albero bilanciato, cioè nel caso in cui usiamo una distribuzione uniforme del valore degli attributi dei peer, che corrisponde a distribuire uniformemente i peer sui siti della rete. Con questo si intende che sottoreti poste allo stesso livello (ovvero nodi di un medesimo livello nell'albero) hanno lo stesso numero di peer.

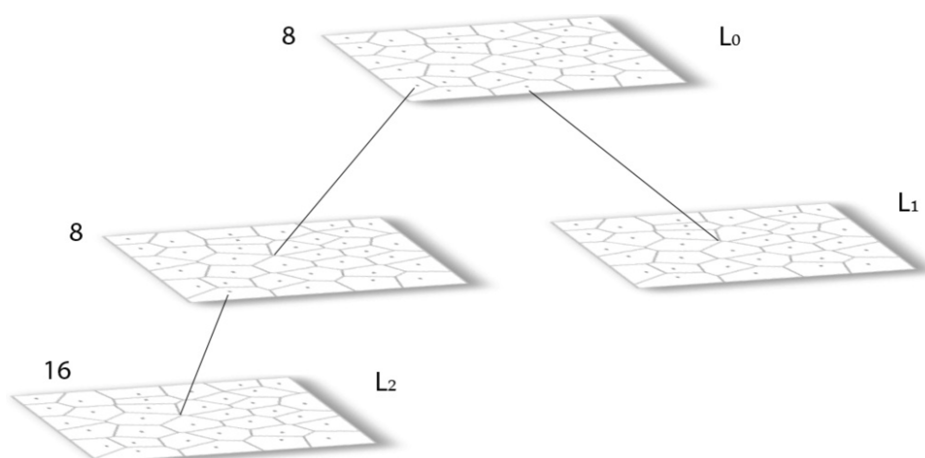


Figura 3.19: Esempio di albero di Hierarchical Voraque: le frecce collegano un punto di un livello superiore al livello sottostante da lui generato

Definiamo N il numero dei peer dell'intera rete. Nel caso di un albero totalmente bilanciato, questi saranno stati inseriti tutti in reti al livello L_{Max} dell'albero. Ogni nodo interno dell'albero rappresenta una suddivisione dei peer in base

ai valori dei loro attributi. Nell'esempio in figura 3.19 viene presentata una parte dell'albero bilanciato, con un numero massimo di livelli $L_{Max} = 2$, cioè in totale una rete con 3 livelli e 6 attributi. Supponendo di avere $N = 1024$, se avessimo 8 siti nella rete di livello L_0 e altrettanti per ogni rete al livello L_1, ogni rete di livello L_2 avrebbe 16 peer, cioè $\frac{1024/8}{8}$. Generalizzando, otteniamo che dati Max livelli e dato P_i il numero di siti di ogni rete di livello L_i, i peer di ognuna delle reti di livello L_{Max} saranno quindi $P_{Max} = \frac{N}{\prod_{i=0}^{Max-1} P_i}$ nel caso peggiore, in cui all'ultimo livello ho un peer per ogni punto. Questo risultato ci servirà sia nello studio della complessità del routing greedy, sia in quella del compass routing, entrambi algoritmi fondamentali per l'operazione di inserzione di nuovi peer e quella di risoluzione delle range query multiattributo.

Algoritmo di inserzione di un nuovo nodo

Il caso di maggior costo per l'inserzione si ha quando i peer iniziano ad inserirsi in una delle reti all'ultimo livello dell'albero che rappresenta la rete di Hierarchical Voraque, sia nel caso di in una distribuzione uniforme che sbilanciata. Supponiamo di avere già inserito N_i peer, e vogliamo inserire l'N_i+1-esimo, nella situazione in cui da N_i in poi tutti i peer si inseriscono in una delle reti al livello L_{Max}. Da ora in poi N_i sarà la dimensione attuale N dell'intera rete di Hierarchical Voraque in cui si inserisce il nuovo peer. Per esempio, supponendo come sopra di avere $N = 1024$ peer già inseriti, noi ora stiamo inseriremo il 1025-esimo.

Premettiamo alcuni risultati di complessità su alcuni casi particolari:

1. La costruzione di una nuova rete ad un certo livello L_i di Hierarchical Voraque ha costo $O(1)$, in quanto il numero di peer coinvolti nella costruzione è uguale alla costante di soglia $S \ll N$.

2. Il passaggio da una rete di livello superiore a quella di livello immediatamente inferiore non comporta nessun costo in quanto il peer coinvolto nella rete di livello superiore è sicuramente un SP della rete di livello inferiore per costruzione.

3. Il costo per ottenere la lista dei SP di livello L_{Max} è costante perché durante l'inserzione si può ottenere direttamente dal peer che ha iniziato il routing greedy all'ultimo livello, che è un SP e che viene comunicato al nuovo peer dal peer gestore.

L'inserimento di un nuovo peer quindi consiste in 3 passi fondamentali:

1. Esecuzione dell'algoritmo greedy a partire da un peer n_{boot} del livello L_0 per giungere fino al sito di livello L_{Max} che contiene il peer gestore dell'inserimento.

2. Inserzione da parte del peer gestore nella rete di livello L_{Max} del nuovo peer.
3. Eventuale elezione di un nuovo SP: il numero di elezioni che si eseguono dopo l'inserzione di un nuovo peer sono al massimo una per ogni livello dell'albero di Hierarchical Voraque.

Iniziamo con l'analisi della complessità del routing greedy.

Possiamo osservare che la struttura ad albero di Hierarchical Voraque potrebbe anche essere vista come una serie di suddivisioni in celle di Voronoi, partendo da una suddivisione grossolana, usando solo una coppia di attributi. Questa divisione viene via via raffinata usando una nuova coppia di attributi. Come esempio, in figura 3.20 si può vedere una delle reti a livello L_1 proiettata nella corrispondente area di Voronoi-padre a livello L_0, delimitata da lati in nero.

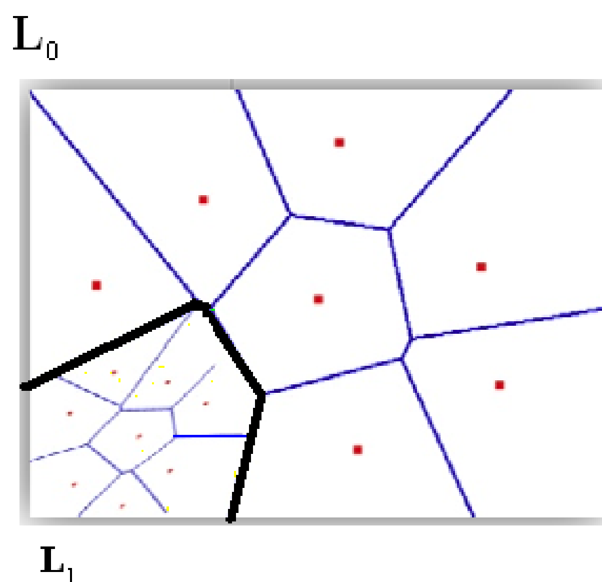


Figura 3.20: Esempio di area a livello L_1 mappata nell'area-padre a livello L_0

Vedendo la situazione in questa ottica, si può pensare al routing greedy come un routing che si muove dapprima in macro-aree determinate dagli attributi di livello superiore, per poi proseguire in celle sempre più fitte, determinate ogni volta da ulteriori attributi. In pratica, si può pensare che il routing è come se avvenisse in una rete di Voronoi bidimensionale, dove solo la zona della cella di destinazione è suddivisa usando tutti gli attributi.

Ricordiamo che la complessità del routing greedy in una rete di Voronoi bidimensionale è $O(\log N)$, con N il numero di peer della rete. In Hierarchical

Voraque il routing viene effettuato a partire dai livelli più alti, scendendo via via nelle sottoreti fino ad arrivare al punto che soddisfa tutte le caratteristiche richieste sugli attributi. Se P_i è il numero di siti di una rete al livello L_i, la complessità per il routing in quella rete risulta $\log P_i$. In totale, avendo Max livelli, il routing complessivo richiederà:

$$\log P_0 + \log P_1 + \ldots + \log P_{Max} = \sum_{i=0}^{Max} \log P_i \tag{3.1}$$

Ricordiamo che in ipotesi di peer distribuiti in modo uniforme tra le varie sottoreti, si ha che $P_{Max} = \frac{N}{\prod_{i=0}^{Max-1} P_i}$. Sostituendo nell'equazione 3.1 otteniamo:

$$\begin{aligned} \log P_0 + \log P_1 + \ldots + \log P_{Max} = \log(P_0 P_1 \ldots P_{Max}) = \\ \log \prod_{i=0}^{Max-1} P_i \frac{N}{\prod_{i=0}^{Max-1} P_i} = \log N \end{aligned} \tag{3.2}$$

Quindi pur gestendo un numero di attributi **qualsiasi**, con peer uniformemente distribuiti il routing greedy ha una complessità uguale a quello eseguito su una rete di Voronoi bidimensionale con lo stesso numero complessivo di peer. Il risultato ottenuto è valido sia come limite superiore al numero dei messaggi inviati sulla rete, sia per il numero degli hop, cioè dei siti attraversati: infatti, nel routing greedy per ogni hop si invia un messaggio ad un solo peer, sia che il sito successivo sia un cluster o meno.

Analizziamo ora il routing greedy nel caso di una rete completamente sbilanciata, come si avrebbe nel caso di una distribuzione dei peer sui siti che segue una power-law[34]. In questo caso abbiamo un sistema Hierarchical Voraque in cui una sola rete al livello L_{Max} ha un numero di peer, e di siti considerando un peer per sito come caso pessimo, $N_{Max}^{Max} \approx N$. Facendo ancora riferimento alla figura 3.17, si vede che tutte le reti dei livelli superiori a L_{Max} determinano un ripartizione tra i siti che le costituiscono dei peer dei livelli sottostanti. Da questo deriva che tutte le altre reti di livello L_{Max} diverse da quella presa in considerazione hanno un numero di peer $N_{Max} \ll N$. Di conseguenza, anche le reti di livello superiore non possono contenere un numero maggiore di siti rispetto al numero dei peer contenuti in esse. Quindi abbiamo che $\forall i < Max.P_i \ll N$. Nel caso peggiore, il punto target del routing potrebbe finire esattamente nella rete con N_{Max}^{Max} peer, quindi otteniamo un costo:

$$\log P_0 + \log P_1 + \ldots + \log P_{Max}^{Max} = \log P_0 + \log P_1 + \ldots + \log N_{Max}^{Max} \underrightarrow{\lim} \log N_{Max}^{Max} \tag{3.3}$$

per $\log N_{Max}^{Max}$ sufficientemente grande. Essendo $N_{Max}^{Max} \approx N$ allora anche $\log N_{Max}^{Max} \approx \log N$. Quindi possiamo concludere che, **indipendentemente** dal numero di attributi usati, sia nel caso di un sistema completamente bilanciato, sia nel caso di una rete completamente sbilanciata, la complessità per effettuare un routing greedy in Hierarchical VoRaQue è $O(\log N)$, come nel caso bidimensionale.

Passiamo ad analizzare il costo di inserzione del nuovo peer nella rete di livello L_{Max} nel caso di una rete perfettamente bilanciata. Se ci mettiamo nel caso peggiore per l'algoritmo greedy, allora significa che ad ogni peer corrisponde un diverso sito nella rete a livello L_{Max}: il nuovo peer quindi genera una nuova area di Voronoi. Il peer ha un numero statisticamente costante di siti vicini della nuova area di Voronoi in quanto la rete di livello L_{Max} è bidimensionale, ed in ognuno di questi è presente un solo peer. Il peer gestore deve inviare un messaggio ad ogni peer che faccia parte dell'insieme dei vicini di Voronoi del nuovo peer affinché aggiorni la propria vista della rete, e riceve una risposta da ognuno di questi da inviare al nuovo peer. Il numero di messaggi è quindi uguale al numero di peer vicini del nuovo peer da inserire, ed essendo questo un insieme costante porta ad una complessità $O(1)$. La ricerca del Long Range corrisponde ad effettuare il greedy solo sulla rete di ultimo livello, quindi costa meno del greedy totale.

Supponiamo invece di avere al livello L_{Max} un numero di siti P_{Max}. In questo caso abbiamo che ogni sito è associato ad un cluster di dimensione $\frac{N}{\prod_{i=0}^{Max} P_i}$: il nuovo peer si inserisce in uno di questi siti. Se la dimensione dei cluster è minore della soglia S, allora il peer ha un numero statisticamente costante di siti vicini (per la bidimensionalità della rete) con un numero costante di peer dato che $S \ll N$. Il peer deve inviare un messaggio per ogni companions e per ogni peer che fa parte dei suoi vicini di Voronoi affinché aggiornino la loro vista della rete, quindi ottengo ancora una complessità $O(1)$. Se invece la dimensione del cluster è maggiore della soglia S allora il nodo si inserisce tra i peer del cluster che non sono visibili alla rete: il costo di inserimento si riduce alla comunicazione ai SP di questo cluster dell'entrata del nuovo nodo. I SP sono esattamente $\log \frac{N}{\prod_{i=0}^{Max} P_i}$, quindi la complessità risulta $O(\log N)$.

Supponiamo ora di avere una rete completamente sbilanciata. Allora anche in questo caso, come descritto per il routing greedy, esiste una sola rete di livello L_{Max} con un numero di peer $N_{Max}^{Max} \approx N$. Ogni altra rete di un qualsiasi livello L_i diversa da quella presa in considerazione, ha un numero di peer $N_i \ll N$. Quindi il costo più alto si avrà nella rete più ampia. Se il numero di peer di questa rete è di uno per ogni sito, allora come nel caso precedente avremo che il costo risulta di complessità $O(1)$. Altrimenti, se esiste un sito nella rete presa in considerazione dove si inseriscono quasi tutti gli N peer, allora abbiamo sempre come nel caso

precedente una complessità $O(\log N)$, esattamente identica ai SP del sito preso in considerazione.

In conclusione il costo per inserire il nuovo peer in una rete di livello L_{Max}, come per il routing greedy, risulta **indipendente** dal numero degli attributi della rete e nel caso pessimo risulta, sia per una rete completamente bilanciata che per una rete sbilanciata, uguale a $O(\log N)$.

Per quanto riguarda le elezioni, faccio un'analisi ammortizzata, perché l'elezione non viene effettuata per ogni peer inserito. In sostanza, verifichiamo quanto costano tutte le elezioni e lo dividiamo per il numero N dei peer della rete per ottenere quanto pesa in media l'elezione per un singolo inserimento, senza preoccuparci se esso abbia o no provocato delle elezioni. Successivamente sommiamo questo valore al costo dell'algoritmo greedy e a quello di inserzione nel punto target, per poter avere la complessità media dell'inserimento per ogni peer. Ricordiamo che l'operazione di elezione di un peer consiste in tre fasi, eseguite da un SuperPeer che farà parte dello stesso insieme di SP in cui entrerà anche il nuovo peer:

1. Ricerca di un peer da eleggere: al massimo, viene inviato un messaggio di richiesta ad ognuno degli altri SP sulla base dell'Osservazione 1, quindi almeno uno degli altri SuperPeer lo deve avere come suo vicino di Voronoi. Nel caso in cui il nuovo peer inserito è visibile a questa rete, questa fase è saltata e viene eletto il peer inserito senza inviare messaggi.

2. Invio agli altri SuperPeer della notifica dell'elezione del nuovo SuperPeer affinché lo aggiungano alla loro lista dei SP.

3. Invio ai peer che fanno parte degli insiemi di vicini di Voronoi della rete di livello superiore di un messaggio di notifica dell'entrata del nuovo peer nella rete affinché aggiornino la loro vista della rete.

Le prime due fasi dell'algoritmo hanno la stessa complessità perché coinvolgono lo stesso insieme di peer con lo stesso numero di messaggi.

Iniziamo dal caso di un albero completamente bilanciato. Supponiamo di voler eleggere un peer ad un certo livello L_i. In questo caso le 3 fasi hanno lo stesso costo: infatti avremo che le reti di livello L_{i+1} hanno tutte dimensione $\frac{N}{\prod_{j=0}^{i} P_j}$, quindi anche quella in cui eleggiamo il nuovo SuperPeer. I peer visibili di questi cluster corrispondono ai SP già eletti, cioè $\log(\frac{N}{\prod_{j=0}^{i} P_j})$. Essendo la rete di livello L_i bidimensionale, ogni sito di questa rete ha un numero di siti vicini costante, quindi il costo complessivo per l'elezione risulta $\log(\frac{N}{\prod_{j=0}^{i} P_j})$.

Il numero totale di elezioni eseguite dalla rete di livello L_{i+1} è uguale al numero attuale dei SP della rete, cioè $\log(\frac{N}{\prod_{j=0}^{i} P_j})$. Quindi il costo totale per eseguire

tutte le elezioni della particolare rete presa in considerazione di livello L_i risulta $\log^2(\frac{N}{\prod_{j=0}^{i} P_j})$. Il numero di reti del livello L_{i+1} è uguale al numero dei siti della rete di livello L_i, cioè P_i. Questo deriva dal fatto che Hierarchical Voraque ha le seguenti caratteristiche:

- Ogni livello del'albero definisce reti di Voronoi basate su coppie di attributi diversi da quelli di tutti gli altri livelli.
- Ogni sito in una rete di livello diverso da L_{Max} definisce una ulteriore rete di Voronoi al livello sottostante.

In conclusione, il numero di elezioni effettuate dalle reti di livello L_{i+1} ha costo complessivo $P_i \log^2(\frac{N}{\prod_{j=0}^{i} P_j})$.

Generalizzando la formula per una qualsiasi rete, otteniamo che il costo totale di tutte le elezioni ad ogni livello dell'albero risulta:

$$P_0 \log^2(\frac{N}{P_0}) + P_1 \log^2(\frac{N}{P_0 P_1}) + \ldots + P_{Max-1} \log^2(\frac{N}{\prod_{i=0}^{Max-1} P_i}) \leq \log^2 N * \sum_{i=0}^{Max-1} P_i \quad (3.4)$$

Questo è il costo di tutte le operazioni di elezione per una rete di Hierarchical Voraque con N peer, rappresentata da un albero bilanciato. Dividendo per N otteniamo il costo ammortizzato per ogni peer, cioè $\sum_{i=0}^{Max-1} P_i * \frac{\log^2 N}{N}$. Concludendo per il caso di una rete rappresentata da un albero bilanciato, il costo ammortizzato delle elezioni risulta di ordine inferiore a $O(\log N)$, quindi trascurabile rispetto al routing greedy e all'inserzione del nuovo peer nel punto target.

Analizziamo il caso di una rete rappresentata da un albero completamente sbilanciato. Come descritto per il routing greedy, esiste una sola rete di livello L_{Max} con un numero di peer $N_{Max}^{Max} \approx N$. Esiste a sua volta una sola rete di livello L_{Max-1} che contiene il sito dove sono mappati gli SP della rete di livello L_{Max} in cui sono inseriti quasi tutti i peer. Questo sito contiene $\log(N_{Max}^{Max}) \approx \log N$ peer. Ogni altra rete, di qualunque livello L_i, ha un numero di peer $N_i \ll N$, e quindi elegge un numero di SP trascurabile al livello superiore. Per questo motivo, la rete di livello L_{Max-1} suddetta ha praticamente un numero di siti uguale a 1, e questo unico sito è associato al cluster di dimensione $N_{Max}^{Max} \approx N$ che contiene tutti i peer, ed elegge quindi un numero di SP uguale a $\log(N_{Max}^{Max}) \approx \log N$.

Questo ragionamento è possibile ripeterlo ad ogni livello fino a raggiungere un sito nella rete al livello L_0. Quindi il numero totale di elezioni è uguale a $Max * \log N$. Avendo ogni rete in pratica un solo sito, il costo delle fasi 1 e 3 dell'algoritmo di elezione sono trascurabili. La fase 2 ha invece complessità $\log N$,

uguale al numero dei SP dell'unico sito. Otteniamo quindi che in caso di una rete rappresentata da un albero completamente sbilanciato il costo ammortizzato per ogni peer è $Max * \frac{\log^2 N}{N}$.

Concludendo anche per il caso di una rete rappresentata da un albero completamente sbilanciato il costo ammortizzato delle elezioni risulta di ordine inferiore a $O(\log N)$, quindi trascurabile rispetto al routing greedy e all'inserzione del nuovo peer nel punto target.

Quindi possiamo concludere che, **indipendentemente** dal numero di attributi usati, sia nel caso di un sistema completamente bilanciato, sia di uno completamente sbilanciato, la complessità dell'inserzione di un peer in Hierarchical VoRaQue è $O(\log N)$, identico a quello di una rete di Voronoi bidimensionale, essendo trascurabile l'elezione.

Algoritmo di risoluzione di una range query multiattributo

La risoluzione di una range query in una singola rete di Hierarchical Voraque raggiunta dalla query, richiede di raggiungere un sito dell'AOI locale tramite routing greedy e quindi di comunicare con tutti i peer all'interno dell'AOI stessa tramite compass routing.

Ci concentriamo innanzitutto sull'analisi del numero dei messaggi necessari per contattare tutti i peer che rispettino i vincoli multiattributo posti dalla query, cioè della complessità totale degli algoritmi di compass routing necessari per visitare le varie AOI.

Iniziamo ancora dal caso in cui l'albero che rappresenta la rete sia bilanciato. Data una determinata rete a livello L_i che contiene P_i siti, se la AOI definita dalla query su quella rete è pari ad una percentuale s_i (che corrisponde alla selettività composta degli attributi mappati nelle reti al livello L_i) dell'area totale della rete considerata, il numero di siti coinvolti risulta $s_i P_i$.

Data una range query multiattributo qualsiasi e data la selettività s_0 del livello L_0, la AOI della query comprenderà quindi $s_0 P_0$ siti del livello L_0. Questi siti definiscono altrettante reti di Voronoi al livello L_1. Data la selettività s_1 del livello L_1 e supponendo che ogni rete di questo livello abbia P_1 siti, anche in questo caso avremo che la AOI della query comprenderà $s_1 P_1$ siti per ognuna delle reti del livello L_1. In totale, la query comprende quindi un numero di siti uguale a $s_0 P_0 s_1 P_1$ del livello L_1, che andranno sommati a quelli compresi al livello L_0. Il processo si itera fino al raggiungimento del livello L_{Max} della rete.

Per esempio, considerando una rete con 3 livelli, come quella in figura 3.17, complessivamente la query contatta un numero di siti uguale a $s_0 P_0 + s_0 P_0 s_1 P_1 + s_0 P_0 s_1 P_1 s_2 P_2$.

In generale, avendo $Max + 1$ livelli, il numero di siti complessivamente contattati risulta:

$$\sum_{i=0}^{Max} \prod_{j=0}^{i} s_j P_j \tag{3.5}$$

Ricordiamo che in ipotesi di peer distribuiti in modo uniforme tra le varie sottoreti, si ha che $P_i = \frac{N}{\prod_{j=0,j\neq i}^{Max-1} P_i}$. Ad esempio, con 3 livelli la formula 3.4 può essere riscritta come $s_0 * \frac{N}{P_1 P_2} + s_0 P_0 s_1 \frac{N}{P_0 P_2} + s_0 P_0 s_1 P_1 s_2 \frac{N}{P_0 P_1} = N(\frac{s_0}{P_1 P_2} + \frac{s_0 s_1}{P_1} + s_0 s_1 s_2)$. Sulla base di queste osservazioni, la formula generale può essere riscritta come:

$$N(\sum_{i=0}^{Max-1} \frac{\prod_{j=0}^{i} s_j}{\prod_{k=i+1}^{Max} P_k} + \prod_{h=0}^{Max} s_h) \tag{3.6}$$

Dimostriamo adesso il seguente teorema:

Teorema 3 *Per ogni* $i = 0 \ldots Max - 1$*, si ha che* $\frac{\prod_{j=0}^{i} s_j}{\prod_{k=i+1}^{Max} P_k} \leq \prod_{h=0}^{Max} s_h$

Dimostrazione: Dato $i \in [0, Max - 1]$, ed essendo $P_k \geq 1 \forall k \in [i+1, Max]$, la formula da dimostrare si può scrivere come:

$$\begin{aligned} & \prod_{j=0}^{i} s_j \leq \prod_{h=0}^{Max} s_h \prod_{k=i+1}^{Max} P_k \\ = & \prod_{j=0}^{i} s_j \leq \prod_{h=0}^{i} s_h \prod_{k=i+1}^{Max} s_k \prod_{k=i+1}^{Max} P_k \\ = & \prod_{j=0}^{i} s_j \leq \prod_{h=0}^{i} s_h \prod_{k=i+1}^{Max} s_k P_k \end{aligned} \tag{3.7}$$

Dato che $s_k P_k$ indica il numero di siti compresi nell'AOI di una rete al livello L_k, si ha che $s_k P_k \geq 1$ visto che l'AOI ricade almeno all'interno di una cella di Voronoi, ed ogni cella di Voronoi ha sempre un sito che la definisce. Come conseguenza si ha che $\prod_{k=i+1}^{Max} s_k P_k \geq 1$. Posto $\prod_{k=i+1}^{Max} s_k P_k = a$ e rinominando gli indici otteniamo infine che $\prod_{j=0}^{i} s_j \leq a \prod_{j=0}^{i} s_j$ con $a \geq 1 \forall i \in [0, Max - 1]$. Quest'ultima formula è sempre vera, quindi il teorema è dimostrato essendo equivalente alla formula da dimostrare.

Sulla base di questo, possiamo concludere che:

$$N(\sum_{i=0}^{Max-1} \frac{\prod_{j=0}^{i} s_j}{\prod_{k=i+1}^{Max} P_k} + \prod_{h=0}^{Max} s_h) \leq N * Max * \prod_{h=0}^{Max} s_h = Max * s_0 \ldots s_{Max} * N \tag{3.8}$$

Quindi la complessità totale per la risoluzione di una range query multi-attributo nel caso di una distribuzione uniforme è al massimo pari al numero complessivo dei peer della rete per Max (numero dei livelli) volte la produttoria delle selettività di ogni coppia di attributi. Concludendo risulta che il numero di nodi contattati dalla query è proporzionale alla selettività combinata di tutte le coppie di attributi.

Consideriamo ora il caso di una rete rappresentata da un albero totalmente sbilanciato. Anche in questo caso esiste una sola rete all'ultimo livello con un numero di peer $N_{Max}^{Max} \approx N$. Ogni altra rete di livello L_i diversa da quella presa in considerazione, ha un numero di peer $N_i \ll N$. Quindi il costo più alto si avrà nella rete più ampia. Data s_{Max}^{Max} indica la selettività combinata degli attributi di questa rete, avremo che il costo risulta proporzionale a $s_{Max}^{Max} N_{Max}^{Max}$. Da un punto di vista di selettività della query rispetto al numero totale di peer, se la query viene diretta verso la rete più popolata, contenente la quasi totalità dei peer, significa che i range espressi sui precedenti attributi avevano uno scarso potere discriminatorio tra tutti i siti. Ne deriva che la coppia di attributi (e quindi la AOI) più selettiva è quella con selettività s_{Max}^{Max}, che possiamo quindi assumere come $s_{Massima}$, cioè la selettività massima. Come conseguenza, possiamo concludere che in questo caso la complessità è $O(s_{Massima}N)$. Questo risultato è valido se a livello L_{Max} la rete ha un peer per sito. Se invece prendiamo il caso in cui anche al livello massimo esiste un sito in cui cadono $N_{Max}^{Max} \approx N$ peer, questi per costruzione hanno il valore di tutti gli attributi uguali: basta quindi contattare un solo peer di questo insieme (uno di quelli eletti SuperPeer) per venire a conoscenza se tutti sono match per la query. In questo caso quindi la complessità risulta $O(1)$, indipendentemente dalle selettività e dal numero degli attributi.

Riassumendo, la complessità per effettuare la risoluzione di una range query multiattributo in Hierarchical VoRaQue è $O(Max * s_0 * \ldots * s_{Max} * N)$ nel caso di un sistema perfettamente bilanciato, e $O(s_{Massima}N)$ o $O(1)$ (a seconda se l'albero è sbilanciato anche al livello L_{Max}) nel caso di un sistema totalmente sbilanciato, dove una sottorete contiene quasi tutti gli N peer.

A questo costo dobbiamo aggiungere il costo del routing greedy necessario a raggiungere il centro di ogni AOI delle reti attraversate. Iniziamo considerando il caso di una rete rappresentata da un albero totalmente bilanciato. Supponendo per esempio di avere come in figura 3.17 una rete con 3 livelli avremo che il routing $\log P_0$ al primo livello, $s_0 P_0 \log P_1$ per le reti del secondo livello e $s_0 P_0 s_1 P_1 \log P_2$ per le reti del terzo livello, cioè in totale $\log P_0 + s_0 P_0 \log P_1 + s_0 P_0 s_1 P_1 \log P_2$. In

generale avremo, nel caso peggiore, che:

$$\log P_0 + s_0 P_0 \log P_1 + \ldots + \prod_{i=0}^{Max-1} (s_i P_i) \log P_{Max} \leq (\sum_{i=0}^{Max} log(P_i)) * \prod_{i=0}^{Max-1} (s_i P_i)$$
$$= \log (\prod_{i=0}^{Max} P_i) * \prod_{i=0}^{Max-1} (s_i P_i) = \log N * \prod_{i=0}^{Max-1} (s_i P_i) \quad (3.9)$$

Passiamo al caso di una rete rappresentata da un albero completamente sbilanciato. Come nei ragionamenti sul compass, quasi tutti i peer sono concentrati in un'unica rete a livello L_{Max}, quindi i costi per effettuare il greedy ai livelli superiori è trascurabile perché contengono un numero di peer $N_i \ll N$. Quindi la complessità risulta $O(\log N)$.

Concludendo, la complessità totale per effettuare la risoluzione di una range query multiattributo in Hierarchical VoRaQue è $O(Max * s_0 * \ldots * s_{Max} * N) + O(\log N * \prod_{i=0}^{Max-1}(s_i P_i))$ per una rete rappresentata da un albero completamente bilanciato e $O(s_{Massima} N) + O(\log N)$ oppure solo $O(\log N)$, a seconda se esiste o no un sito al livello L_{Max} in cui si concentrano tutti i peer, in una rete rappresentata da un albero completamente bilanciato.

Possiamo notare un particolare risultato ottenuto dalla nostra analisi: **l' ordine in cui sono mappati gli attributi sui livelli della rete di Hierarchical Voraque è non influisce sulla complessità della risoluzione di una range query multiattributo**. Questo significa che in realtà non è importante, come invece abbiamo indicato nel paragrafo 3.1, mappare gli attributi in ordine di selettività.

Capitolo 4

Implementazione di Hierarchical Voraque

In questo capitolo vengono specificati gli algoritmi per la risoluzione delle range query multiattributo e della inserzione di nuovi peer nella rete Hierarchical Voraque descritta nel precedente capitolo.

La descrizione dell'implementazione viene fatta ad alto livello, specificando gli algoritmi tramite uno **pseudocodice** e descrivendo i campi dei diversi messaggi ed il loro significato. Per ogni sezione di pseudocodice vengono indicati i parametri di ingresso e le strutture dati principali utilizzate. In particolare:

- **VL** è il vettore dei livelli indicato nel paragrafo relativo alla descrizione delle strutture dati.
- **CreateTypeMSG** sono funzioni che creano un messaggio di tipo *Type* contenente i valori indicati come parametri.
- **HTVN, HTCN, LLRN e HTBLRN** sono le hash table, specifiche per peer e per livello, dei vicini come indicato nel paragrafo 3.1. LLRN non è una hash table ma è una semplice lista di peer perché il Long Range è un unico punto.
- **SelectRandomNode** è una funzione che seleziona un peer a caso da un insieme di peer indicato come parametro
- **EstraiLvl** è una funzione che estrae il livello contenuto nel messaggio passato come parametro.
- **S** rappresenta la costante di soglia

4.1 Range Query multiattributo: l'implementazione

Data una rete di livello L_i la risoluzione di una range query per gli attributi locali $[A_{2i}, A_{2i+1}]$ corrisponde ad eseguire le seguenti fasi:

1. algoritmo di routing greedy per raggiungere l'AOI locale.
2. algoritmo di routing compass per visitare tutti i peer dell'AOI.
3. propagazione al livello inferiore della query per ogni sito dell'AOI costituito da un cluster di dimensione maggiore di S.

Alla fase di routing greedy e di compass routing corrispondono una serie di messaggi caratterizzati dai seguenti campi comuni:

- *Livello*: indica il livello di Hierarchical Voraque a cui è destinato il messaggio. É necessario perché il peer destinatario può essere visibile in più di un livello.
- *QueryID*: identificatore univoco della query, utile in caso più query siano immesse simultaneamente nella rete.
- *PoligonoQuery*: un vettore che contiene i valori Min e Max per ogni attributo della range query.

Il messaggio di tipo GreedyMSG contiene inoltre i seguenti campi:

- $ID_{creator}$: identificatore univoco del peer che ha immesso la query nel sistema, necessario per poter inviare le risposte alla query.
- $FollowUp$: valore booleano che viene settato a true se dobbiamo propagare il messaggio al livello superiore. Se è settato, significa che stiamo eseguendo l'algoritmo per ricercare un peer al livello 0. Raggiunto questo peer, viene settato a false.
- $StopAtCenter$: valore booleano che indica se l'algoritmo deve terminare con target il punto vicino al centro dell'AOI, o se termina al primo punto interno all'AOI più vicino al peer che sta eseguendo l'algoritmo greedy.

Il messaggio di tipo RangeMSG contiene inoltre i seguenti campi:

- ID_{root}: identificatore univoco del peer radice del compass routing. Serve per poter calcolare i figli di un peer facente parte dell'AOI.

- *ListHitNodes*: contiene la lista dei peer che sono risposta alla query fino a questo passo dell'algoritmo.

- ID_{sender} identificatore univoco del peer che mi ha spedito il messaggio di compass routing, insieme al punto p delle sue coordinate a questo livello. Necessario per poter calcolare i figli di un peer facente parte dell'AOI.

Il peer che immette la query nel sistema, che chiamo $n_{creator}$ inizia generando un messaggio di tipo GreedyMSG, ed esegue l'algoritmo di ricerca di un peer di livello L_0 descritto nel capitolo 3, come possiamo vedere nello pseudocodice dell'algoritmo 2.

Algorithm 2: Procedura di immissione query e ricerca di un peer di livello 0 eseguita dal peer $n_{creator}$

Input: Il livello L_i di inserimento del peer $n_{creator}$, VL di $n_{creator}$, *PoligonoQuery* che contiene i valori Min e Max per ogni attributo della query, il valore scelto dall'utente di *StopAtCenter*

```
1  begin
2      Calcola QueryID con una funzione random // il terzo parametro
           corrisponde al campo FollowUp
3      GreedyMSG = CreateGreedyMSG(L_i, ID_{n_creator}, true, StopAtCenter,
       QueryID, PoligonoQuery);
       // Controllo se sono già al livello 0
4      if (L_i == 0) then
5          GreedyMSG.FollowUP = false;
6          ExecuteQueryGreedy(GreedyMSG);
7      end
8      else
           // Propago al livello superiore
9          if (n_creator ∈ VL[L_i].SuperPeer) then
10             GreedyMSG.Livello=GreedyMSG.Livello − 1;
11             ExecuteQueryGreedy(GreedyMSG);
12         end
13         else
14             GreedyMSG.Livello=GreedyMSG.Livello − 1;
15             while risposta SP negativa do
                   // Si suppone che la lista degli SP contenga
                      almeno un SP valido
16                 SP = SelectRandomNode(VL[i].SuperPeer);
17                 Invio GreedyMSG a SP;
18                 Attendo risposta se SP è effettivamente un SuperPeer
19             end
20         end
21     end
22     Attendo risposte alla query;
23     IntersectReply();
24 end
```

Non mi interesso della ricerca di un SuperPeer in caso la lista non sia aggiornata: si suppone che il tempo T descritto nel paragrafo 3.1 sia sufficiente a fare sì che almeno un SP sia valido, inviando il messaggio di tipo Greedy ad ogni peer della lista fino a trovare un SP valido. La funzione IntersectReply mi fonde le liste dei peer che sono risposta alla query per ottenere il risultato finale, eliminando le ripetizioni di peer che sono presenti in più livelli in seguito all'eventuale elezione a SP.

Lo pseudocodice presentato nell'algoritmo 3 illustra la procedura ExecuteQueryGreedy che un peer n_i esegue alla ricezione di un messaggio di tipo Greedy.

Algorithm 3: Procedura ExecuteQueryGreedy eseguita dal peer n_i alla ricezione di un messaggio di tipo GreedyMSG

```
   Input: GreedyMSG, VL di n_i, il punto p_{n_i} al livello L_i
1  begin
2      L_i = EstraiLvl(SearchNewSPeerMSG);
3      if (GreedyMSG.FollowUP==true) e (L_i ≠ L_0) then
4          GreedyMSG.Livello=GreedyMSG.Livello − 1;
           // Se sono SP, non propago ma rieseguo io stesso il
              greedy al livello superiore
5          Invio GreedyMSG ad un SP;
6      end
7      else
           // Estraggo il punto target del greedy in base agli
              attributi del livello L_i
8          Target = EstraiPuntoTargetAOI(L_i, GreedyMSG.PoligonoQuery,
           GreedyMSG.StopAtCenter);
9          if (Target ∈ VL[i].AreaDiVoronoi) then
10             if (p_{n_i} is not in GreedyMSG.PoligonoQuery.L_i) then
11                 Rispondo a n_creator con un messaggio vuoto ed esco ;
12             end
13             n_root=ID_{n_i};
14             ListHitNodes = ListaVuota;
15             RangeMSG=CreateRangeMSG(L_i, ID_{n_root}, ListHitNodes,
               ID_{n_root}, GreedyMSG.QueryID, PoligonoQuery);
16             ExecuteCompass(RangeMSG);
17         end
18         else
19             GreedyMSG.FollowUp=false;
20             Seleziono punto p più vicino a Target tra HTVN, HTCN e
               LLRN;
21             n_j = SelectRandomNode(HTCN, HTVN o LLRN);
22             Invio GreedyMSG a n_j;
23         end
24     end
25 end
```

Lo pseudocodice presentato nell'algoritmo 4 illustra la procedura Execute-Compass che un peer n_i esegue alla ricezione di un messaggio di tipo RangeMSG o se n_i è il peer radice del compass routing per il livello L_i.

Algorithm 4: Procedura ExecuteCompass eseguita da un peer n_i alla ricezione di un messaggio di tipo RangeMSG

Input: *RangeMSG*, *VL* di n_i

```
1  begin
2      L_i = EstraiLvl(RangeMSG);
       // Controllo se il mio punto cade nella AOI del
          livello L_i
3      if (p_{n_i} è esattamente in RangeMSG.PoligonoQuery.get(L_i)) then
4          if (ExactMatch(n_i,i + 1,Max,RangeMSG.PoligonoQuery)==true)
           then
5              RangeMSG.ListHitNodes = RangeMSG.ListHitNodes ⋃ ID_{n_i};
6          end
7          if (VL[L_i]) == null) o (L_i==L_Max) then
8              foreach (n in VL[L_i].Companions) do
9                  PropagateCMSG = CreatePropagateCMSG(L_i,
                   RangeMSG.ID_creator, RangeMSG.QueryID,
                   RangeMSG.PoligonoQuery);
10                 Invio PropagateCompassMSG a n;
11             end
12         end
13         else
14             GreedyMSG = CreateGreedyMSG(L_{i+1}, ID_{n_creator}, false,
               StopAtCenter ,QueryID, PoligonoQuery);
15             ExecuteQueryGreedy(GreedyMSG);
16         end
17     end
18     ListaFigli = CalcolaListaFigli(L_i, RangeMSG.ID_root, ID_{n_i},
       VL[L_i].HTVN);
19     foreach (p in ListaFigli) do
20         peerFiglio = SelectRandomNode(p);
21         if (peerFiglio interseca RangeMSG.PoligonoQuery.get(L_i)) then
22             RangeMSG.ID_sender = ID_{n_i};
23             Invio RangeMSG a peerFiglio;
24         end
25     end
26     if (ListaFigli == ListaVuota) then
27         Rispondo a n_creator con un messaggio contenente
           RangeMSG.ListHitNodes;
28     end
29 end
```

4.2 Inserzione di un nuovo peer: l'implementazione

Dato un peer n_i, l'inserzione del peer nella rete di Hierarchical Voraque consiste in 3 passi fondamentali:

1. Esecuzione del routing greedy per ricercare il punto target $Target$ dove deve essere inserito, partendo dal livello L_0.

2. Inserzione del peer nel punto target, effettuata dal peer gestore chiamato NG_{p_i}: se $Target$ rientra nel raggio di assorbimento del peer gestore, questo reinizia l'algoritmo di inserzione nella rete di livello inferiore, usando il valore degli attributi di n_i che corrispondono agli attributi mappati in questa rete.

3. Controllo delle eventuali elezioni necessarie per mantenere gli insiemi di SP delle reti attraversate a $\log K$ con K la dimesione del cluster del sottoalbero radicato in ognuna di queste reti.

A queste fasi corrispondono una serie di messaggi caratterizzati dai seguenti campi comuni:

- *Livello*: indica il livello di Hierarchical Voraque a cui è destinato il messaggio. É necessario perché il peer destinatario può essere visibile in più di un livello.

Il messaggio di tipo Insert contiene inoltre i seguenti campi:

- ID_{n_i}: indicatore univoco del peer n_i, occorre per poter inviare ad n_i il messaggio di inizializzazione necessario a renderlo visibile nella rete.

- $ValoreAttributi$: la tupla dei valori degli M attributi dell'oggetto che si vuole pubblicare.

- ID_{Greedy}: identificatore univoco del peer che inizia l'algoritmo greedy per un livello L_i diverso da L_0. Si può osservare che questo è sempre un SP del livello L_i. Viene successivamente comunicato al peer n_i per ottenere la lista dei SP del livello L_i e dal peer gestore per poter inviare un messaggio di tipo ReplyInsert a n_greedy affinché inizi il controllo sull'elezione.

Il messaggio di tipo NeighInsert contiene inoltre i seguenti campi:

- ID_{n_i}: identificatore univoco del nuovo peer che deve essere inserito come Voronoi Neighbour. Insieme all'ID viene comunicato anche il sito p_i in cui è inserito il peer.

Il messaggio di tipo ReplyNeighInsert contiene inoltre i seguenti campi:

- ID_{n_j}: identificatore univoco del peer che ha ricevuto il messaggio di tipo NeighInsert.
- VN_{n_i}: lista dei VN di n_j che sono anche VN di n_i. É divisa per sito.
- CN_{n_i}: lista dei CN di n_j che sono anche CN di n_i. É divisa per sito.

Il messaggio di tipo CloseNeigh contiene inoltre i seguenti campi:

- ID_{n_i}: identificatore univoco del nuovo peer che deve essere inserito come Close Neigbour. Insieme all'ID viene comunicato anche il sito p_i in cui è inserito il peer.

I peer che ricevono il messaggio di tipo CloseNeigh, si limitano ad inserire il n_i nel loro insieme dei CN, senza rispondere.

Il messaggio di tipo LongRange contiene inoltre i seguenti campi:

- ID_{n_i}: identificatore univoco del peer n_i mittente che deve essere inserito come Back Long Range Neighbours. Insieme all'ID viene comunicato anche il punto p_i.
- P_{LRN}: il punto target scelto casualmente da n_i come destinazione del long link.

Il messaggio di tipo PropagateLongRange contiene inoltre i seguenti campi:

- ID_{n_i}: identificatore univoco del peer n_i che deve essere inserito come Back Long Range Neighbours. Insieme all'ID viene comunicato anche il punto p_i.
- ID_{n_j}: identificatore univoco del peer n_j che ha ricevuto il messaggio di tipo LongRange per permettere di inviare una risposta.

Viene utilizzato se il punto target P_{LRN} è un cluster di qualunque dimensione maggiore di 1.

Il messaggio di tipo ReplyPropagateLongRange contiene inoltre i seguenti campi:

- ID_{n_k}: identificatore univoco del peer che ha ricevuto il messaggio di tipo PropagateLongRange per farsi aggiungere da n_j nella lista degli LRN_{n_i}.

Il messaggio di tipo ReplyLongRange contiene inoltre i seguenti campi:

- LRN_{n_i}: lista dei nuovi LRN di n_i.

I messaggi di tipo Init e ReplyInit sono omessi: servono soltanto per inizializzare il nuovo nodo n_i e sono inviati e ricevuti da NG_{p_i}.

Il messaggio di tipo ReplyInsert contiene inoltre i seguenti campi:

- ID_n: Identificatore univoco del nuovo peer inserito o eletto SuperPeer. Il campo è nullo se il nuovo peer inserito non è visibile per il livello del destinatario *Livello*.

Viene utilizzato per far avviare a n_{greedy_i} il controllo sull'elezione. Ai livelli superiori può essere aggiornato con l'ID del SuperPeer eletto e usato per far attivare la ripetizione del controllo sull'elezione.

Routing greedy

Lo pseudocodice presentato nell'algoritmo 5 illustra l'algoritmo di routing greedy di un peer n_j visibile in una rete di livello L_i nel sito p_{n_j} che riceve il messaggio di tipo Insert relativo all'inserimento del peer n_i nel punto $Target$ che non cade nel sito p_{n_j}.

Nell'algoritmo 5 si usa la struttura $AreaDiVoronoi$ che contiene i dati sui vertici, i lati e il punto centrale dell'area di Voronoi generata da n_j nel livello L_i per calcolare se il $Target$ è contenuto nell'area di n_j.

Algorithm 5: Procedura ExecuteGreedy per l'inserimento di un peer n_i eseguita da n_j al livello L_i

```
   Input: InsertMSG, ID_{n_j}, VL di n_j
1  begin
2      L_i = EstraiLvl(InsertMSG);
3      Target_x = EstraiAttr1(L_i,InsertMSG);
4      Target_y = EstraiAttr2(L_i,InsertMSG);
5      Target = (Target_x,Target_y);
6      PuntoDistMin = MaxValue ;
7      if (ID_{n_j}.VL[L_i].AreaDiVoronoi not contains Target) then
           // HTVN: Ht contenente l'insieme dei VN di Target
8          HTVN = ID_{n_j}.VL[L_i].InsiemeVoronoiNeighbours;
9          foreach punto P in HTVN do
10             Dist = CalcolaDistanzaEuclidea(P,Target);
11             if PuntoDistMin > Dist then
12                 PuntoDistMin = Dist;
                   // Seleziono a caso il peer a cui propagare il
                      messaggio tra i VN_P
13                 peerScelto = SelectRandomNode(HTVN.get(P));
14             end
15         end
           // HTCN: Ht contenente l'insieme dei CN di Target
16         HTCN = ID_{n_j}.VL[L_i].InsiemeCloseNeighbours;
17         foreach punto P in HTVN do
18             Dist = CalcolaDistanzaEuclidea(P,Target);
19             if (PuntoDistMin > Dist) then
20                 PuntoDistMin = Dist;
                   // Seleziono a caso il peer a cui propagare il
                      messaggio tra i CN_P
21                 peerScelto = SelectRandomNode(HTCN.get(P));
22             end
23         end
           // LLRN: Lista contenente i LRN di Target
24         LLRN = ID_{n_j}.VL[L_i].LongNeighbour;
25         Dist = CalcolaDistanzaEuclidea(P,Target);
26         if (PuntoDistMin > Dist) then
27             PuntoDistMin = Dist;
               // Seleziono a caso il peer a cui propagare il
                  messaggio tra gli LRN_P
28             peerScelto = SelectRandomNode(LLRN.get(P));
29         end
30         Invio InsertMSG al peer di indice peerScelto;
31     end
32 end
```

Inserzione nel punto target

Lo pseudocodice presentato nell'algoritmo 6 illustra la procedura che il peer NG_{p_i} esegue alla ricezione di un messaggio di tipo Insert quando il punto $Target$ cade nel raggio di assorbimento di p_{NG}.

n_{greedy_i} rappresenta il SuperPeer del livello L_i che ha iniziato il routing greedy. La funzione ReplyForElection() invia un messaggio di risposta di tipo ReplyInsert a n_{greedy_i} affinché possa fare il controllo per l'eventuale elezione.

Algorithm 6: Procedura di inserzione eseguita da NG_{p_i} in caso il nuovo peer cada nel nel raggio di assorbimento di p_{NG}

Input: $InsertMSG$, $ID_{NG_{p_i}}$, VL di NG_{p_i}

1 **begin**
2 L_i = EstraiLvl($InsertMSG$);
3 **if** *($VL[L_i]$ == null)* **then**
4 $VL[L_i].Companions.add(n_i)$;
5 **if** *(($VL[L_i].Companions.size() < S$) o ($L_i == L_{Max}$))* **then**
6 Copio i miei 4 insiemi di vicini nelle liste VN_{n_i},CN_{n_i}, LRN_{n_i} e $BLRN_{n_i}$;
7 $Companions_{n_i}$ = Copy($VL[L_i].Companions$);
// Estraggo da Insert il peer che ha iniziato il greedy per questo livello e lo invio a n_i
8 $ID_{n_{greedy}} = InsertMSG.n_{greedy}$;
9 $InitMSG$ = CreateInitMSG(L_i, VN_{n_i}, CN_{n_i}, LRN_{n_i}, $BLRN_{n_i}$, $Companions_{n_i}$, $ID_{n_{greedy}}$);
10 **foreach** *(peer n_j in HTVN, in HTCN, in HTBLRN ed in HTLRN di NG_{p_i})* **do**
11 Invio messaggio di aggiunta peer n_i nei corrispondenti insiemi al livello L_i;
12 **end**
13 **foreach** *(peer n_j in $VL[L_i].Companions - NG_{p_i}$)* **do**
14 Invio messaggio di aggiunta peer n_i nei corrispondenti insiemi dei *Companions* al livello L_i;
15 **end**
16 Attendo risposta dai *Companions* e dai vicini;
17 Invio $InitMSG$ al peer n_i;
18 Attendo risposta peer n_i;
19 **end**
20 **else**
21 CreateNewLevel(L_{i+1}, ID_{n_i});
22 **end**
23 **end**
24 **else**
25 $VL[L_{i+1}].NumPeer = VL[L_{i+1}].NumPeer + 1$;
26 Salvo $InsertMSG.n_{greedy_i}$ in una variabile;
27 $InsertMSG$ = CreateInsertMSG(L_{i+1}, ID_{n_i}, $T_{N'}$, NG_{p_i});
28 ExecuteGreedy($InsertMSG$);
29 Attendo risposta peer n_i;
30 **end**
31 ReplyForElection();
32 **end**

Lo pseudocodice presentato nell'algoritmo 7 illustra la procedura che il peer NG_{p_i} esegue alla ricezione di un messaggio di tipo Insert quando il punto $Target$ non cade nel raggio di assorbimento di p_{NG} e deve essere creata una nuova area di Voronoi nel sito p_i.

Algorithm 7: Procedura CreateNewVArea per creare la nuova area di Voronoi di n_i eseguita NG_{p_i}

```
   Input: InsertMSG, ID_{n_i}, ID_{NG_{p_i}}, VL di NG_{p_i}
 1 begin
 2     L_i = EstraiLvl(InsertMSG);
       // EstraiTarget:  corrisponde alle righe 3-5 del
          greedy
 3     Target = EstraiTarget(L_i,InsertMSG);
 4     ID_N.VL[L_i].AreaDiVoronoi.update(Target);
 5     Inserisco ID_{n_i} in HTVN;
 6     VN_{n_i} = Copy (ID_{NG}.VL[L_i].Companions);
 7     foreach (peer n_{Comp} in ID_{NG}.VL[L_i].Companions − NG_{p_i}) do
 8         Invio messaggio di aggiunta peer n_i nella HTVN di n_{Comp};
 9     end
10     foreach (p_j in HTVN o HTCN confinante con p_i do
11         foreach (n_j in HTVN[p_j] o HTCN[p_j]) do
12             VN_{n_i}.add(n_j);
13             NeighInsertMSG = CreateNeighInsertMSG(L_i, ID_{n_i}, Target);
14             Invio NeighInsertMSG a n_j;
15             ListaReplyNeighInsertMSG.add(ID_{n_j})
16         end
17     end
18     while (ListaReplyNeighInsertMSG.NotEmpty()) do
19         Attendo risposta da un n_j;
20         ListaReplyNeighInsertMSG.remove(ID_{n_j});
21         foreach (n_k in ReplyNeighInsertMSG.VN_{n_i} o in
           ReplyNeighInsertMSG.CN_{n_i}) do
22             if (n_k not in VN_{n_i} or CN_{n_i}) then
23                 VN_{n_i}.add(n_k);
24                 NeighInsertMSG = CreateNeighInsertMSG(L_i, ID_{n_i}, p_i);
25                 Invio NeighInsertMSG a n_k;
26                 ListaReplyNeighInsertMSG.add(n_k);
27             end
28         end
29     end
       // Estraggo da Insert il peer che ha iniziato il
          greedy per questo livello e lo invio a N
30     ID_{N_{greedy}} = InsertMSG.N_{greedy};
31     InitMSG = CreateInitMSG(L_i, VN_{b_i}, CN_{n_i}, null, null, null, ID_{N_{greedy}});
32     Invio InitMSG a N e attendo la risposta;
33 end
```

Lo pseudocodice presentato nell'algoritmo 8 illustra la procedura che il peer NG_{p_i} esegue alla ricezione di un messaggio di tipo Insert quando il punto $Target$ cade nel raggio di assorbimento di p_{NG} e viene raggiunto il valore della costante S di soglia, nel caso in cui la rete non è al livello L_{Max}. La procedura costruisce un nuova nuova rete di Voronoi di livello L_{i+1}.

Algorithm 8: Procedura CreateNewLevel eseguita da NG_{p_i} quando si riceve un messaggio di Insert e viene superata la soglia S

Input: $InsertMSG$, ID_{n_i}, VL di NG_{p_i}, S

1 **begin**
2   Genero le strutture dati per il nuovo livello per il peer NG_{p_i};
3     $InsertList = VL[L_i].Companions \bigcup n_i - NG_{p_i}$;
      `// Se` L_i `== 0 questo valore è null`
4   Salvo $InsertMSG.n_{greedy_i}$ in una variabile;
5     $Target$ = EstraiPuntoTarget(L_{i+1}, $InsertMSG$);
6   Inserisco NG nella nuova rete nel punto $Target$;
7     $VL[L_{i+1}].SuperPeer$ =Copy($VL[L_i].Companions - n_i$);
8   CancelList($VL[L_i].Companions$);
9     $VL[L_{i+1}].NumPeer=S$;
10   **foreach** *(peer n_j in InsertList)* **do**
11         $InsertMSG$ = CreateInsertMSG(L_{i+1}, ID_{n_j}, T_{n_j}, NG_{p_i}, ID_{n_i});
12   ExecuteGreedy($InsertMSG$);
13   Attendo risposta peer n_j;
14   **end**
15   ReplyForElection();
16 **end**

Controllo dell'elezione

Lo pseudocodice presentato nell'algoritmo 9 illustra l'algoritmo di controllo dell'elezione che il SuperPeer n_{greedy} esegue al ricevimento del messaggio di tipo ReplyInsert. ReplyInsert può essere ricevuto dal nodo NG_{p_i} o da un SuperPeer del livello inferiore. Non è specificato il livello perché viene eseguito dai vari peer n_{greedy} dai livelli da L_i a L_0.

Algorithm 9: Procedura ControlloElezione eseguita dal peer N_{greedy} al ricevimento del messaggio di tipo ReplyInsert

```
   Input: ReplyInsertMSG, VL di n_greedy
1  begin
2      L_i = EstraiLvl(ReplyInsertMSG);
       // VL[L_i].NumPeer è già stato aggiornato con
          l'inserimento di n_i
3      if (log (VL[L_i].NumPeer) >VL[L_i].SuperPeer.size()) then
4          Election(L_i, ReplyInsertMSG.ID_{n_i});
5          NewSuperPeer = ReplyInsertMSG.ID_{n_i};
6      end
7      else
8          NewSuperPeer = null;
9      end
       // Controllo se il livello superiore esiste,
          altrimenti propago il messaggio
10     if (L_{i-1} ≠ 0) then
11         ReplyInsertMSG = CreateReplyInsertMSG(L_{i-1}, NewSuperPeer);
12         Invio ReplyInsertMSG a n_greedy salvato del livello superiore;
13     end
14 end
```

La variabile NewSuperPeer indica se è stato eletto un nuovo SuperPeer e ne memorizza l'identificatore. Viene utilizzata per semplificare l'algoritmo di elezione del livello superiore, in modo che se non vale null, il peer di identificatore NewSuperPeer viene eletto SuperPeer anche al livello superiore se necessario. É possibile che il peer stesso sia anche un SuperPeer del livello superiore: in questo caso si occupa lui stesso di eseguire la procedura ControlloElezione per il livello superiore senza inviare messaggi di tipo ReplyInsert.

4.3 Elezione di un nuovo SuperPeer: l'implementazione

Dato un n_{greedy_i} che ha effettuato il controllo sulla necessità di eleggere un nuovo SuperPeer in una rete di livello L_i con esito positivo, l'algoritmo di elezione si svolge in due fasi fondamentali:

- ricerca di un peer da eleggere, la cui esistenza è garantita dall'Osservazione 1.
- elezione del peer a SuperPeer

Alla fase di ricerca e di elezione corrispondono una serie di messaggi caratterizzati dai seguenti campi comuni:

- *Livello*: indica il livello di Hierarchical Voraque a cui è destinato il messaggio. É necessario perché il peer destinatario può essere visibile in più di un livello.

Il messaggio di tipo SearchNewSPeer contiene inoltre i seguenti campi:

- $ID_{n_{greedy_i}}$: identificatore univoco del peer n_{greedy_i} per permettere al destinatario di inviare una risposta.

Viene inviato in broadcast a tutti gli altri SuperPeer della rete di livello L_i da parte di n_{greedy_i}.

Il messaggio di tipo ReplySearchNewSPeer contiene inoltre i seguenti campi:

- $ID_{NewSPeer}$: identificatore univoco del peer da eleggere SuperPeer, in modo che n_{greedy_i} possa contattarlo direttamente. Il campo è vuoto se non si è trovato un peer eleggibile.

I messaggi di tipo NotifyNewPeer e NotifyNewSPeer hanno la stessa struttura che contiene inoltre i seguenti campi:

- $ID_{NewSPeer}$: identificatore univoco del peer eletto.
- $ID_{n_{greedy_i}}$: identificatore univoco del peer mittente per permettere di inviare una risposta.

I messaggi di tipo ReplyNotifyNewPeer, ReplyNotifyNewSPeerm, ReplyElect hanno la stessa struttura che contiene inoltre i seguenti campi:

- ID_{n_i}: identificatore univoco del peer che ha ricevuto il corrispondente messaggio di notifica o di elezione.

Il messaggio di tipo Elect viene omesso, perché come Init si tratta di un messaggio di inizializzazione del nuovo SuperPeer.

Ricerca del peer da eleggere

Lo pseudocodice presentato nell'algoritmo 10 illustra la procedura eseguita da n_{greedy_i} per ricercare il peer da eleggere.

Per semplificare, nello pseudocodice il messaggio di tipo SearchNewSPeer viene mandato a tutti i SuperPeer, ma si può anche ottimizzare evitando di replicare il messaggio ai SuperPeer che cadono nello stesso sito di n_{greedy_i} . La procedura Elect corrisponde alla seconda fase dell'algoritmo di elezione.

Algorithm 10: Procedura Election, eseguita dal SuperPeer n_{greedy_i}, quando riceve un messaggio di tipo ReplyInsert

Input: L_i,$ReplyInsertMSG.ID_n$, $ID_{n_{greedy_i}}$, VL di n_{greedy_i}

```
1  begin
2      NewSuperPeer=null;
       // Controllo se ho già un peer eleggibile nel
          messaggio di ReplyInsert
3      if (ReplyInsertMSG.IDn ≠ null) then
4          NewSuperPeer = ReplyInsertMSG.ID_n;
5      end
6      else
7          foreach (peer n_j in HTVN, in HTCN, in HTBLRN e HTLRN di
           n_greedy_i) do
8              if (VL_L_i.SuperPeer non contiene n_j) then
9                  NewSuperPeer = n_j;
10                 Esco dal ForEach;
11             end
12         end
13         if (NewSuperPeer==null) then
14             foreach (n_j in VL_L_i.SuperPeer) do
15                 SearchNewSPeerMSG = CreateSearchNewSPeerMSG(L_i,
                   ID_n_greedy_i);
16                 Invio SearchNewSPeerMSG a n_j;
17             end
18         end
19         while (NewSuperPeer == null) do
20             Attendo risposta da un n_j;
21             if (ReplySearchNewSPeerMSG.ID_NewSPeer ≠ null) then
22                 NewSuperPeer=ReplySearchNewSPeerMSG.ID_NewSPeer;
23             end
24         end
25     end
26     Elect(L_i-1,NewSuperPeer);
27 end
```

Elezione del SuperPeer

Lo pseudocodice presentato nell'algoritmo 11 illustra la procedura eseguita da un SuperPeer n_i alla ricezione di un messaggio di tipo SearchNewSPeer da parte di n_{greedy_i}.

Algorithm 11: Procedura eseguita da un SuperPeer n_i, quando riceve un messaggio di tipo SearchNewSPeer da parte di n_{greedy_i}

Input: $SearchNewSPeerMSG$, VL di n_i

```
1  begin
2      L_i = EstraiLvl(SearchNewSPeerMSG);
3      NewSuperPeer=null;
4      foreach (peer n_j in HTVN) do
5          if (VL_{L_i}.SuperPeer non contiene n_j) then
6              NewSuperPeer = n_j;
7              Esco dal ForEach;
8          end
9      end
10     ReplySearchNewSPeerMSG = CreateReplySearchNewSPeerMSG(L_i,
       ID_{NewSPeer});
11     Invio SearchNewSPeerMSG a SearchNewSPeerMSG.ID_{n_{greedy_i}};
12 end
```

In particolare, questo algoritmo differisce da quello della procedura Election perché il peer visita soltanto i suoi vicini di Voronoi.

Lo pseudocodice presentato nell'algoritmo 12 illustra l'algoritmo eseguito dal SuperPeer n_{greedy_i} per eleggere effettivamente il nuovo SuperPeer ottenuto nella fase di ricerca.

Algorithm 12: Procedura Elect eseguita dal SuperPeer n_{greedy_i} per eleggere il nuovo SuperPeer n

---

Input: L_{i-1}, l'id del nuovo peer eletto ID_n, VL di n_{greedy_i}

```
1  begin
2      foreach (peer n_j in VL[L_i].SuperPeer - n_j) do
3          NotifyNewSPeerMSG= CreateNotifyNewSPeerMSG(L_i, ID_n,
           ID_{n_j});
4          Invio CreateNotifyNewSPeerMSG al peer n_j;
5      end
6      Attendo risposte dei SuperPeer;
7      VL[L_i].SuperPeer.add(ID_{NewSuperPeer});
8      foreach (peer n_j in HTVN, HTCN, HTBLRN e HTLRN del livello
       L_{i-1}) do
9          NotifyNewPeerMSG= CreateNotifyNewPeerMSG(L_i ,ID_n, ID_{n_j});
10         Invio CreateNotifyNewPeerMSG al peer n_j;
11     end
12     Attendo risposte vicini di Voronoi del livello L_{i-1};
13     Copio i vicini del livello L_{i-1} in VN_n, CN_n, BLRN_n e LRN_n;
14     ListSuperPeer_n = Copy(VL[L_i].SuperPeer);
15     ListSuperPeer_n = Copy(VL[L_{i-1}].SuperPeer);
16     ElectMSG = CreateElectMSG(L_{i-1}, VN_n, CN_n, BLRN_n, LRN_n, P,
       ListSuperPeer_n, VL[L_i].NumPeer);
17     Invio ElectMSG al peer n;
18     Attendo risposta da n;
19 end
```

In particolare, si osserva che nel messaggio di Elect si passa la lista dei SuperPeer del livello superiore. Al termine della procedura n_{greedy_i} invia un messaggio di tipo ReplyInsert al livello superiore se non è L_0 al peer $n_{greedy_{i-1}}$ che contiene l'ID del SuperPeer eletto.

Capitolo 5

Risultati sperimentali e strumenti usati

In questo capitolo vengono valutate la **scalabilità** della soluzione proposta all'aumentare del numero dei peer di una rete costruita secondo l'architettura del sistema Hierarchical Voraque e il costo effettivo delle operazioni eseguibili su di essa (la metrica viene descritta nel paragrafo 5.2), descritte nel capitolo 3: l'inserzione di nuovi peer e la risoluzione di range query multiattributo. Per poter effettuare la valutazione, è stato implementato un prototipo del sistema tramite il software di simulazione PeerSim[36].

Per poter gestire la costruzione e la modifica geometrica di diagrammi bidimensionali di Voronoi in modo semplice e sufficientemente efficiente, abbiamo scelto di utilizzare la libreria Vast[37]. Nelle prossime due sezioni viene data una breve descrizione dei due strumenti utilizzati per l'implementazione e valutazione di Hierarchical Voraque.

5.1 Gli strumenti utilizzati: PeerSim e Vast

5.1.1 PeerSim: caratteristiche e descrizione

PeerSim[36] è una libreria per la simulazione di reti P2P implementata con il linguaggio Java. La libreria contiene un simulatore che consente di creare dei nodi virtuali, il comportamento dei quali è personalizzabile dal programmatore. Tale libreria è stata utilizzata per la realizzazione di applicazioni per la simulazione di sistemi peer-to-peer per scopi differenti, quali la creazione di overlay per social network cooperativi. La simulazione è completamente sequenziale a thread unico: grazie a PeerSim un'unica macchina è in grado di simulare un'intera rete P2P,

senza la complessità relativa alla sincronizzazione e allo scambio di messaggi su una rete fisica.

Peersim: Architettura dei nodi

In PeerSim, una rete non è altro che un vettore di nodi virtuali: ogni nodo è rappresentato da un'istanza della classe Node e possiede un ID univoco intero che lo identifica. I nodi della rete sono tutti direttamente accedibili dal simulatore e raggiungibili dagli eventi generati da PeerSim, quindi la topologia di base è rappresentata da una rete completamente connessa. I nodi della rete virtuale costruita tramite PeerSim, interagiscono tra di loro inviandosi degli **eventi**, ai quali reagiscono con tempistiche determinate da una linea temporale. Ogni evento è caratterizzato da un istante in cui viene ricevuto dal destinatario e uno in cui viene effettuata l'attivazione del destinatario stesso, il quale provvederà a gestire il messaggio tramite una apposita procedura. L'istante che caratterizza l'evento dipende sia dal momento della simulazione in cui viene creato, sia dalla latenza virtuale che PeerSim può aggiungere nella comunicazione virtuale tra mittente e destinatario. Questa latenza può essere personalizzata al momento della configurazione del simulatore.

Ogni singolo nodo è composto da tre parti principali che ne determinano il funzionamento e che sono organizzate a livelli: un livello di tipo Linkable, un livello di tipo Protocol ed un livello di tipo Transport.

Il livello Linkable contiene i metodi che permettono di gestire i collegamenti con gli altri nodi della rete, i cosiddetti vicini: con esso si possono simulare i link esistenti tra i nodi fisici in modo che non si possa inviare eventi tra nodi che non hanno un link, oppure simulare, ad un determinato istante della linea temporale, il fallimento di un nodo. Una classe che implementa questa interfaccia permette quindi di variare la topologia di base della rete di PeerSim, anche dinamicamente.

Il livello Protocol implementa il nucleo applicativo del nodo. Un nodo può avere molti livelli di tipo Protocol: ognuno di questi implementa un diverso tipo di protocollo virtuale. Ogni classe che implementa l'interfaccia di tipo Protocol contiene un metodo, invocato dal simulatore, che effettua la consegna di un determinato evento ad un certo nodo destinatario. Il corpo di tale metodo implementa la gestione dell'evento. In Hierarchical Voraque, ogni evento rappresenta un messaggio scambiato tra due nodi, i parametri dell'evento rappresentano i corrispondenti parametri del messaggio, mentre il corpo del metodo invocato alla ricezione dell'evento rappresenta la gestione del messaggio.

Il livello Transport implementa il sistema di comunicazione del nodo, con il quale è possibile immettere eventi all'interno della rete: in particolare, è possibile, per esempio, simulare la latenza dei collegamenti fisici o specificare particolari protocolli di invio o di ricezione dei messaggi per simulare meglio la realtà

dell'applicazione che vogliamo simulare. Una classe che implementa questa interfaccia implementa il metodo Send con cui viene simulata la spedizione di un messaggio a livello fisico.

I tre livelli sono interamente personalizzabili dal programmatore tramite l'estensione delle interfacce che virtualizzano i metodi da implementare.

In PeerSim il simulatore ha un ridotto numero di compiti: gestisce la linea temporale, permettendo la generazione e l'attivazione degli eventi, effettua la creazione dei nodi che devono essere presenti all'avvio della simulazione ed interpreta il file di configurazione della simulazione, per caricare le classi che devono essere utilizzate e che implementano i vari protocolli e per inizializzarne i parametri della simulazione ai valori richiesti dall'utente.

Esiste la possibilità per l'utente di aggiungere dei controllori esterni (che implementano un'interfaccia di tipo Control) per agire sui componenti della rete. Questi possono essere attivati ad intervalli regolari o in fase di inizializzazione, ed hanno una visione globale della rete in quanto possono raccogliere dati relativi a qualunque componente della rete. Gli elementi di tipo Control vengono solitamente utilizzati per raccogliere i dati che descrivono l'evolversi della simulazione, estraendo valori contenuti nei nodi, oppure possono intervenire nella simulazione, generando eventi esterni alla rete.

PeerSim: Simulazione a cicli e ad eventi

Il simulatore permette un ulteriore livello di personalizzazione nella gestione della linea temporale, mettendo a disposizione due tipologie di simulazione: CycleBased ed EventBased.

La simulazione EventBased prevede la strutturazione a scambio di eventi tra nodi descritta nel paragrafo precedente, mentre la simulazione CycleBased opera tramite un modello semplificato di simulazione in cui ogni nodo viene attivato ad istanti periodici definiti **cicli**. Per quest'ultima tipologia viene specificato nel file di configurazione il numero dei cicli da svolgere, il loro periodo e l'istante della simulazione in cui deve partire il primo ciclo.

Il simulatore si occupa di attivare alla fine di ogni ciclo ciascun nodo della rete, per poi attivare i controlli elencati nel file di configurazione.

Questo modello semplificato consente al programmatore di non ridefinire il livello di tipo Transport all'interno del nodo.

È possibile infine effettuare simulazioni di tipo ibrido, ovvero che consentono contemporaneamente sia l'esecuzione CycleBased che quella EventBased. In questo caso, sulla linea temporale sono presenti sia le chiamate periodiche dovute all'esecuzione dei cicli, sia le registrazioni degli eventi scambiati tra i nodi. I due modelli non sono in conflitto, dato che i nodi, per poter essere attivati dalle due tipologie di simulazione, devono implementare interfacce diverse.

5.1.2 VAST: Voronoi-based Adaptive Scalable Transfer

La libreria VAST[37] ha l'obiettivo di fornire una gestione scalabile ed efficiente di diagrammi di Voronoi a due dimensioni, con inserimento dinamico dei nodi. In questa tesi è stata usata la versione Java della libreria, che fornisce una semplice interfaccia che permette di inserire e rimuovere nodi, ricavare i confini di un'area di Voronoi ed estrarre l'elenco dei vicini di Voronoi di un dato nodo.

Parte fondamentale della libreria è la classe SFVoronoi che implementa tutte le operazioni atte alla manipolazione del diagramma di Voronoi. Per aggiornare dinamicamente la struttura del diagramma di Voronoi, VAST usa l'algoritmo proposto da Fortune[35] che permette di ricalcolare un intero diagramma con un costo pari ad $O(N * \log N)$ in tempo e $O(N)$ in spazio, con N numero totale di siti della rete, e sfrutta un albero di ricerca binario, messo a disposizione in SFVoronoi con un'istanza della struttura Java TreeMap.

5.2 Hierarchical Voraque: misurazioni effettuate

In questo paragrafo descrivo brevemente che cosa è stato misurato ed i motivi della misurazione.

Ricordiamo i principali obiettivi che ci siamo posti nella costruzione del sistema di Hierarchical Voraque:

- risolvere range query multiattributo, superando il problema delle reti di Voronoi multidimensionali, in cui il numero dei siti vicini del sito in cui è inserito un peer cresce in modo esponenziale.
- scalare in presenza di cluster di peer, la cui dimensione dipende dalla distribuzione del valore degli attributi: più peer hanno il valore degli attributi identico, più un cluster cresce di dimensione. Questo viene risolto definendo un insieme di SP per ogni cluster uguale a $\log K$, con K grandezza del cluster.
- minimizzare il numero dei peer attraversati e dei messaggi scambiati durante la risoluzione di una query sfruttando la selettività degli attributi.

Per raggiungere questi obiettivi, Hierarchical Voraque è stato progettato sotto forma di rete gerarchica, rappresentata da un albero i cui nodi rappresentano reti di Voronoi bidimensionali. Le reti allo stesso livello dell'albero mappano una diversa coppia degli attributi gestiti da Hierarchical Voraque. Siti diversi di una rete di un qualsiasi livello L_i generano reti diverse al livello inferiore qualora vi sia associato un cluster di dimensione maggiore della costante di soglia S: in ognuno di questi siti sono mappati i SP della rete di livello inferiore.

Nei nostri test vogliamo far vedere che questi obiettivi sono stati raggiunti, in accordo ai risultati teorici di complessità degli algoritmi di inserzione e di risoluzione delle range query multiattributo calcolati nel paragrafo 3.3.5.

La metrica utilizzata nei test è il **numero di messaggi** scambiati tra i vari peer della rete, simulati in PeerSim da eventi che incapsulano i parametri dei messaggi. Infatti, il tempo speso a comunicare sulla rete è preponderante sul tempo speso localmente dai vari peer, quindi quest'ultimo viene trascurato.

Per poter misurare il numero di messaggi scambiati, è stato generato per ogni test un file di log dove ogni riga rappresenta la registrazione di un messaggio. I messaggi sono raccolti per mezzo di un metodo apposito che incapsula il metodo che invia un messaggio a livello Transport di PeerSim. Per ogni messaggio sono registrati:

1. L'ID dell'operazione di inserimento o di risoluzione di range query di cui il messaggio fa parte.

2. L'ID e il livello del peer mittente e del peer destinatario.

3. Il tipo del messaggio. Come possiamo vedere nel capitolo 4, nell'implementazione del sistema vengono utilizzati un certo insieme di tipi di messaggi. In questa tesi ci interessa solo distinguere tra:

   (a) i messaggi inviati durante l'esecuzione del routing greedy relativo all'inserzione di un nuovo peer, che chiameremo **messaggi per la fase di routing greedy dell'inserzione**

   (b) i messaggi inviati durante l'inserzione nel punto target e per le eventuali elezioni, che chiameremo **messaggi di gestione della rete per l'inserzione**

   (c) i messaggi inviati durante l'esecuzione del routing greedy relativo alla risoluzione delle range query multiattributo, che chiameremo **messaggi per la fase di routing greedy di una range query multiattributo**

   (d) i messaggi inviati durante l'esecuzione del compass routing relativo alla risoluzione delle range query multiattributo, che chiameremo **messaggi per la fase di compass routing di una range query multiattributo**

Successivamente abbiamo trattato automaticamente i risultati registrati nei vari file tramite un programma Java scritto ad hoc per ottenere i grafici mostrati nel paragrafo successivo.

Per poter verificare i risultati teorici ottenuti e controllare se sono stati raggiunti gli obiettivi, abbiamo testato la complessità in numero di messaggi dell'algoritmo

di inserzione e di quello di risoluzione di una range query. Variando il numero dei peer che compongono la rete, possiamo dare una stima della scalabilità raggiungibile dai due algoritmi. In questo modo è possibile confrontare i risultati ottenuti con quelli di altri approcci di tipo P2P come MAAN[7] o i Diagrammi di Voronoi multidimensionali, che risolvono le range query multiattributo.

Abbiamo utilizzato due diverse distribuzioni dei valori degli attributi degli oggetti pubblicati dai peer per simulare sia la creazione di cluster di peer di varie dimensioni sia per poter meglio approssimare vari casi di studio reali, come descritto nel paragrafo 3.1:

- una **distribuzione uniforme random**, in cui i peer pubblicano oggetti i cui valori degli attributi si distribuiscono casualmente ma uniformemente. In questo caso la rete è rappresentata come un albero completamente bilanciato. Per poter valutare il caso più costoso sia per l'inserzione che per la risoluzione delle query, abbiamo fatto in modo che tutti i peer si inserissero nelle reti di livello massimo. Per simulare ciò, data una rete di Hierarchical Voraque con numero di livelli L, abbiamo fissato un numero di siti P_i per ogni rete in modo tale che $\frac{N}{\prod_{i=0}^{L} P_i} \geq 1$. Per esempio, data una rete con 3 livelli e con un numero di peer $N = 1024$, se avessimo 8 siti nella rete di livello L_0 e altrettanti per ogni rete al livello L_1, ogni rete di livello L_2 avrebbe 16 peer, cioè $\frac{1024/8}{8}$. Questo significa per esempio che a livello L_0 ogni sito genera un cluster di dimensione $1024/8 = 128$ ed ha un numero di peer visibili uguale a 7, cioè $\log 128$.

- una **distribuzione di tipo power-law** [34], in cui i peer pubblicano oggetti i cui valori degli attributi si distribuiscono casualmente ma non uniformemente. La distribuzione che segue una funzione power-law ha la particolarità di distribuire gli oggetti con alta probabilità su un numero molto limitato di punti e con probabilità molto bassa su tutto il resto dei punti. La costante α della distribuzione regola questa probabilità: più è alta, più gli oggetti si concentrano su pochi siti con altissima probabilità. In questo caso la rete è rappresentata da un albero tanto più sbilanciato quanto più è alto il valore di α. Per simulare ciò, abbiano fissato ancora una volta un numero di siti P_i uguale per tutte le reti e abbiamo distribuito secondo la power-law i peer su questi siti fissando un valore di α abbastanza alto in modo tale che almeno nel sito della rete di Voronoi a livello L_0 con le coordinate corrispondenti al valore degli attributi più probabile si formasse un cluster di dimensione tale che tutti i peer si inserissero al livello massimo della rete, in modo da poter valutare il caso più costoso sia per l'inserzione che per la risoluzione delle query.

Per i test sulle query abbiamo infine scelto anche di valutare la loro complessità con diversi valori di selettività degli attributi. La selettività degli attributi viene simulata costruendo le varie AOI locali come rettangoli grandi una percentuale s_i dell'area totale delle reti corrispondenti. La posizione di questi rettangoli viene scelta casualmente da rete a rete. La lunghezza dei lati delle AOI locali alle reti dello stesso livello L_i rappresentano i range degli attributi di indice $[A_{2i}, A_{2i+1}]$ da impostare come parametro di ricerca nella query. Variando la selettività, otteniamo AOI più grandi, quindi statisticamente un numero maggiore di siti da visitare per ogni livello.

5.3 Parametri dei test e risultati

In questo paragrafo descrivo i test, dividendoli tra quelli effettuati per valutare il costo dell'inserimento di un nuovo peer nel sistema e quelli effettuati per valutare il costo della risoluzione delle range query multiattributo. Tutti i test hanno in comune i seguenti parametri:

1. Un numero di attributi uguale a 6, quindi un massimo di 3 livelli possibili. Ogni attributo è stato fissato di tipo intero.

2. La costante di soglia S fissata al valore 4, molto piccolo rispetto al numero dei peer utilizzati nei test.

3. La grandezza di tutte le reti bidimensionali che compongono i vari livelli di Hierarchical Voraque fissata a $1000 * 800 = 800000$ punti. Questo significa che ogni attributo dispari ha dominio $[0, 1000]$ e ogni attributo pari ha dominio $[0, 800]$.

4. il raggio di assorbimento fissato a 1.

5. il raggio dei Close Neighbour, che è fissato in base alla grandezza delle reti in modo da limitare il numero dei CN di ogni peer: nel nostro caso esso vale 15.

Passiamo ora ad esaminare in dettaglio i test eseguiti.

5.3.1 Risultati dei test sull'algoritmo di inserimento

I test sull'algoritmo di inserimento sono stati effettuati su reti strutturate secondo Hierarchical Voraque con un numero di peer variabile tra 1000 e 10000, a passi di 1000 peer.

Ricordiamo che l'inserimento di un peer in una rete strutturata secondo Hierarchical Voraque è composto da tre fasi:

1. Esecuzione del routing greedy a partire da un peer n_{boot} del livello L_0 per giungere fino al sito di livello L_{Max} che contiene il peer gestore dell'inserimento.

2. Inserzione da parte del peer gestore nella rete di livello L_{Max} del nuovo peer.

3. Eventuale elezione di un nuovo SuperPeer: il numero di elezioni che si eseguono dopo l'inserzione di un nuovo peer sono al massimo una per ogni livello della rete.

Abbiamo quindi valutato separatamente il numero di messaggi per la fase di routing greedy dell'inserzione di un peer dal numero di messaggi di gestione della rete (fasi corrispondenti ai punti 2 e 3). Questo per verificare che la presenza e la dimensione dei cluster effettivamente non incide sul costo del routing, come indica anche la complessità teorica.

I test di inserimento sono stati effettuati prima simulando una rete con distribuzione uniforme random dei valori degli attributi: per simulare questo comportamento, abbiamo fissato un numero di siti con coordinate distribuite casualmente per ogni rete che compone l'albero di Hierarchical Voraque. Il numero per semplicità è stato fissato uguale per ogni livello. Successivamente si è proceduto a fissare i valori degli attributi dei peer in modo che coincidessero con i valori di questi siti distribuendoli uniformemente e ripetendo questa operazione per ogni livello. Ogni test è stato ripetuto 100 volte, e sono stati presi i risultati ottenuti sull'inserimento degli ultimi 20 peer, mediandoli rispetto al numero dei peer (20) e dei test (100) per ottenere il costo medio di una inserzione. Possiamo notare che il costo di inserirzione degli ultimi è infatti generalmente più alto rispetto a tutti gli altri in quanto si inseriscono in una rete quasi completa.

La tabella 5.1 mostra il numero di siti fissati, che variano con il crescere della grandezza della rete, in modo che tutti i peer si inseriscano nelle reti di livello massimo e che in queste reti in un sito sia mappato un solo peer.

Successivamente è stata simulata una distribuzione dei valori degli attributi basata su una distribuzione power-law. Per simulare questo comportamento, abbiamo fissato un numero di siti massimo, in questo caso uguale per tutti i test, anche al variare della dimensione della rete. Questo numero è stato scelto uguale a 100. Dopodiché abbiamo distribuito secondo la power-law i peer su questi siti fissando un valore di α abbastanza alto in modo tale che nel sito con le coordinate corrispondenti al valore degli attributi più probabile si formasse un cluster di dimensione tale che tutti i peer si inserissero al livello massimo della rete. Il numero di siti è stato scelto così elevato per fare in modo che un buon numero di siti abbiano una probabilità bassa di essere target di un peer; questo significa che avremo molti siti privi di peer e molti altri con pochi o un solo peer. La variabile α è stata fissata a due valori diversi: $\alpha = 2$ e $\alpha = 3$, che si differenziano per il

Grandezza della rete	Numero di siti	Numero di peer per ogni rete di livello 3
1000	10	10
2000	15	8
3000	20	7
4000	25	6
5000	30	5
6000	35	4
7000	40	4
8000	45	3
9000	50	3
10000	55	3

Tabella 5.1: Numero di siti fissati al variare della grandezza della rete

fatto che rispettivamente la rete viene rappresentata da un albero meno sbilanciato ed uno molto più sbilanciato. Per $\alpha = 3$ quasi tutti i peer si raggruppano in un unico cluster e la rete è formata da pochissimi siti. Anche in questo caso i test sono stati ripetuti 100 volte ed è stata presa la media dei messaggi degli ultimi 20 peer inseriti.

Inserimento con distribuzione uniforme

In figura 5.1 vediamo l'andamento del numero medio dei messaggi per la fase di routing greedy dell'inserzione di un peer in una rete generata in accordo ad una distribuzione uniforme dei valori degli attributi. Siamo nel caso pessimo, in quanto abbiamo una rete rappresentata da un albero completamente bilanciato. Secondo la complessità teorica il numero di messaggi scambiati dovrebbe essere al massimo $\log N$. Per esempio, nel caso di $N = 1000$ avremmo dovuto avere circa $\log 1000 \approx 10$, mentre ne risultano circa $3,6$. Lo stesso fenomeno si ripete per tutti gli altri valori di N. Il valore quindi rientra molto al disotto del limite di complessità teorico, potendo concludere che il costo della fase di routing è molto basso.

L'andamento del grafico mostra un aumento del numero di messaggi all'aumentare della dimensione della rete molto lento, mantenendone il numero sempre al di sotto del limite di complessità teorico. Questo significa che l'algoritmo di routing greedy raggiunge un buon livello di scalabilità.

In figura 5.2 vediamo l'andamento del numero medio dei messaggi diversi dal greedy per una rete generata in accordo ad una distribuzione uniforme. In questo caso, possiamo notare che sono in media più numerosi rispetto ai messaggi

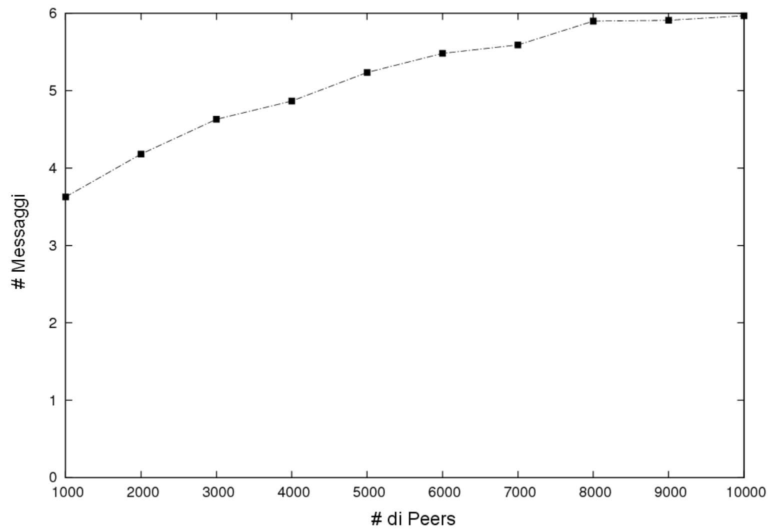


Figura 5.1: Numero medio di messaggi del routing greedy in una rete generata in accordo ad una distribuzione uniforme

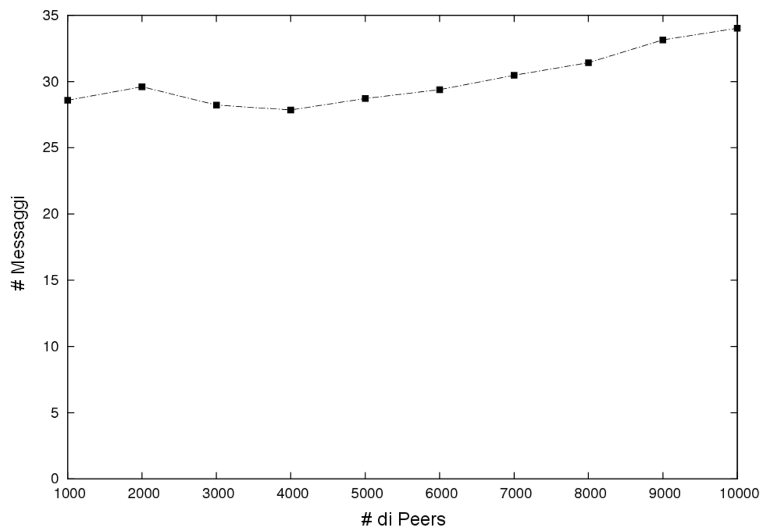


Figura 5.2: Numero medio di messaggi di gestione della rete in una rete generata in accordo ad una distribuzione uniforme

del routing greedy: questo perché le costanti moltiplicative dietro la complessità dell'elezione e dell'inserzione nel punto target di ogni peer non sono trascurabili.

Comunque il valore di circa 30 è molto piccolo, ma soprattutto possiamo vedere che è **costante** con l'aumentare della dimensione della rete. Questo rispecchia la complessità teorica che era stata calcolata nel caso di distribuzione uniforme: il piccolo aumento da 30 con $N = 1000$ a 32 con $N = 10000$ dipende dal fatto che il costo ammortizzato delle elezioni non è costante, ma ha un andamento comunque sub-logaritmico, e quindi praticamente trascurabile.

Possiamo concludere che il costo di inserimento di un peer in una rete costruita secondo l'architettura del sistema Hierarchical Voraque nel caso di una distribuzione uniforme a cluster dei peer risulta migliore della complessità teorica, ma soprattutto che la scelta di costruire una rete gerarchica con un numero logaritmico di SP per ogni livello è risultata ottimale per raggiungere un risultato equivalente a quello di una rete di Voronoi bidimensionale, anche se i due sistemi non sono confrontabili per il diverso numero di attributi gestibili.

Inserimento con distribuzione power-law

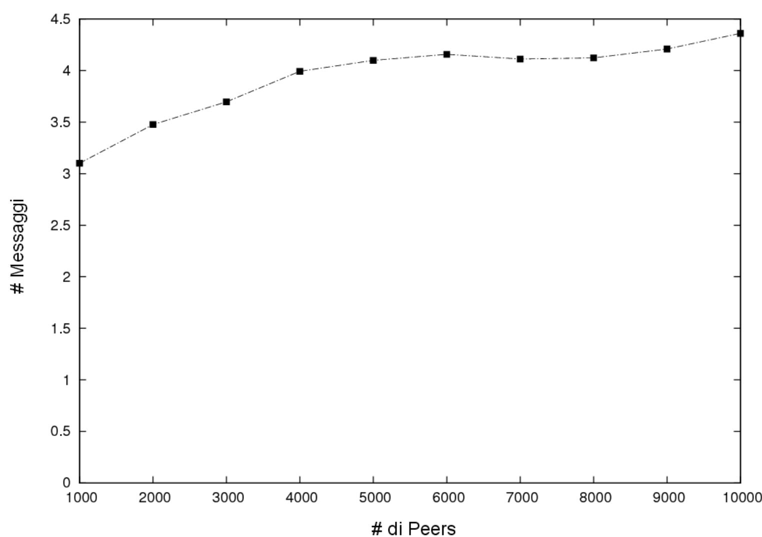


Figura 5.3: Numero medio di messaggi del routing greedy in una rete generata usando una distribuzione Power-law: $\alpha = 2$

Le figure 5.3 e 5.4 mostrano l'andamento del numero medio dei messaggi del routing greedy per una rete che sfrutta una distribuzione power-law dei peer con due diversi valori della costante α. Osserviamo come, rispetto ai messaggi di routing greedy del caso precedente, aumentando il valore della costante α il numero di messaggi diminuisca. Ciò accade per qualsiasi valore di N: questo è dovuto

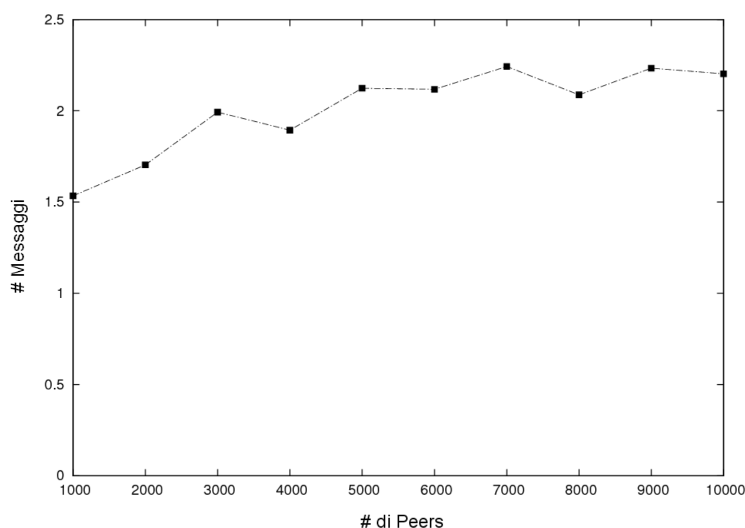


Figura 5.4: Numero medio di messaggi del routing greedy in una rete generata usando una distribuzione Power-law: $\alpha = 3$

al fatto che i peer si concentrano su un numero sempre più piccolo dei siti fissati. Infatti, secondo la distribuzione power-law, più è alto il valore di α, più è alta la probabilità che i peer si concentrino su un unico sito, riducendosi la probabilità di essere mappati su siti con coordinate diverse. Questo significa che molti dei siti fissati non hanno peer mappati, quindi il loro numero totale effettivo nella rete diminuisce. Sappiamo che il greedy ad ogni passo seleziona un sito vicino e sceglie casualmente un peer tra quelli visibili in quel sito: di conseguenza si ottiene che se il numero dei siti totali della rete diminuisce, anche il numero di messaggi del routing greedy diminuisce di conseguenza, ottenendo il risultato illustrato.

L'andamento del grafico mostra un aumento del numero di messaggi all'aumentare della dimensione della rete anche in questi casi molto lento, mantenendone il numero sempre al di sotto del limite di complessità teorico. Questo significa che l'algoritmo di routing greedy raggiunge un buon livello di scalabilità anche per reti rappresentate da alberi sbilanciati, confermato anche su diversi gradi di sbilanciamento.

Le figure 5.3 e 5.4 mostrano l'andamento del numero medio dei messaggi di gestione della rete per una rete che sfrutta una distribuzione power-law dei peer con due diversi valori della costante α. Come ci aspettavamo anche per questo tipo di distribuzione possiamo notare che sono in media più numerosi rispetto ai messaggi del routing greedy. In particolare, il costo della gestione della rete inizia

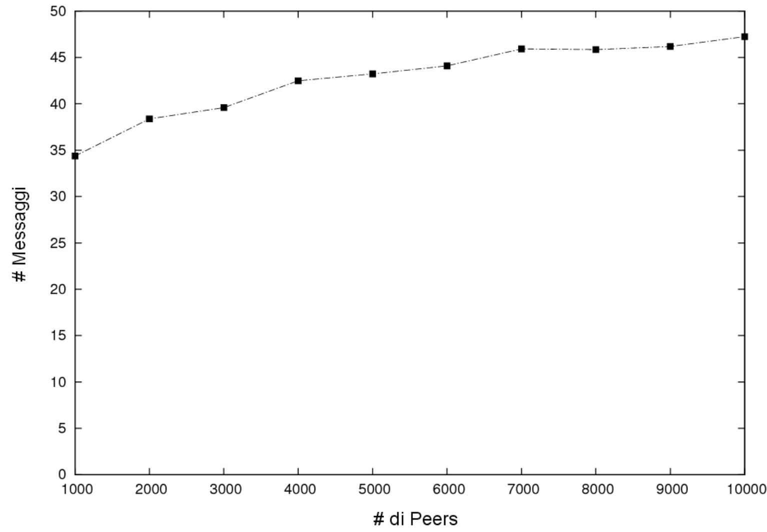


Figura 5.5: Numero medio di messaggi di gestione della rete in una rete generata usando una distribuzione Power-law: $\alpha = 2$

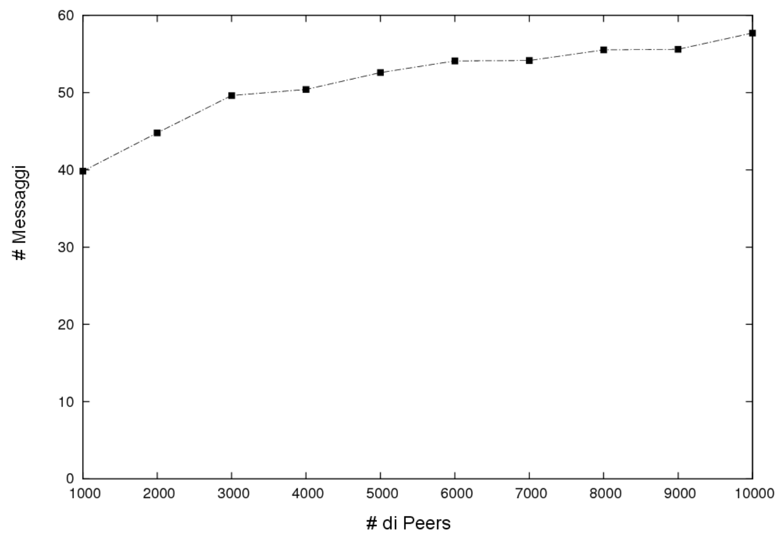


Figura 5.6: Numero medio di messaggi di gestione della rete in una rete generata usando una distribuzione Power-law: $\alpha = 3$

ad aumentare, mentre il peso del routing greedy diminuisce. Inoltre, in accordo al valore della complessità teorica, la curva prende un andamento logaritmico con

l'aumentare della grandezza della rete, e non costante come nel caso di una rete generata secondo una distribuzione uniforme. Questo perché la maggior parte dei peer va ad inserirsi in un unico sito al livello massimo formando un cluster che tende a crescere di dimensione con l'aumentare di N. Il costo di questa operazione dipende dal numero di SP eletti nel cluster in questione, che a sua volta è logaritmico rispetto alla sua dimensione, quindi al massimo otteniamo $O(\log N)$.

Possiamo concludere che il costo di inserimento di un peer in una rete di strutturata secondo l'architettura di Hierarchical Voraque nel caso sia utilizzata una distribuzione power-law dei peer risulta migliore della complessità teorica, ma soprattutto si raggiunge un risultato equivalente a quello di una rete di Voronoi bidimensionale.

5.3.2 Risultati dei test sull'algoritmo di risoluzione delle range query multiattributo

I test sull'algoritmo di risoluzione di range query multiattributo sono stati effettuati su reti con un numero di peer variabile tra 1000 e 10000, a passi di 1000 peer.

Ricordiamo che la risoluzione di una range query multiattributo in una rete strutturata secondo l'architettura di Hierarchical Voraque è composto da tre fasi:

1. Raggiungimento di un peer di livello L_0 da cui iniziare la risoluzione della range query.

2. Esecuzione del routing greedy per raggiungere il sito centrale dell'AOI del livello L_0.

3. Costruzione dello spanning tree per visitare tutti i siti dell'AOI locale utilizzando l'algoritmo di compass routing: per ogni sito che genera un cluster di dimensione > 3, viene selezionato un peer che, in qualità di SuperPeer della rete sottostante, ripete i punti 1 e 2 sulla rete di livello inferiore per propagare la query. Se si raggiunge il massimo livello L_2 e un sito genera un cluster di dimensione > 3, allora esso manda direttamente un messaggio di risposta anche per conto di tutti gli altri peer del cluster, perché tutti hanno lo stesso valore dei 6 attributi.

Abbiamo quindi valutato separatamente il numero di messaggi per le fasi di routing greedy dal numero di messaggi delle fasi di compass routing, escludendo i messaggi di risposta al peer che ha immesso la query nel sistema. Abbiamo scelto di raggiungere sempre il centro delle AOI per ogni query. Inoltre per ogni query il peer che la genera è stato scelto a caso.

In particolare, i test sulle query sono stati eseguiti scegliendo la selettività degli attributi seguendo il metodo descritto nel paragrafo precedente, cioè fissando le AOI per una determinata coppia di attributi ad una percentuale dell'area totale, che ricordiamo è uguale per tutte le reti di tutti i livelli nei test da noi effettuati. Abbiamo però semplificato il problema scegliendo un uguale valore di selettività per tutte le coppie di attributi mappate ai vari livelli. I valori scelti sono 5%, 10% e 20%.

I test sono stati effettuati prima simulando l'esecuzione una range query multiattributo casuale per ognuno dei 3 diversi valori di selettività suddetti su una rete con distribuzione uniforme random dei valori degli attributi, creata come descritto nel paragrafo precedente. Abbiamo ripetuto questi test 100 volte con query generate casualmente e poi mediato il numero di messaggi ottenuto per il numero di query(100) per ottenere il costo totale di risoluzione di una range query multiattributo, per ognuno dei parametri di selettività.

Successivamente è stata simulata l'esecuzione di una range query multiattributo per ognuno dei 3 diversi valori di selettività su una rete con distribuzione dei valori degli attributi che segue una power-law. La variabile α è stata fissata a due valori diversi: $\alpha = 2$ e $\alpha = 3$, gli stessi valori dei test sull'inserzione. Anche in questo caso abbiamo ripetuto i test 100 volte con query casuali sulla stessa rete e poi abbiamo fatto la media dei valori ottenuti rispetto al numero delle query(100) per ottenere il costo totale di risoluzione di una range query multiattributo, per ognuno dei parametri di selettività.

Query su rete generata con una distribuzione uniforme

In figura 5.7 vediamo l'andamento del numero medio dei messaggi del routing greedy per le query immesse in una rete generata con una distribuzione uniforme. La complessità totale del routing greedy per la risoluzione delle range query multiattributo è uguale alla somma dei routing greedy effettuati per raggiungere il sito centrale di ogni AOI: data la AOI di una certa rete di livello L_i raggiunta dalla query, il numero di livelli sottostanti a cui propagare la query dipende dal numero dei siti contenuto nell'AOI, cioè dalla selettività composta degli attributi del livello. Otteniamo, in accordo alla complessità teorica, che il numero di questi siti equivale a moltiplicare la selettività composta degli attributi per il numero dei siti totali fissati per ogni livello. Ripetiamo questo procedimento per ogni rete visitata dalla query. Otteniamo quindi che il numero di messaggi totale per il routing greedy di una query aumenta linearmente con l'aumentare della selettività.

L'andamento del grafico, per ogni tipo di selettività, mostra un aumento del numero di messaggi all'aumentare della dimensione della rete molto lento, mantenendone il numero sempre al di sotto del limite di complessità teorico. Infatti

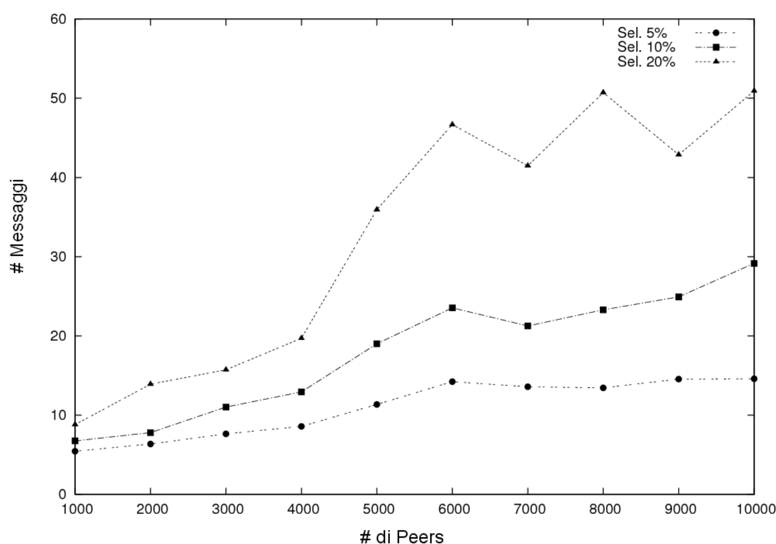


Figura 5.7: Numero medio dei messaggi del routing greedy per query, ottenuto su una rete generata usando una distribuzione uniforme

la complessità teorica indica che il limite superiore sul numero di messaggi è logaritmico sulla dimensione della rete e dipendente dal numero dei siti fissati, che nel nostro caso aumentano con la dimensione della rete.

Questo significa che l'algoritmo di routing greedy raggiunge un buon livello di scalabilità. Per esempio, rispetto a MAAN [7] il tempo del routing è confrontabile, ma i costi di inserimento sono sensibilmente inferiori.

Come si ripresenterà anche nei successivi grafici sulla distribuzione power-law, nelle 2 selettività più alte il grafico, che dovrebbe sempre tendere a crescere, ha invece dei picchi e dei cali, come si può vedere in $N = 7000$ e in $N = 8000$. Molto probabilmente questo è dovuto alla distribuzione dei peer sulle reti di livello 3: come possiamo vedere nella tabella, man mano che la rete aumenta di dimensione, la differenza tra il numero di siti fissato ed i peer inseriti nelle reti di livello 3 aumenta. Data la corrispondenza tra siti e peer al livello massimo, otteniamo che il numero di siti effettivi totali della rete diminuisce con l'aumentare della dimensione della rete. Conseguentemente il numero di messaggi necessari per eseguire il routing greedy in una qualunque rete del livello massimo diminuisce e sommandoli a quelli necessari per eseguire il routing ai livelli superiori, possiamo ottenere un valore minore di messaggi anche se la rete aumenta di dimensione.

In figura 5.8 vediamo l'andamento del numero medio dei messaggi del routing compass per le query immesse in una rete generata con una distribuzione uniforme. Premettiamo che con il meccanismo che utilizziamo per calcolare la

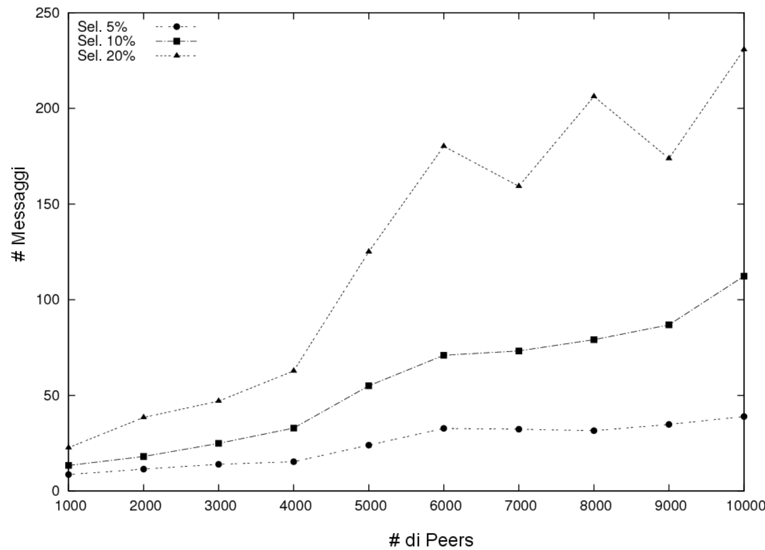


Figura 5.8: Numero medio di messaggi del routing compass per query, ottenuto su una rete generata usando una distribuzione uniforme

selettività degli attributi, non possiamo sapere a priori quanti siti cadono all'interno di una determinata AOI, quindi il grafico presenta un andamento piuttosto altalenante. Comunque il numero totale dei messaggi per query è, in accordo alla complessità teorica, linearmente dipendente dalla dimensione della rete, dal prodotto delle selettività degli attributi e dal numero di livelli. Questo significa che per reti con un maggior numero di attributi, la selettività sarebbe risultata discriminante per la scalabilità del sistema.

Possiamo concludere quindi che anche per il routing compass otteniamo un buon livello di scalabilità.

Questa è confrontabile con MAAN, dove otteniamo che per N peer, il routing con attributo dominante ha complessità $O(N * S_{min})$ dove quest'ultima è la selettività minima di tutti gli attributi. Effettuare una query su una rete generata con una distribuzione uniforme costituisce il caso pessimo, in quanto i cluster risultano tutti della stessa dimensione. In questo caso durante il compass routing vengono visitati mediamente anche un buon numero di siti che non contengono peer che sono risposta per la query. Inoltre le AOI, dato che abbiamo un numero di siti totali della rete maggiore nel caso della distribuzione uniforme rispetto alla distribuzione di tipo power-law come descritto per l'algoritmo di inserimento, comprendono mediamente un numero di siti da visitare più elevato, costruendo spanning tree di maggiori dimensioni.

Query su rete generara con una distribuzione power-law

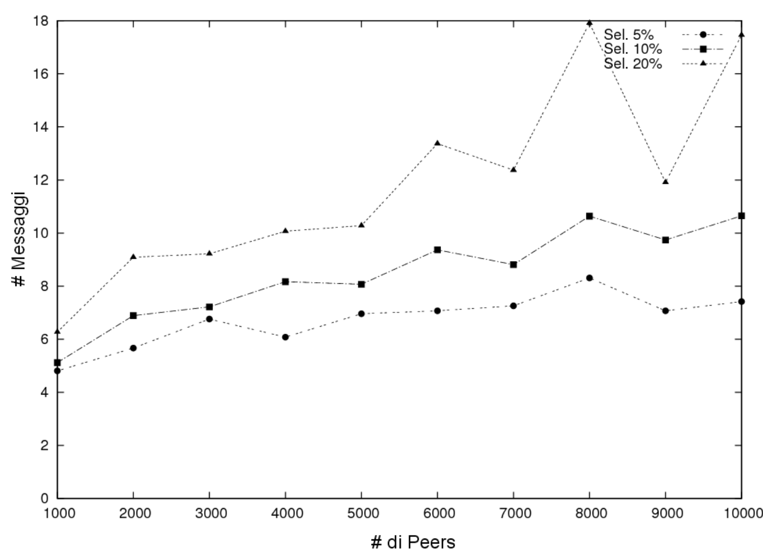


Figura 5.9: Numero medio di messaggi del routing greedy per query, ottenuto su una rete generata con una distribuzione power-law: $\alpha = 2$

Le figure 5.9 e 5.10 mostrano l'andamento del numero medio dei messaggi del routing greedy per le query risolte in una rete generata con una distribuzione power-law con due valori diversi di α. Il numero medio di messaggi diminuisce drasticamente rispetto al caso di rete generata con distribuzione uniforme perché, come per l'inserzione, il numero dei siti totali della rete è nettamente inferiore. Vediamo comunque come, in accordo con la complessità teorica, il numero dei messaggi sia logaritmico rispetto al prodotto del valore di selettività maggiore tra gli attributi della rete, che ricordiamo è uguale per tutti i livelli, e la dimensione della rete. Questo ultimo risultato si può vedere osservando che il numero dei messaggi, per qualunque valore della selettività, cresce sempre meno con l'aumentare della dimensione della rete. L'andamento molto variabile probabilmente deriva dal fatto che, contrariamente al caso della rete rappresentata da un albero bilanciato, le reti dei vari livelli hanno un numero più variabile di siti l'una dall'altra e le query, che vengono generate casualmente, possono andare a visitare reti che contengono siti con un alto numero di peer e siti invece che ne contengono pochissimi.

Confrontando i risultati ottenuti per i diversi valori della selettività, possiamo osservare che anche in questo caso i valori rientrano nei limiti della complessità

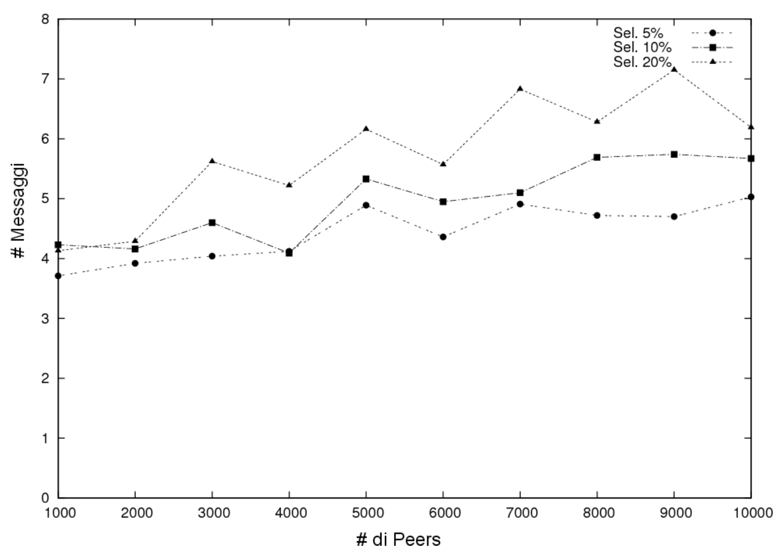


Figura 5.10: Numero medio di messaggi del routing greedy per query, ottenuto su una rete generata con una distribuzione power-law: $\alpha = 3$

teorica: infatti i messaggi aumentano linearmente con l'aumentare della grandezza delle AOI, cioè della percentuale di selettività, sebbene molto altalenanti (in alcuni casi, come per $N = 4000$ i valori per $SEL = 10\%$ e $SEL = 5\%$ vengono a coincidere).

Quindi anche in questo caso otteniamo un buon livello di scalabilità del routing greedy. Anzi, possiamo notare che il numero dei messaggi diminuisce sensibilmente quando i peer tendono a concentrarsi su un numero molto ristretto di siti della rete.

Le figure 5.11 e 5.12 mostrano l'andamento del numero medio dei messaggi del routing compass per le query risolte in una rete generata con una distribuzione power-law con due valori diversi di α. Vediamo come, rispetto al caso di query immesse in una rete generata con una distribuzione uniforme, il numero di messaggi diminuisca di un ordine di grandezza, e tenda a diminuire con l'aumentare della costante α.

Questo può essere dovuto al fatto che in una rete generata con una distribuzione power-law, la maggior parte dei peer va ad inserirsi in un unico sito in una rete al livello massimo dell'albero formando un cluster la cui dimensione tende a crescere con N. Essendo questo cluster formato da peer che hanno il valore di tutti gli attributi identico, basta che uno solo di questi peer venga contattato duran-

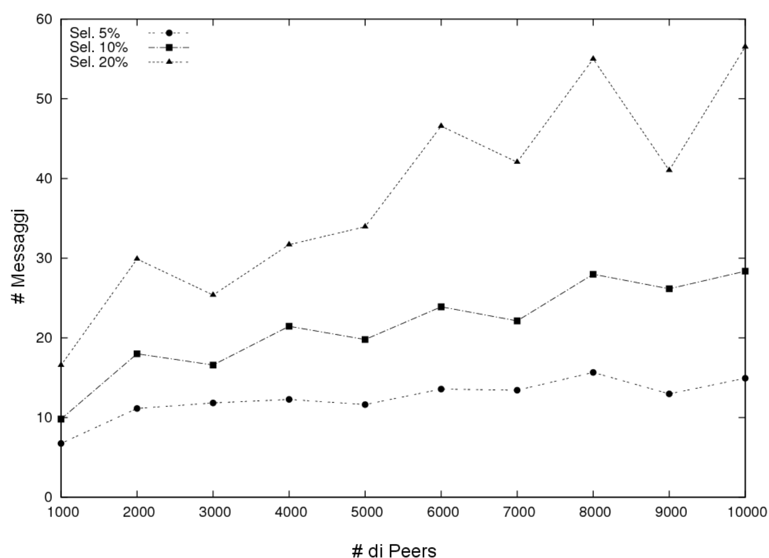


Figura 5.11: Numero medio di messaggi del routing compass per query, effetuato su una rete generata con una distribuzione power-law: $\alpha = 2$

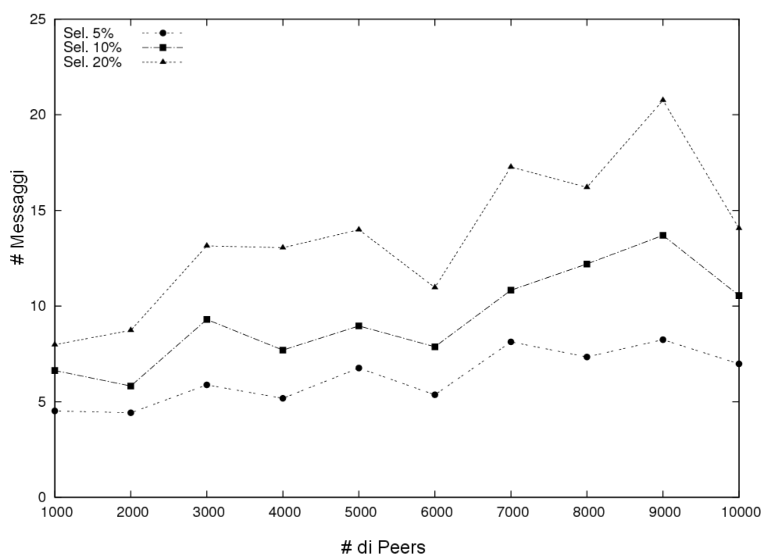


Figura 5.12: Numero medio di messaggi del routing compass per query, effettuato su una rete generata con una distribuzione power-law: $\alpha = 3$

te il compass routing affinché trasmetta una risposta alla query per conto di tutti, generando un solo messaggio per l'intero cluster. Questo se il cluster è match

per la query per almeno il valore di tutti gli attributi mappati ai livelli superiori al massimo. Se non lo è per almeno il valore di un attributo mappato ad un livello superiore al massimo, il sito corrispondente non viene visitato dal compass routing. Oltre a ciò, contribuisce ad ottenere il risultato descritto anche il fatto che il numero di peer non mappati in questo cluster è molto piccolo.

Confrontando i risultati ottenuti per i diversi valori della selettività, possiamo osservare che anche in questo caso il numero di messaggi cresce proporzionalmente all'aumentare del valore della selettività, come indicato dalla complessità teorica.

Anche in questo caso otteniamo un buon livello di scalabilità del routing compass, come possiamo osservare dall'andamento dei grafici.

Capitolo 6

Conclusioni e sviluppi futuri

Possiamo concludere che il sistema Hierarchical Voraque ha dato degli ottimi risultati di scalabilità ed è perfettamente in grado di soddisfare gli obiettivi che ci eravamo proposti:

- Gestire un numero di attributi maggiore di 2 senza cadere nei problemi di efficienza e scalabilità di una rete di Voronoi multidimensionale
- Gestire le range query multiattributo efficacemente ed efficientemente.
- Limitare il numero di vicini di un sito per ottenere un costo di inserimento e cancellazione dei peer nella rete sufficientemente contenuto anche in presenza di cluster di peer, la cui probabilità è aumentata dal fatto di avere più attributi.

Il sistema Hierarchical Voraque è quindi in grado di poter essere utilizzato come Grid Information Service, grazie al supporto per le range query multiattributo. I buoni risultati in termini di numero di messaggi raggiunti dall'algoritmo di inserimento di nuovi peer della rete è dovuto alla scelta di limitare il numero di peer vicini di un sito ad un numero logaritmico: questo ha permesso di contenere il costo a $O(\log N)$, ma soprattutto ha permesso di rendere il risultato **indipendente** dal numero di attributi e quindi di livelli della rete.

L'efficienza nella gestione delle query è paragonabile a quella di MAAN[7] in quanto, come quest'ultimo, sfrutta il routing ad attributo dominante: in questo modo il numero di peer visitati è limitato rispetto alla grandezza della rete, evitando di visitare reti che contengono peer il cui oggetto pubblicato non è a priori match per le query. Rispetto a MAAN però l'inserimento è molto meno costoso.

Inoltre lo spazio occupato dalle strutture dati di MAAN è maggiore rispetto a quello necessario per le strutture dati di Hierarchical Voraque. Infatti mentre il primo deve mantenere una rete Chord per ogni attributo e memorizzare l'intera

risorsa nella DHT di ognuna di queste reti, nel nostro caso ogni risorsa è inserita una sola volta nella rete. In più, la grandezza delle tabelle di routing di MAAN è uguale al numero di attributo moltiplicato per il logaritmo dei peer della rete. In Hierarchical Voraque invece è uguale alla metà del numero di attributi, cioè il numero di livelli, per il numero dei peer vicini ad ogni livello, che nel Teorema 2 abbiamo dimostrato essere al massimo il logaritmo dei peer della rete; nel nostro caso però questi sono valori massimi: un peer è visibile in più di un livello solo se viene eletto SuperPeer, e questo evento non è frequente. Inoltre il numero di vicini dipende dalla dimensione del cluster più grande tra questi, che solo per una rete rappresentata da un albero completamente sbilanciato è uguale alla grandezza della rete, ma questa è solo una delle tante possibili distribuzioni.

Un ultimo risultato di notevole importanza che abbiamo spiegato in dettaglio nello studio della complessità teorica della risoluzione delle range query multiattributo è che l'ordine in cui sono mappati gli attributi sui livelli della rete di Hierarchical Voraque è ininfluente. Questo significa che in realtà non è importante, come invece abbiamo indicato nel paragrafo 3.1, mappare gli attributi in ordine di selettività. Resta comunque vero che il costo reale di risoluzione delle range query multiattributo è influenzato dall'ordine degli attributi, ma soltanto per quanto riguarda le costanti moltiplicative del risultato teorico ottenuto dal calcolo della complessità.

Il problema del bilanciamento del carico, che non è trattato da nessun metodo che sfrutta un approccio basato su diagrammi di Voronoi, è stato risolto utilizzando SuperPeer multipli variabili in numero con la grandezza del cluster, supponendo che le query dirette ad un cluster di dimensione più grande siano più numerose e più frequenti di quelle dirette ad un cluster di dimensione più piccola.

Purtroppo, l'uso di un simulatore che sfrutta un unico thread non ha permesso lo studio del carico dei SuperPeer. Inoltre la presenza di più SP porta ad avere una elevata fault tollerance per i cluster, evitando un partizionamento tra i livelli e quindi alla perdita della consistenza della rete nel caso che un SP esca improvvisamente dalla rete.

Come sviluppi futuri possiamo indicare l'implementazione dell'architettura per mezzo di un'applicazione che sfrutti la rete fisica per poter studiare il bilanciamento del carico raggiungibile e il comportamento in caso di inserzione contemporanea di più peer. Inoltre, in un sistema reale possiamo studiare la fault tollerance effettiva di una rete costruita secondo Hierarchical Voraque.

L'aggiunta delle operazioni di abbandono della rete di un peer, revoca dei SuperPeer e cancellazione dei livelli di rete vuoti è un altro ramo di sviluppo possibile, in quanto da questo dipende lo studio del comportamento di Hierarchical Voraque quando si presenta un alto churn rate.

Un ulteriore campo di studio è quello relativo al comportamento di un sistema reale basato su Hierarchical Voraque in caso di attributi con valori dinamici.

Infine si potrebbe aggiungere il supporto ai peer che pubblicano più di un oggetto, caso che Hierarchical Voraque non tratta, e che è stato definito multi-siting.

Bibliografia

[1] **R. Steinmetz, K. Wehrle**. Peer-to-Peer systems and Applications. *Lecture Notes in Computer Science*, (3485), 2005.

[2] **Documentazione online su Napster**, 1999.
`http://opennap.sourceforge.net/napster.txt`.

[3] **The Annotated Gnutella Protocol Specification v0.4**, 2000.
`http://rfc-gnutella.sourceforge.net/developer/stable/index.html`.

[4] **I. Stoica, R. Morris, D. Karger, F. Kaashoek, e H. Balakrishnan**. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In ACM Press, editor, *Proc. of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, pages 149,160, 2001.

[5] **Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp e Scott Shenker**. A Scalable ContentAddressable Network. In ACM Press, editor, *Proc. of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, pages 161,172, 2001.

[6] **A. Rowstron e P. Druschel**. Pastry: Scalable, Decentralized Object Location and Routing for Large Scale Peer-to-Peer Systems. In *Proc. IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware 2001)*, pages 329–350,vol.2218. LNCS, Primavera 2001.

[7] **Min Cai, Martin Frank, Jinbo Chen e Pedro Szekely**. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Proc. of 4th Int. Workshop on Grid Computing*, pages 184,191, 2003.

[8] **Christina Schmidt e Manish Parashar**. Flexible Information Discovery in Decentralized Distributed Systems. In *Proc. of the 12th Int. Symp. on High-Performance Distributed Computing (HPDC-12 2003)*, pages 226–235, 2003.

[9] **Christina Schmidt e Manish Parashar**. Analyzing the Search Characteristics of Space Filling Curve-based indexing within the Squid P2P Data Discovery System. Technical report, CAIP,Rutgers University, Dicembre 2004.

[10] **Documentazione online sui Voronoi Diagrams**. `http://en.wikipedia.org/wiki/Voronoi`.

[11] **Jon Kleinberg**. The Small-World Phenomenon: an algorithmic perspective. In *Proc. 32nd ACM Symposium on Theory of Computing*, 2000.

[12] **Jon Kleinberg**. Navigation in a Small World. *Nature*, (406), 2000.

[13] **Farnoush Banaei-Kashani, Cyrus Shahabi**. Conference on informatio and knowledge management (cikm'04). In *SWAM: A family of access methods for similarity-search in Peer-to-Peer Data networks*, Washington D.C., Novembre 2004. ACM.

[14] **Olivier Beaumont, Anne-Marie Kermarrec, Loris Marchal, Etienne Rivière** . Voronet: A scalable object network based on Voronoi Tessellations. Inria reaserch report, INRIA, Febbraio 2006.

[15] **Baldanzi Martina**. Voraque: Range Query in reti P2P. Master's thesis, Università degli studi di Pisa, 2006/2007.

[16] **M.Albano, M. Baldanzi, R. Baraglia, L. Ricci**. VoRaQue: Range Queries on Voronoi Overlays. In *13th IEEE Symposium on Computers and Communications*, pages 495–500, Luglio 2008.

[17] **Mascitti Davide**. Estensione di uno strumento per range query multi-attributo. Master's thesis, Università degli studi di Pisa, 2007/2008.

[18] **P. Maymounkov e D. Maziueres**. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proc. First Int. Workshop on Peer-to-Peer Systems*, pages 53–65, 2002.

[19] **Moreno Marzolla, Matteo Mordacchini e Salvatore Orlando**. Peer-to-Peer Systems for Discovering Resources in a Dynamic Grid. In *Proc. of the 2nd Core-GRID Workshop on Grid and Peer to Peer Systems Architecture*, 2006.

[20] **H. Johner, L. Brown, F.S. Hinner, W. Reis e S. Shenker**. *Understanding LDAP*. IBM Redbooks, Giugno 1998.

[21] **Bittorrent Protocol Specification v1.0**.
`http://wiki.theory.org/BitTorrentSpecification`.

[22] **Documentazione online su Gnutella 0.6**, 2002.
`http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html`.

[23] **Documentazione online sul protocollo FastTrack e su Kazaa**.
`http://www.kazaa.com`
`http://it.wikipedia.org/wiki/FastTrack`.

[24] **Documentazione online sul protocollo di Dynamic Querying**.
`http://www.the-gdf.org/wiki/index.php?title=Dinamic_Query_Protocol`.

[25] **Qin Lv, Pei Cao, Edith Cohen, Kai Li e Scott Shenker**. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th ACM International Conference on Supercomputing*. ICS, 2002.

[26] **Christos Gkantsidis e Milena Mihail**. Hybrid Search Schemes for Unstructured Peer-to-Peer Networks. In *IEEE Infocom*. Georgia Institute of Technology, 2005.

[27] **David Oppenheimer, Jeannie Albrecht, David Patterson e Amin Vahdat**. Scalable Wide-Area Resource Discovery. Technical Report TR CSD04-1334, University of California Berkeley, 2004.

[28] **Franz Aurenhammer**. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3)::345–405, 1991.

[29] **F. Nielsen, J.D. Boissonnat, R. Nock**. Bregman Voronoi Diagram: Properties, Algortithms and Applications. In *Proc. of 18th ACM-SIAM Symposium Discrete Algorithms*, 2007.

[30] **M. DeBerg, M. vanKreveld, M. Overmars, O. Schwarzkopf**. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[31] **D. Watts, S. Strogatz**. Collective dynamics of small-world networks. *Nature*, pages 393–400, 1998. Londra.

[32] **E. Kranakis, H. Singh, J. Urrutia**. Compass Routing on Geometric Networks. In *Proc. of the 11th Canadian Conference on Computational Geometry*, 1999.

[33] **J. Liebherr, M. Nahas, W. Si**. Application -layer Multicasting with Delaunay Triangulation Overlays. *IEEE Journal on Selected Areas in Communications*, pages 1472–1488, Ottobre 2002.

[34] **Documentazione online sulla distribuzione Power Law**.
http://en.wikipedia.org/wiki/Power_law.

[35] **Steven Fortune**. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the second annual symposium on Computational geometry*, 1986.

[36] **Documentazione online su The Peersim Simulator**.
http://peersim.sourceforge.net/.

[37] **Documentazione online su Vast: Voronoi-based Adaptive Scalable Transfer**.
http://vast.sourceforge.net/.