UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

VORAQUE: RANGE QUERY IN RETI P2P

Relatore:
Prof. Laura Ricci
Prof. Ranieri Baraglia

Candidata:
Martina Baldanzi

Anno Accademico 2006/07

Alla mia famiglia,

per il passato, il presente e il futuro,

grazie per sempre

Indice

1 Introduzione **5**

2 Stato dell'Arte **13**

2.1 Sistemi non strutturati 14

2.1.1 Napster: P2P Centralizzato 14

2.1.2 Gnutella 0.4: P2P Puro 15

2.1.3 Gnutella 0.6: P2P Ibrido 16

2.2 Sistemi Strutturati 19

2.2.1 Chord 19

2.2.2 CAN (Content-Addressable Network) 19

2.2.3 Pastry 21

2.3 Confronto dei due modelli 23

2.4 Le reti P2P e le Griglie 24

2.4.1 Sicurezza: Differenze 25

2.4.2 Connessione: Differenze 26

2.4.3 P2P e Griglie: convergenza 26

2.5 P2P e Griglie: Scoperta di Risorse e Dinamicità 27

2.6 P2P e Griglie: Sistemi Strutturati 28

2.6.1 A DHT-based Peer-to-Peer Framework for Resource Discovery in Grids 28

2.6.2 Scalable, efficient range queries for grid information services . 30

2.6.3 Peer-to-Peer Discovery for Computantional Resources for Grid Applications 31
2.6.4 SQUID 33
2.6.5 Conclusioni 34
2.7 P2P e Griglie: Sistemi Non Strutturati 36
2.7.1 A Grid Information Service based on Peer-to-Peer 36
2.7.2 Peer-to-Peer Systems for Discovering Resources in a Dynamic Grid 37
2.7.3 Conclusioni 39

3 Approcci basati su tessellazione di Voronoi 41

3.1 Diagrammi di Voronoi 43
3.1.1 Triangolazione di Delaunay 44
3.2 Small World 46
3.3 VoroNet: un overlay basato su Voronoi 48
3.3.1 VoroNet: Routing 50
3.3.2 VoroNet: Inserimento 51
3.3.3 VoroNet: Cancellazione 51
3.3.4 VoroNet: Conclusioni 52
3.4 SWAM-V 53
3.5 VON 56
3.6 I diagrammi di Voronoi: altre applicazioni 59
3.6.1 Astronomia 59
3.6.2 Biologia 59
3.6.3 Chimica 59
3.6.4 Geologia 59
3.7 Compass Routing 60

4 VoRaQue: algoritmi e procolli 65
4.1 Ipotesi di base 67
4.2 Query Esatte 69
4.2.1 Query Esatte: ExactMSG 69
4.2.2 Query Esatte: ReplyExactMSG 70
4.2.3 Query Esatte: ExpiredTTL 70
4.2.4 Query Esatte: protocollo 70
4.3 Range Query 76
4.3.1 Range Query: RangeMSG 77
4.3.2 Range Query: ReplyRangeMSG 78
4.3.3 Range Query: VoronoiNeighMSG 79
4.3.4 Range Query: ReplyVoronoiNeighMSG 79
4.3.5 Range Query: protocollo 80
4.3.6 Range Query: Ricerca incrementale dei match 92
4.3.7 Range Query: Caching 96

5 VoRaQue:
Architettura e Implementazione 99
5.1 Package 99
5.2 Gestione ad eventi del Server 102
5.3 Elaborazione di una richiesta 108
5.4 Nomade e lowerlevel Package 111
5.4.1 Nomade API 112
5.4.2 Interazione VoRaQue - Nomade 115
5.5 Strutture Dati utilizzate 122
5.5.1 QueryStory 122
5.5.2 IdProcessedMessage 123
5.5.3 IdReplyRangeProcessed 123
5.5.4 VoronoiNeighNeigh 124

5.5.5 MatchRangeQuery 126

6 Risultati sperimentali **129**

6.1 Interazione con il sistema 129

6.2 Prove eseguite 134

6.2.1 Ambiente di prova 134

6.2.2 Valutazioni analitiche 134

6.2.3 Valutazione del traffico 137

6.2.4 Valutazione del numero di hop 138

6.2.5 Valutazione della latenza 142

6.3 Perdita match 144

6.3.1 Modifiche proposte per il compass routing 149

Conclusioni **153**

Bibliografia **157**

Capitolo 1

Introduzione

Nel corso degli ultimi anni si è assistito ad una crescente diffusione di applicazioni distribuite basate sul modello **Peer to Peer (P2P)**. Di pari passo vi è stata un'evoluzione algoritmica e architetturale dei sistemi che fanno uso di questo paradigma di comunicazione, sotto l'impulso della ricerca che è stata svolta in ambito industriale - accademico su questi argomenti.

Le reti P2P includono un numero di nodi equivalenti (**peer**, appunto) che fungono sia da client che da server verso altri nodi della rete. L'evoluzione delle applicazioni P2P ha visto l'adozione, da parte degli sviluppatori, di stili architetturali che si distaccano sempre più dal paradigma centralizzato, che in precedenza si era imposto come il modello di riferimento per lo sviluppo di applicazioni in rete.

Il limite delle applicazioni client/server si individua nella definizione di un unico server, che rappresenta il singolo punto di fallimento del sistema e pone un forte vincolo alla scalabilità.

Attualmente il modello P2P non pretende di sostituirsi al paradigma client/server, ormai consolidato e adeguato per tante tipologie di applicazioni, ma vuole piuttosto offrire un modo alternativo per sfruttare le potenzialità di Internet, basandosi su un'architettura quanto più possibile decentralizzata.

Esistono due topologie di sistemi P2P: sistemi **Non Strutturati** e sistemi **Strut-**

turati

Nei sistemi P2P **Non Strutturati**, ogni risorsa presente nella rete è localizzata sul nodo che la mette a disposizione, non esistono informazioni relative alla loro locazione e la ricerca risulta essere difficoltosa a causa della mancanza di "organizzazione" all'interno del sistema. D'altra parte, la rimozione o l'aggiunta di nodi dalla rete è un'operazione semplice e poco costosa. Fanno parte dei sistemi P2P non strutturati Napster[15] e Gnutella [16, 17, 18].

Napster [15] è stato storicamente il primo sistema P2P. E' caratterizzato da un server centrale che memorizza la lista delle informazioni condivise da ogni nodo connesso. Per individuare una risorsa è necessario interrogare il server che restituisce l'elenco dei peer che condividono l'informazioni richiesta. Anche se la copia del file richiesto avviene in modalità P2P, la scoperta delle risorse è centralizzata. Poiché il server rappresenta il collo di bottiglia delle prestazioni dell'intero sistema, Napster risulta poco scalabile. D'altra parte, il sitema garantisce che, se una risorsa è presente sicuramente verrà localizzata e verrà impiegato un solo passo per raggiungere la sua origine.

Il progetto Gnutella [16, 17, 18] ha eliminato la necessità di un server centrale per fornire le funzionalità di ricerca. I "pari" nella rete Gnutella [16] sono responsabili non solo di fornire le risorse, ma anche di rispondere alle richieste e ai messaggi di routing degli altri peer: essi operano dunque sia da client che da server e sono denominati "servent". Per la localizzazione degli oggetti, Gnutella sfrutta il *query flooding* limitato da *TTL (Time-To-Live)* [16]. Quando un utente invia il suo messaggio di richiesta il *TTL* viene inizializzato a un limite specifico. Tutte le volte che la richiesta raggiunge un nuovo pari, quest'ultimo decrementa il valore del *Time-To-Live* prima di inoltrare la query ai suoi vicini. Se un nodo riceve una richiesta con *TTL* uguale a zero, questa non verrà inoltrata ad altri nodi. In questo modo, l'inondazione delle query è localizzata in una regione della rete di overlay.

In Gnutella la quantità di memoria richiesta è dell'ordine di $O(1)$, in quanto è indipendente dal numero di nodi partecipanti. Mentre la risoluzione di una query, che

si basa su inondazione di richieste, risulta lineare nel numero di risorse disponibili nel sistema. Inoltre, il problema principale di questa soluzione è la possibilità che non vengano trovati tutti i pari con il contenuto desiderato.

Le Distributed Hash Table nascono per superare i limiti dei sistemi P2P di prima e seconda generazione. L'aspetto chiave delle DHT 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. Esempi tipici di DHT sono Chord [20], CAN [21] e Pastry [22].

Le Distributed Hash Table 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.

Con il termine **range query multi-attributo** si definisce una richiesta in cui almeno una coordinata dello spazio degli attributi è un **range di valori**.

Le range query hanno riscontrato particolare interesse nell'ambito di **Grid Information Service**.

Con il termine Griglia, si intende un'infrastruttura distribuita che consente l'utilizzo di risorse di calcolo e di memorizzazione provenienti da un numero indistinto di calcolatori interconnessi da una rete.

Una funzionalità fondamentale richiesta alle Griglie è la capacità di individuare la locazione di una risorsa che soddisfa particolari vincoli. Questo è necessario quando un utente sottomette un job, specificando le caratteristiche (come quantità di memo-

ria, spazio disco, sistema operativo, ecc) delle risorse computazionali che dovranno eseguirlo. Il problema della scoperta di risorse è molto complesso, soprattutto se quest'ultime sono di tipo dinamico. In ambiente Grid, per individuare una certo numero di risorse computazionali vengono utilizzate le **range query multi-attributo**.

In letteratura sono stati sviluppati molti sistemi che si preoccupano di risolvere le problematiche legate alla scoperta di risorse nelle Griglie. La differenza sostanziale tra le varie soluzioni proposte si riscontra nella tipologia di architettura P2P che viene utilizzata.

Il sistema proposto in [6] si basa su un'architettura di tipo multi-DHT e ogni risorsa viene descritta da una lista di attributi, che vengono divisi in statici e dinamici. Per la risorsa computazionale un attributo statico è, ad esempio, il nome del sistema operativo installato, mentre un attributo dinamico è il carico di lavoro della CPU. In questo sistema ogni attributo che descrive la risorsa fa parte di una DHT diversa e le range query su attributi statici vengono eseguite sfruttando le corrispondenti DHT. In più, viene aggiunta una **DHT general purpose** per gestire le range query su attributi dinamici. La DHT general purpose può sfruttare il broadcast per raggiungere tutti i nodi.

Il sistema proposto in [11] sfrutta la rete di overlay di Pastry e si preoccupa di mappare su tale DHT le risorse, rappresentate come collezione di attributi. La tattica adottata si basa sul combinare la parte statica e dinamica degli attributi della risorsa per ottenere un ResourceID. Gli attributi dinamici hanno la caratteristica di assumere valori continui e devono essere discretizzati, rappresentandoli tramite una percentuale e utilizzando un numero fisso di bit. In questo modo, per individuare lo stato esatto della risorsa basta combinare la rappresentazione statica con quella dinamica. L'obiettivo principale è quello di inserire l'informazione nella DHT sfruttando il valore degli attributi statici. Successivamente, definire un intervallo di identificatori, a partire da quel punto, che descriva la disponibilità degli attributi dinamici. Gli autori propongono tre diverse euristiche per la risoluzione delle query.

Squid [1, 2] è un sistema P2P per la scoperta di risorse, la sua architettura si basa

su Chord, ma differisce da una vera e propria DHT per la modalità con cui le informazioni vengono mappate nello spazio degli indirizzi. Squid tenta di preservare la località utilizzando lo **Hilbert Space Filling Curves (HSFC)** che è una funzione che mappa lo spazio *d-dimensionale* in uno spazio a *1 dimensione*. Ogni dato condiviso viene descritto da un'insieme di parole chiave. Tramite la funzione HSFC, ogni dato viene mappato nello spazio degli indici, mentre i nodi ricevono un identificatore, appartenente allo stesso spazio, scelto casualmente.

Squid elabora una query in due passi: prima traduce la query in un'area significativa nello spazio degli indici, poi interroga i nodi corrispondenti per identificare il dato.

I sistemi appena descritti hanno l'obiettivo di realizzare un'architettura che possa gestire le range query. In realtà i sistemi proposti non presentano soluzioni efficienti per il problema della scoperta di risorse. Ognuno di questi implementa l'overlay network con una DHT e, come abbiamo evidenziato sopra, le Distributed Hash Table sono il miglior sistema per risolvere le **query esatte**, ma non sono in grado di risolvere efficientemente query per range di valori. Inoltre, non rappresentano la soluzione migliore in presenza di dati altamente dinamici come il carico di lavoro della CPU poichè, ogni volta che il valore della risorsa cambia, occorre re-indicizzare l'oggetto sulla rete.

Diversamente, in sistemi P2P non strutturati [7, 5], viene messa in evidenza la capacità di gestire efficientemente le range query utilizzando la strategia del Routing Index [9], con minor costi di gestione della rete. La tecnica Routing Index decide a quale nodo inviare la query, tenendo conto della disponibilità che il potenziale destinatario ha del dato richiesto. Questa strategia supporta i frequenti aggiornamenti dei valori degli attributi delle risorse, senza la necessità di notificare il cambiamento a tutti i nodi. L'uso di architetture 'meno strutturate' non garantisce però l'individuazione di tutti i possibili match per una query.

Ultimamente sono stati proposti approcci alternativi. Tra questi i più interessanti sono quelli basati sui diagrammi di Voronoi [57].

VoroNet [12] è un sistema P2P che 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 dal fatto che non sfrutta funzioni hash né per distribuire il carico tra i peer, né per assegnare gli identificatori. Piuttosto, VoroNet descrive uno spazio bi-dimensionale, dove ogni dimensione rappresenta un attributo. Ogni nodo fisico, partecipante alla rete, publica un certo numero di oggetti che vengono rappresentati come punti in un spazio bi-dimensionale. Lo spazio degli attributi viene mappato in un diagramma di Voronoi in 2 dimensioni.

Un obbiettivo fondamentale di VoroNet è quello di fornire un routing poli-logaritmico nel numero dei nodi partecipanti. A questo scopo il sistema si ispira al modello dello small-world proposto da Kleinberg [13, 14].

Un altro sistema P2P la cui rete di overlay si basa sui diagrammi di Voronoi è SWAM-V [49]. SWAM-V è stato realizzato con lo scopo 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*.

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 della rete e descrive le connessioni tra i nodi vicini. Inoltre, vengono aggiunti dei *random link* in accordo al grafo definito dallo Small World di Kleinberg [13, 14], garantendo un routing poli-logaritmico nel numero di nodi della rete.

Consideriamo per semplicità di trattazione il caso bi-dimensionale. Una range query viene definita da una chiave $\overrightarrow{q}$ e da un range **r** e descrive una circonferenza nello spazio. La risoluzione di tale richiesta viene eseguita in due fasi:

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

2. A partire da n_k viene applicato un algoritmo di routing **flooding** per recuperare

tutti i possibili match che sono contenuti nella circonferenza descritta dalla range query.

Questo approccio si rivela efficace, ma non efficiente dal punto di vista dei messaggi spediti sulla rete.

Questa tesi propone VoRaQue (Voronoi Range Query), un sistema P2P basato sull'uso dei diagrammi di Voronoi. *L'aspetto innovativo che introduce VoRaQue rispetto agli altri sistemi basati sui diagrammi di Voronoi, è la definizione di un protocollo per la risoluzione efficiente delle range query che viene realizzato sfruttando l'algoritmo di **compass routing** [47].*

Il compass routing è un algoritmo di instradamento applicato su grafi il cui obiettivo non è quello di trovare il cammino più breve per connettere due vertici, ma quello di garantire il raggiungimento della destinazione.

In particolare, prendendo qualsiasi coppia di punti appartenenti alla triangolazione di Delaunay, il compass routing determina sempre uno ed un solo cammino che unisce tali punti [47]. Inoltre, questa tecnica di instradamento permette di costruire uno spanning tree a partire dalla triangolazione di Delaunay. Tale albero viene creato in maniera distribuita da ogni nodo.

Una funzionalità definita dal protocollo è la ricerca incrementale dei risultati. In VoRaQue il compass routing è limitato da *Time-To-Live* e quando un peer azzera tale valore, si preoccupa di costruire il messaggio di risposta con tutti i risultati fino ad allora raccolti, da inviare direttamente al nodo richiedente. Quest'ultimo estrae dal pacchetto i match e le informazioni che individuano il peer che ha interrotto la ricerca. In questo modo è possibile ottimizzare il protocollo perché per ottenere ulteriori risultati possono essere contattati direttamente i nodi che hanno arrestato la ricerca, per continuare la risoluzione della query a partire da loro.

Il protocollo sfrutta inoltre, tecniche di caching per velocizzare l'elaborazione di una range query, sia sui risultati resituiti dal compass routing, sia sulle informazioni utili per il calcolo della relazione padre-figlio.

In conclusione, VoRaQue realizza tecniche per la risoluzione di range query innovative ed efficienti. Innovative perchè nessun sistema presente in letteratura utilizza il compass routing per la scoperta di risorse. Inoltre tale algoritmo, costruendo uno spanning tree distribuito, permette di minimizzare il numero di messaggi spediti in rete per raggiungere tutti i match.

La struttura della tesi è così organizzata. Nel capitolo 2 si illustra l'evoluzione delle reti P2P, le similitudini e le differenze tra P2P Computing e Grid Computing, infine viene fatta una panoramica sui sistemi Peer-to-Peer strutturati e non, presenti in letteratura. Nel capitolo 3 viene data la definizione di diagramma di Voronoi e ne vengono elencate le principali proprietà, vengono presentati tre sistemi P2P la cui overlay network è organizzata secondo la tessellazione di Voronoi, infine viene descritto il compass routing ed evidenziate le sue potenzialità. Nel capitolo 4 vengono definiti in dettaglio i protocolli e gli algoritmi implementati da VoRaQue per la risoluzione delle query esatte e query complesse, riportando un'insieme di esempi per chiarirne il funzionamento. Nel capitolo 5 viene descritta l'architettura del sistema, le scelte implementative e progettuali, ed infine viene fatta una panoramica sulle strutture dati utilizzate. Nel capitolo 6 vengono descritti gli esperimenti eseguiti e vengono mostrati i risultati ottenuti, inoltre viene evidenziato un caso in cui il compass routing su un'area limitata presenta dei limiti e vengono fornite due soluzioni per superare tale limite. Infine, il capitolo delle conclusioni descrive un caso di studio e gli sviluppi futuri.

Capitolo 2

Stato dell'Arte

Con il termine **Peer to Peer** (**P2P**) si intende una rete di computer o una qualsiasi rete che non possiede client o server fissi, ma un numero di nodi equivalenti (peer, appunto) che fungono sia da client che da server verso altri nodi della rete. L'evoluzione delle applicazioni P2P ha visto l'adozione, da parte degli sviluppatori, di stili architetturali sempre più distaccati dal paradigma centralizzato, il quale, dopo la nascita di Internet, si era imposto come il modello di riferimento per lo sviluppo di applicazioni in rete. Oggi il modello P2P non pretende di sostituirsi al paradigma client/server, ormai consolidato e adeguato per tante tipologie di applicazioni, ma vuole piuttosto offrire un modo alternativo per creare applicazioni, sfruttando le potenzialità di Internet e basandosi su un'architettura quanto più possibile decentralizzata. Le reti P2P si basano sul principio che tutte le componenti del sistema hanno le stesse responsabilità e si comportano sia da client che da server. Esistono due topologie di sistemi P2P: sistemi **Non Strutturati** e sistemi **Strutturati**.

2.1 Sistemi non strutturati

Nei sistemi P2P non strutturati, ogni risorsa è localizzata sul nodo che la mette a disposizione. Infatti, non esistono informazioni relative alla loro locazione e la ricerca risulta essere difficoltosa a causa della mancanza di "organizzazione" all'interno del sistema. D'altra parte, la rimozione o l'aggiunta di nodi dalla rete è un'operazione semplice e poco costosa. Ma vediamo più in dettaglio quali sono i sistemi P2P non strutturati.

2.1.1 Napster: P2P Centralizzato

Napster [15] venne introdotto nel 1999 e storicamente è stato il primo sistema P2P. Nato dal lavoro di un giovane studente universitario americano, Napster offriva agli utenti la possibilità di scambiarsi file mp3. Essendo un software libero, ha avuto una diffusione capillare all'interno della rete fino a quando è poi tramontato a causa delle implicazioni legali relative all'infrazione del copyright. Napster è caratterizzato da un server centrale, il quale memorizzava la lista dei file mp3 presenti in ogni macchina, ad esso connessa (figura 2.1).

Per individuare un file era necessario interrogare il server, utilizzando il nome della canzone, il quale restituiva una lista di indirizzi IP di tutti i peer che condividevano l'informazione richiesta. La funzionalità di trasferimento file era poi coordinata direttamente tra i peer, i quali stabilivano una connessione diretta tra loro. Se la fase di copia del file richiesto avveniva in modalità P2P, la scoperta delle risorse era centralizzata. Infatti, Napster risultava non scalabile ed il server rappresentava il collo di bottiglia delle prestazioni dell'intero sistema. D'altra parte, il sistema garantiva che se una risorsa era presente, sicuramente sarebbe stata localizzata e sarebbe stato impiegato un solo passo per raggiungere la sua origine.

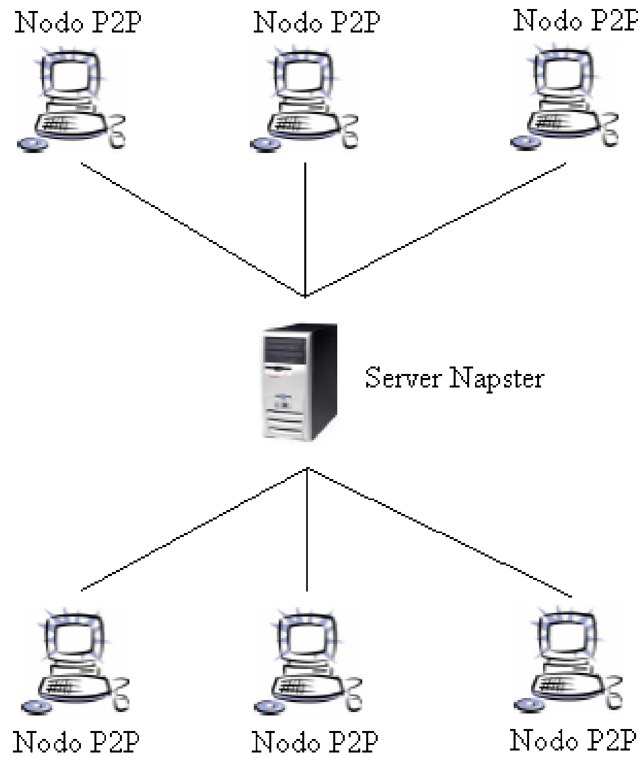


Figura 2.1: Architettura Napster.

2.1.2 Gnutella 0.4: P2P Puro

Crescendo l'esigenza di realizzare soluzioni sempre più decentralizzate, il progetto Gnutella [18] ha rafforzato il concetto di file sharing proposto da Napster, eliminando la necessità di un server centrale per fornire le funzionalità di ricerca. I "pari" nella rete Gnutella [16] sono responsabili non solo di fornire i file, ma anche di rispondere alle richieste e ai messaggi di routing degli altri peer: essi operano dunque sia da client che da server e sono denominati "servent" (figura 2.2). Per la localizzazione degli oggetti, Gnutella sfrutta il *query flooding* limitato da *TTL (Time-To-Live)* [16].

Quando un utente invia il suo messaggio di richiesta, il *TTL* viene inizializzato a un limite specifico e viene inviato ad ogni nodo suo vicino. Tutte le volte che la richiesta raggiunge un nuovo pari, quest'ultimo decrementa il valore del *Time-To-Live* prima di rilanciare la query ai suoi vicini. Se un nodo riceve una richiesta con *TTL*

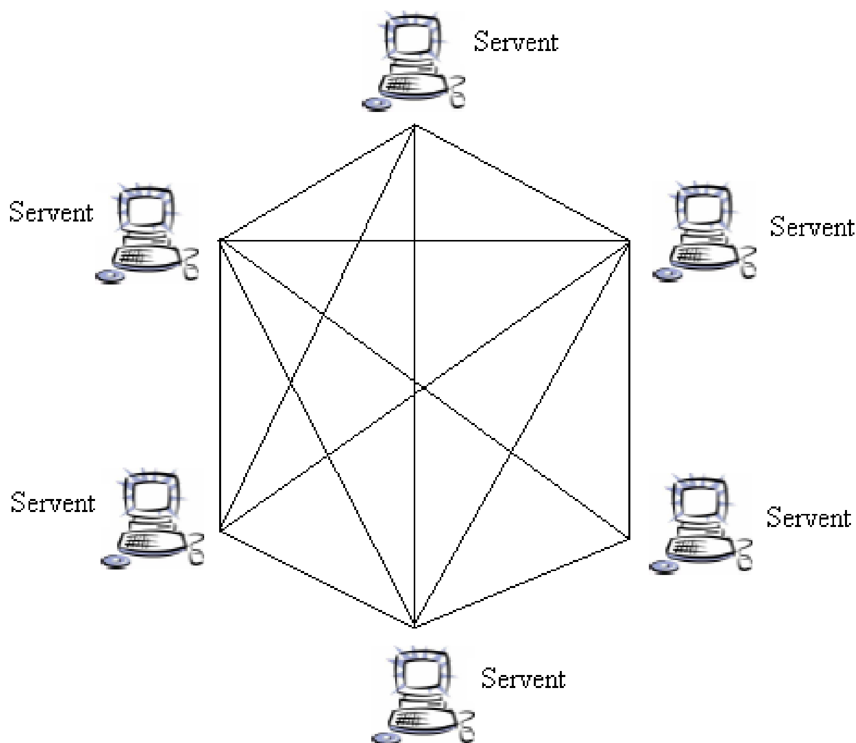


Figura 2.2: Architettura Gnutella 0.4.

uguale a zero, questa non verrà inoltrata ad altri nodi. In questo modo, l'inondazione delle query è localizzata in una regione della rete di overlay[1], con lo svantaggio che è possibile che non vengano trovati tutti i pari con il contenuto desiderato.

2.1.3 Gnutella 0.6: P2P Ibrido

La versione 0.6 di Gnutella [17, 18] nasce nella primavera del 2001, sviluppata a partire da Gnutella 0.4, con l'obbiettivo principale di ridurre il numero di messaggi scambiati nelle reti P2P pure. Affinché ciò fosse possibile, fu introdotta una gerarchica a due

[1]Per *rete di overlay* (od *overlay network*), si intende una rete di computer che viene costruita sopra un'altra rete. I nodi che ne fanno parte sono connessi tramite collegamenti virtuali o logici, ognuno dei quali corrisponde ad un cammino su collegamenti fisici della rete sottostante[19]. Nel caso di sistemi P2P, la rete sottostante è Internet.

livelli, caratterizzata da peer che si comportano o da *SuperPeer* o da *LeafNode* (figura 2.3).

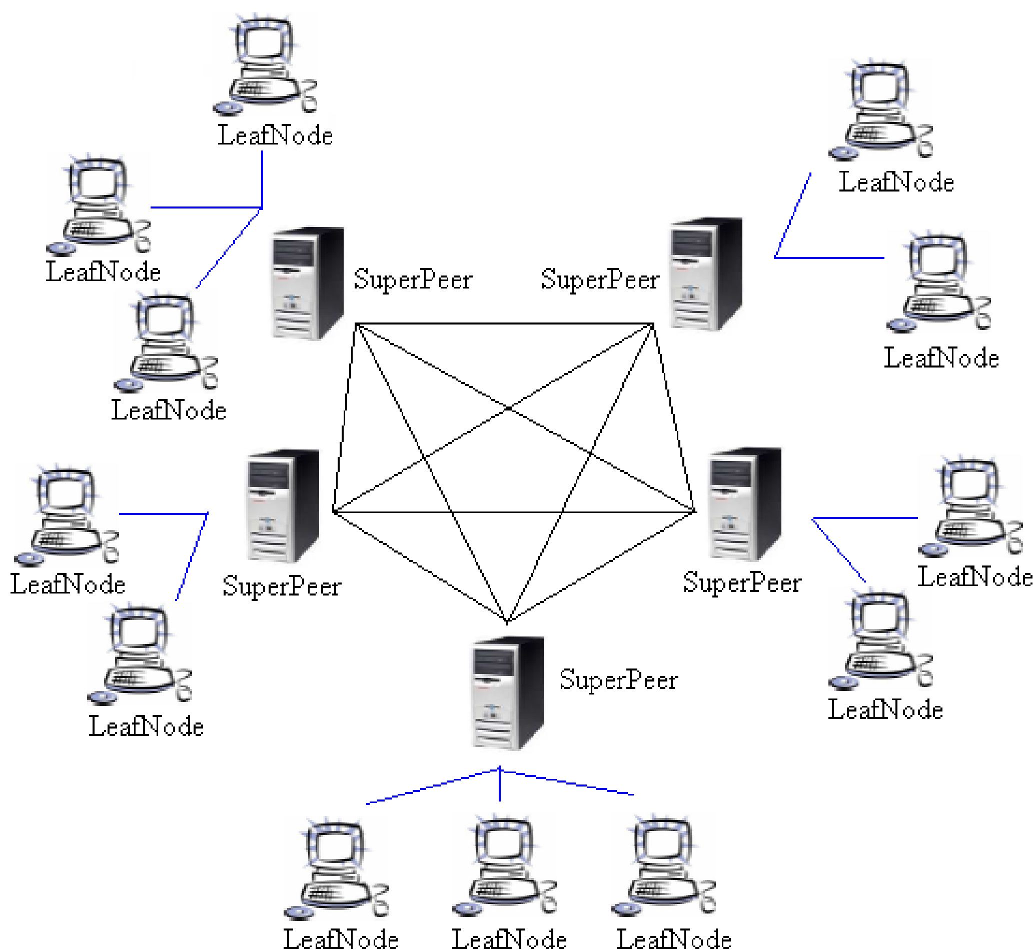


Figura 2.3: Architettura Gnutella 0.6.

L'elezione dei SuperPeer avviene in maniera completamente decentralizzata, infatti ogni host, prima di connettersi alla rete Gnutella, verifica se ha le caratteristiche fisiche per diventare SuperPeer. Ad esempio, non deve essere a monte di un firewall oppure deve disporre di una certa quantità di banda sia in ingresso che in uscita. In questa soluzione ibrida, per risolvere una richiesta vengono eseguite le seguenti operazioni: il LeafNode fa la richiesta al SuperPeer, il quale controlla se qualcuno tra i nodi ad esso collegati offre la risorsa cercata, in caso affermativo riceveranno la query. Quest'ultima

viene inviata anche agli altri SuperPeer per individuare altri host che condividono il file richiesto. Quando il LeafNode riceve la query, la risposta viene instradata all'indietro lungo lo stesso cammino che aveva percorso la richiesta [17]. Questa tipologia di sistemi permette di mantenere la semplicità delle soluzioni non strutturate aggiungendo scalabilità. Anche in questo caso, è possibile che non vengano trovati tutti i pari con il contenuto desiderato, in quanto la rete dei SuperPeer mantiene la struttura definita da Gnutella 0.4.

2.2 Sistemi Strutturati

Nei sistemi basati su *overlay* strutturato i nodi sono organizzati logicamente in modo che ciascuno possa essere raggiunto in un numero limitato di passi, tipicamente di ordine logaritmico rispetto al numero di nodi presenti sulla rete. Ogni nodo attua delle precise politiche di *routing* per inoltrare i messaggi. La maggior parte di essi si basa sull'uso delle tabelle DHT. Questo tipo di sistemi consentono di stabilire dei percorsi di comunicazione tra i nodi che costituiscono la rete e se una risorsa richiesta è presente nel sistema, verrà sicuramente localizzata. D'altra parte le operazioni di inserimento e cancellazione di un nodo sono operazioni molto costose.

2.2.1 Chord

Chord [20] nacque nel 2001 ed è stato sviluppato presso l'Università della California. Sia i nodi che i file vengono mappati, sfruttando la funzione hash SHA-1, in uno spazio delle chiavi, definito dall'intervallo $[0, 2^m - 1]$. Gli identificatori *(ID)* vengono ordinati mediante un anello, in cui il successore di $2^m - 1$ è 0, come si vede in figura 2.4. Ogni nodo memorizza tutte le risorse la cui chiave risulta minore o uguale all'*ID* del nodo.

Chord, per risolvere una query, emula una ricerca binaria, infatti occorrono $O(\log N)$ hop per raggiungere la destinazione, dove N rappresenta il numero di nodi che fanno parte della rete. Inoltre, ogni peer ha conoscenza soltanto di $O(\log N)$ vicini. D'altra parte però, sono necessari meccanismi costosi per mantenere consistente l'Anello di Chord e manca la nozione di prossimità fisica.

2.2.2 CAN (Content-Addressable Network)

CAN [21] nasce con l'obbiettivo di limitare il numero dei vicini di ogni peer, indipendentemente dalla dimensione che può assumere la rete di overlay o l'intervallo delle chiavi possibili. In questo sistema, lo spazio degli indirizzi viene rappresentato come un *toro d-dimensionale*, questo significa che ogni peer o dato che appartiene a questa

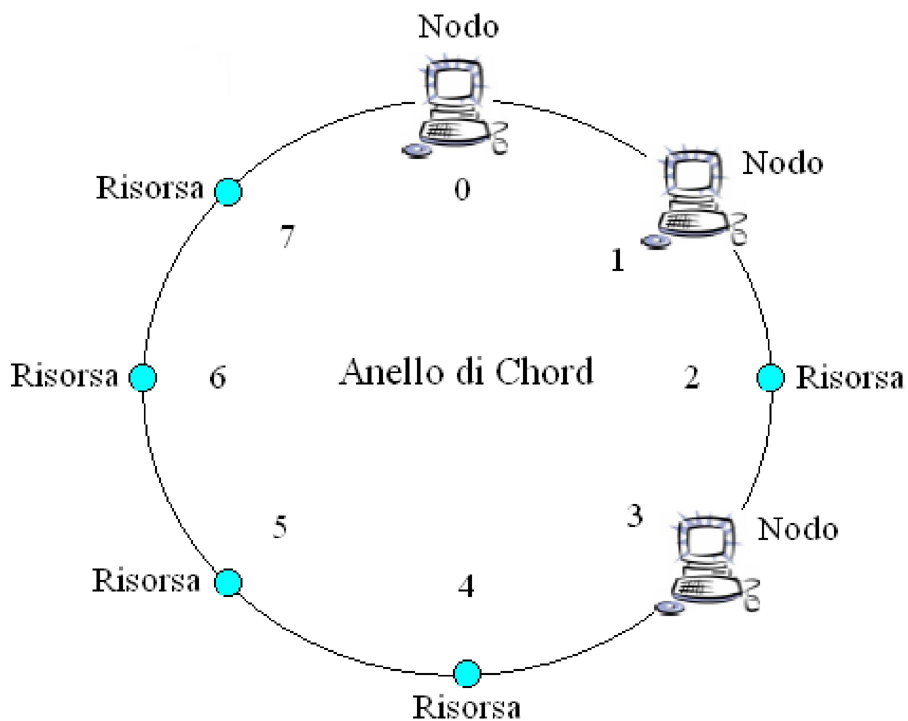


Figura 2.4: In questo esempio abbiamo m = 3 e l'intervallo per lo spazio delle chiavi è $[0, 7]$. La risorsa, con chiave 2, è memorizzata dal nodo che ha ID = 3. Mentre, le risorse che hanno chiave 4, 5, 6 e 7, sono memorizzate dal nodo con ID = 0.

rete, viene definito come un punto in *d-coordinate*. Tale spazio d-dimensionale, viene diviso equamente tra i nodi disponibili ed ogni risorsa viene memorizzata dal peer che gestisce la zona in cui è stata mappata (figura 2.5).

In questo sistema, la dimensione delle tabelle di routing è dell'ordine di $O(d)$ e il numero medio di passi necessari per individuare un'informazione è $d * N^{\frac{1}{d}}$, con N numero totale di nodi. Anche per CAN occorrono meccanismi per mantenere consistente la sua topologia e sono piuttosto complicati quando un nodo lascia la rete.

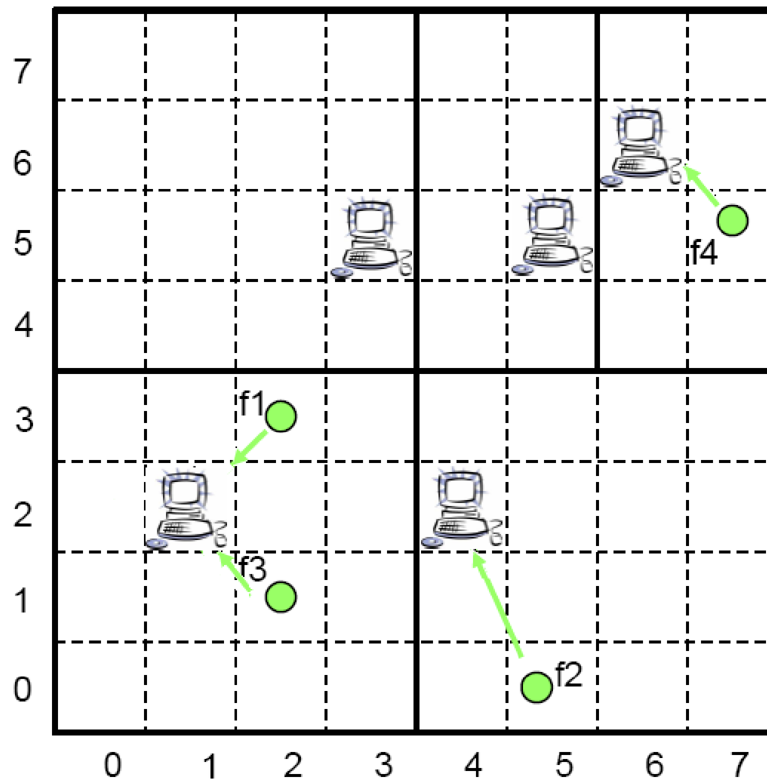


Figura 2.5: È rappresentato uno spazio bi-dimensionale, dove il nodo $(1, 2)$ memorizza le risorse f1 e f3, il nodo $(4, 2)$ memorizza la risorsa f2 e il nodo $(6, 6)$ memorizza la risorsa f4.

2.2.3 Pastry

Pastry [22], proposto nel 2001, assegna ad ogni peer e ad ogni dato un identificatore di 128 bit mediante una funzione hash, chiamati rispettivamente *nodeId* e *key*. I nodi sono organizzati in un anello secondo il loro nodeID e si preoccupano di gestire tutte le chiavi, il cui valore è numericamente vicino al valore del suo identificatore, come si vede in figura 2.6.

I nodi sono logicamente raggruppati in base al loro indirizzo, ed ognuno di essi conosce l'IP di un nodo "rappresentante" di ogni altro gruppo. In questo modo, ogni peer conosce $O(\log N)$ vicini ed ogni file viene localizzato in $O(\log N)$ hop. Diversamente dalle altre DHT, Pastry introduce la nozione di prossimità fisica nel

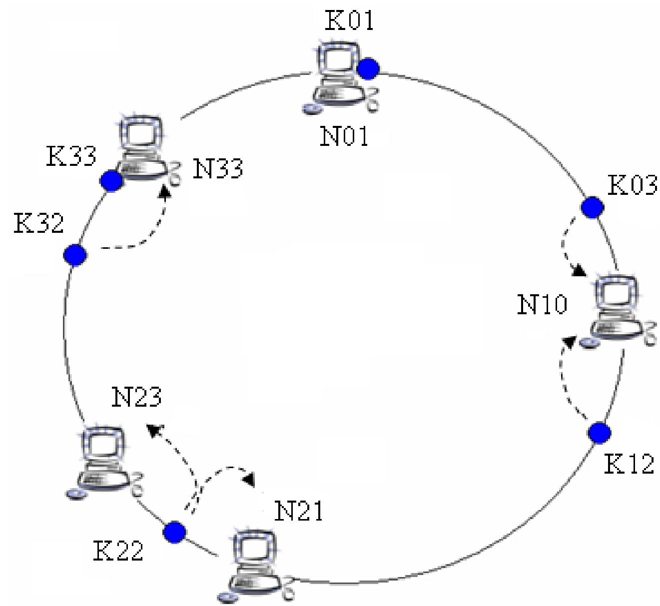


Figura 2.6: Pastry.

routing.

2.3 Confronto dei due modelli

I sistemi P2P strutturati riescono a superare i limiti dei sistemi non strutturati, in quanto garantiscono che:

1. la risorsa richiesta, se è presente nel sistema, verrà sicuramente localizzata;
2. un overhead di comunicazione logaritmico nel numero dei nodi presenti nel sistema;
3. l'occupazione di memoria per le tabelle di routing è logaritmica nel numero dei nodi presenti nel sistema;
4. il sistema è scalabile al caso medio.

In presenza di un elevato numero di inserimenti e cancellazioni da parte dei peer sulla rete di overlay, la scalabilità è compromessa a causa del traffico di messaggi non trascurabile, utile soltanto per mantenere consistente la struttura rigida del sistema. Per contro, abbiamo che le soluzioni P2P non strutturati sono meno scalabili rispetto alle DHT, ma non soffrono delle problematiche sopra descritte.

2.4 Le reti P2P e le Griglie

Le reti Peer-to-Peer e le Griglie [23] sono sistemi computazionali distribuiti che rendono possibile una collaborazione decentralizzata dei nodi che vi partecipano, integrando gli elaboratori in un'unica rete di overlay nella quale ognuno di essi può sfruttare ed offrire dei servizi. Con il termine Griglia, si intende un'infrastruttura distribuita che consente l'utilizzo di risorse di calcolo e di storage provenienti da un numero indistinto di calcolatori (anche e soprattutto di potenza non particolarmente elevata) interconnessi da una rete.

Le reti P2P e le Griglie sono i più comuni sistemi per la condivisione di risorse, si sono evoluti a partire da comunità differenti e per questo motivo soddisfano esigenze diverse. Le Griglie interconnettono cluster di computer ed altre infrastrutture per rendere possibile la condivisione di risorse esistenti, come l'utilizzo della CPU, la memoria, i dati e le applicazioni software.

Il primo campo di applicazione delle Griglie fu la fisica delle particelle elementari, dove la necessità di gestire elevate quantità di calcolo, ha portato l'**INFN** a mettere a punto già nel '99 il progetto **INFN-Grid**, che prendeva spunto dalle proposte avanzate pochi mesi prima in America da Ian Foster e Carl Kesselman [28].

La fisica delle particelle elementari è stato il primo campo scientifico ad utilizzare le capacità di calcolo delle Griglie, ma oggigiorno, questo sistema viene applicato con successo anche nella cosmologia, nella chimica computazionale ad alta risoluzione e nei modelli climatici. Inoltre, la Griglia è un sistema caratterizzato da un'elevata dinamicità in quanto l'insieme delle risorse computazionali e di rete che la costituiscono possono variare nel tempo sia in numero che in tipo.

D'altra parte, il servizio più popolare fornito dai sistemi P2P è il *file sharing*, tramite il quale i nodi connessi alla rete di overlay condividono file con gli altri (ad es. Napster e Gnutella per la musica). Altre applicazioni sono:

- *il trasferimento di dati in tempo reale* (telefonia con Skype);

- *computazioni parallele (SETI@home* [29]).

L'utente che desidera entrare a far parte di una rete P2P, accede tramite un'applicazione in esecuzione su un comune host. Successivamente, effettuata l'operazione di unione alla comunità, può richiedere di scaricare musica, video o qualsiasi altra risorsa di interesse, tramite connessione TCP sulla rete Internet. Il sistema mostra un'alta dinamicità determinata sia dal comportamento degli utenti, in quanto possono entrare, uscire ed unirsi alla rete in un modo completamente asincrono, sia dalla tipologia delle risorse fornite dai peer che in ogni istante possono decidere di inserirle o rimuoverle.

Riassumendo, le reti P2P e le Griglie mostrano sostanziali differenze sia dal punto di vista degli utenti interessati, ma soprattutto nella natura dinamica delle risorse della Griglia contrariamente al puro file sharing che è di gran lunga il più comune servizio nei sistemi P2P.

2.4.1 Sicurezza: Differenze

La sicurezza è un tema centrale nelle Griglie, infatti molti sforzi sono stati dedicati per aggiungere a questa piattaforma i meccanismi fondamentali per l'autenticazione, l'autorizzazione, l'integrità e la confidenzialità. Ciò nonostante, alcuni di questi aspetti, sono stati pensati principalmente per "comunità ristrette", nelle quali il sistema permette agli utenti di partecipare senza una fase di autenticazione. A causa della loro natura, alcuni meccanismi di sicurezza non garantiscono l'anonimato né degli utenti, né delle risorse.

Contrariamente, i sistemi P2P nascono da "comunità open", nelle quali gli utenti condividono scopi più generici (recuperare musica da Internet), piuttosto che obiettivi più specifici, come partecipare alla simulazione di un fenomeno fisico. Per questa ragione, nella maggior parte dei sistemi P2P non si ha una fase di autenticazione, ma piuttosto offrono protocolli per garantire l'anonimato.

2.4.2 Connessione: Differenze

Le Griglie sono caratterizzate da potenti macchine che sono staticamente connesse ad una rete ad alte prestazioni con un alto livello di disponibilità. D'altra parte, il numero di nodi accessibili è basso perché le risorse sono vincolate da rigorosi meccanismi di sicurezza.

Contrariamente, nei sistemi P2P sono connessi utenti tramite dei comuni PC, rimanendo disponibili per un limitato periodo di tempo e con una ridotta affidabilità. Il numero di nodi che fanno parte di una rete P2P, in un dato istante, è molto più grande rispetto ai partecipanti di una Griglia. Infatti, le procedure per la connessione di un nuovo peer a quest'ultima risulta sempre troppo rigido.

2.4.3 P2P e Griglie: convergenza

Nonostante questa breve introduzione risalti le differenze tra il P2P e le Griglie, questi due sistemi tendono a convergere [4] per tre motivi fondamentali:

1. devono risolvere lo stesso problema, ovvero, l'organizzazione della condivisione delle risorse all'interno di comunità virtuali;
2. entrambi tentano di risolvere la problematica sopra descritta sfruttando la stessa strategia, ovvero, la creazione di una rete di overlay che coesiste con la struttura sottostante (non è detto che esista una corrispondenza);
3. la natura complementare di questi due sistemi suggerisce che gli interessi delle due comunità possano avvicinarsi nel tempo.

2.5 P2P e Griglie: Scoperta di Risorse e Dinamicità

Una funzionalità fondamentale delle Griglie è la capacità di individuare la locazione di una risorsa che soddisfa particolari vincoli. Questo accade quando un utente sottomette un job, specificando le caratteristiche delle risorse computazionali che dovranno eseguirlo, come quantità di memoria, spazio disco, sistema operativo.

Il problema della scoperta di risorse è molto complesso, soprattutto se quest'ultime sono di tipo dinamico. In ambiente Grid, per individuare una certo numero di risorse computazionali vengono utilizzate le *range query multi-attributo*. Un esempio di range query multi-attributo è il seguente:

$$Q = \{R \in \{R_1, ..., R_N\} | CpuSpeed[R] \geq 2.0GHz$$
$$\text{and } RamSize[R] \geq 512MB$$
$$\text{and } 100MB \leq FreeSpace \leq 300MB\}$$

In questo caso, si richiede risorse computazionali, la cui velocità di CPU è almeno $2.0GHz$, almeno $512MB$ di RAM e uno spazio disco libero compreso tra 100 e $300MB$. In letteratura, troviamo molti sistemi P2P con architettura DHT, che forniscono soluzioni al problema della scoperta di risorse e supportano *range query multi-attributo* [1, 2, 6, 10, 11]. Ad esempio, SQUID supporta ricerche con *wildcard (*)*, con parole chiave parziali e per range di valori e garantisce che l'informazione verrà trovata se esiste nel sistema [2]. Nonostante i buoni risultati raggiunti, il problema principale delle reti basate su DHT è collegato al fatto che ogni cambiamento nel valore di un attributo, richiede la *re-indicizzazione* dell'oggetto [7] sulla rete di overlay. Mentre soluzioni P2P meno strutturate rispetto alle DHT sono più adeguate per gestire dati dinamici [7].

2.6 P2P e Griglie: Sistemi Strutturati

In questo paragrafo, vengono descritti dei sistemi P2P, presenti in letteratura, che sfruttano la tecnologia DHT per risolvere le problematiche legate alla scoperta di risorse nelle Griglie. Ognuno di essi fornisce una propria soluzione per supportare le range query e gestire efficientemente le indicizzazioni degli attributi dinamici.

2.6.1 A DHT-based Peer-to-Peer Framework for Resource Discovery in Grids

L'obiettivo principale evidenziato in [6] è quello di proporre un framework per la scoperta di risorse in ambiente Grid, che si basa su architetture di tipo DHT. Tale framework, nasce per fornire due tipi di funzionalità: rendere possibile l'individuazione di risorse dinamiche e permettere all'utente di definire sia query arbitrarie che per range di valori.

Definiamo due concetti fondamentali:

1. La **classe Risorsa**, che rappresenta un modello per descrivere risorse dello stesso tipo. Ogni classe è definita da un'insieme di attributi che specificano le sue caratteristiche.

2. La **Risorsa**, è un'istanza della classe Risorsa. Ogni Risorsa ha uno specifico valore per ogni attributo.

Gli attributi che descrivono una classe Risorsa possono essere **statici** o **dinamici**:

1. **Statici**, se non cambiano frequentemente, come il *nome del Sistema Operativo* e la *frequenza della CPU*.

2. **Dinamici**, se si hanno rapidi cambiamenti, come nel caso del *carico della CPU* e la *quantità di memoria libera*.

Inoltre, vengono definiti tre tipi di query per la scoperta di risorse: **match esatto**, **range di valori** e **query arbitrarie**. Le **multi-attribute query** sono composte da più sotto-query, una per ogni attributo. Queste sotto-query appartengono a uno dei tre tipi elencati sopra. Nel sistema descritto, le DHT vengono utilizzate per le query con match esatto e per range di valori su attributi statici, mentre la tecnica del flooding viene utilizzata per ricercare le risorse dinamiche e per le query arbitrarie su attributi statici. Tutto ciò viene riassunto sinteticamente nella figura 2.7.

	Risorse Statiche	Risorse Dinamiche
Query Esatte	DHT	**Flooding**
Range Query	DHT	**Flooding**
Query Arbitrarie	**Flooding**	**Flooding**

Figura 2.7: Tabella riassuntiva

L'architettura. Il sistema può definire più di una classe di risorse, dove ogni attributo statico che le descrive, fa parte di una DHT diversa. Le query esatte o per range di valori, su attributi *statici*, vengono eseguite sfruttando la DHT corrispondente. Quest'ultime sono DHT standard, dove gli identificatori dei nodi e delle risorse vengono mappati sullo stesso anello. In più, viene aggiunta una DHT **general purpose**, per gestire le query arbitrarie su attributi statici e le query su attributi dinamici. In questo caso, soltanto gli identificatori dei nodi vengono mappati sull'anello, in altre parole, non ci sono puntatori alle risorse. La DHT general purpose sfrutta il **broadcast** per raggiungere tutti i nodi.

Conclusioni. In [6], viene presentato un framework per supportare la scoperta di risorse in ambiente Grid, che si basa su un sistema DHT. Il vantaggio di sfruttare una *DHT general purpose* si ha nell'istante in cui le risorse dinamiche, messe a disposizione da un nodo, subiscono degli aggiornamenti. Infatti, utilizzando questa soluzione, dove soltanto gli identificatori dei nodi vengono mappati sull'anello, tali aggiornamenti non provocano una successiva re-indicizzazione del nodo sulla rete. Il problema però,

si ha nella ricerca dei peer che sono potenziali match per una query su attributi dinamici, infatti anche se viene proposto un broadcast efficiente [8], comunque sia abbiamo sempre che il costo della ricerca è lineare nel numero dei nodi. Inoltre, come evidenziato in [7], sistemi che associano una DHT per ogni attributo mostrano problemi di scalabilità con l'aumento del numero degli attributi stessi.

2.6.2 Scalable, efficient range queries for grid information services

In [10] viene proposta un'estensione di CAN per supportare le range query e il routing viene fornito di nuove funzionalità per gestire efficientemente i frequenti cambiamenti dei dati. Le risorse condivise sulla rete vengono descritte da un'insieme di attributi, se questi ultimi sono di tipo statico, vengono indicizzati tramite una DHT tradizionale, mentre per tutti gli attributi che assumono valori "continui" viene utilizzata l'estensione di CAN. Per localizzare una risorsa, tale infrastruttura interroga, per ogni attributo presente nella query, la DHT appropriata ed infine viene eseguita un'intersezione dei risultati.

Un sottoinsieme dei server che partecipano alla Griglia, sono utilizzati per far parte della rete P2P in questione, preoccupandosi di memorizzare le coppie <valore dell'attributo, identificatore>. Questi prendono il nome di **interval keeper (IK)** ed ognuno di loro è responsabile di un sottointervallo di valori appartenente all'intervallo $[0.0, 1.0]$. Ogni server della Griglia informa l'appropriato IK in base al valore corrente dell'attributo. Gli autori propongono tre strategie di routing per propagare le richieste sulla rete e sfruttano meccanismi di caching per minimizzare l'overhead di comunicazione durante la fase di aggiornamento degli attributi.

Conclusioni. In [10] viene proposto un sistema, che tramite simulazione, è in grado di gestire efficientemente sia le range query che l'aggiornamento degli attributi. D'altra parte, l'impiego di una DHT comporta elevati costi di manutenzione per la rete di overlay.

2.6.3 Peer-to-Peer Discovery for Computantional Resources for Grid Applications

L'obbiettivo principale del sistema P2P, proposto in [11], è realizzare un protocollo per la scoperta di risorse nelle Griglie, che sia *robusto*, *scalabile*, *efficiente* e che sia in grado di supportare *query complesse*.

Il sistema sfrutta la rete di overlay di Pastry e si preoccupa di mappare su tale DHT le risorse, rappresentate come collezione di attributi. La tattica adottata si basa sul combinare la parte statica e dinamica degli attributi della risorsa per ottenere un *ResourceID*. Gli attributi dinamici hanno la caratteristica di assumere valori continui, quindi possono essere discretizzati, rappresentandoli tramite una percentuale, utilizzando un numero specifico di bit. In questo modo, per individuare lo stato esatto della risorsa basta combinare la rappresentazione statica con quella dinamica. L'obiettivo principale di questo tipo di naming è quello di inserire l'informazione in un punto sull'anello della DHT sfruttando il valore degli attributi statici. Successivamente, costruire un arco, a partire da quel punto, che descriva la disponibilità degli attributi dinamici (figura 2.8).

Quindi, l'inizio dell'arco rappresenta una risorsa caratterizzata da un'insieme di attributi statici e la sua lunghezza rappresenta lo stato attuale della risorsa. Il concetto chiave è che l'anello della DHT contiene solo un numero finito di nodi, mentre esiste un numero infinito di configurazioni per gli attributi dinamici.

L'inserimento di risorse nel sistema si basa su due concetti fondamentali. Primo, questi dati dovrebbero essere il più possibile distribuiti sulla DHT, per ottenere maggiore scalabilità. Secondo, la configurazione statica della risorsa dovrebbe essere rappresentata con un arco che rappresenti il suo grado di disponibilità. Tutto questo è possibile eseguendo le operazioni mostrate in figura 2.9.

Il risultato è un *ResourceID* di 160 bit che soddisfa entrambi i criteri sopra citati.

Per quanto riguarda la gestione delle query, gli autori propongono tre euristiche, mentre i messaggi di aggiornamento vengono inviati o periodicamente oppure quando

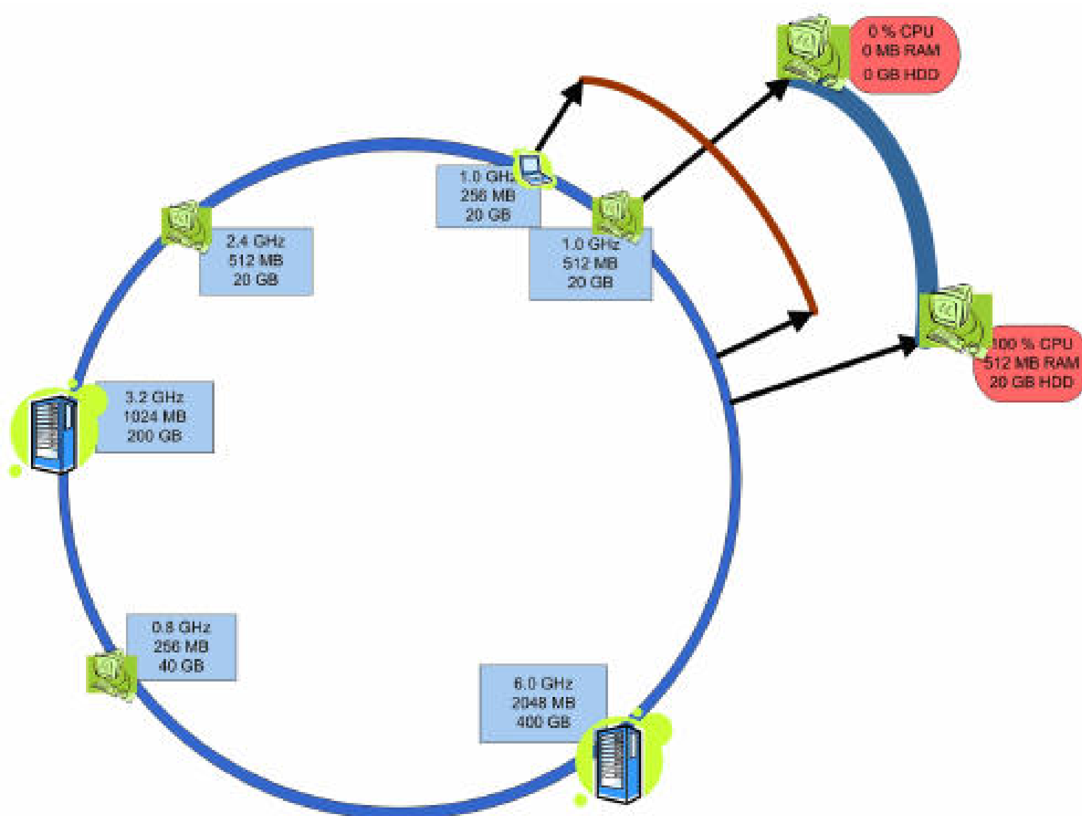


Figura 2.8: Architettura del Sistema

si osserva un cambiamento importante nei valori della risorsa.

Conclusioni. Il sistema proposto in [11] indicizza le risorse sulla DHT in un modo originale ed interessante. Ma a causa di questo tipo di indicizzazione, il sistema soffre di una limitazione non trascurabile. Infatti, se un utente richiede una risorsa che abbia almeno 256 MB di memoria RAM, l’algoritmo per la risoluzione della query andrebbe ad interrogare tutti i peer che hanno:

- 256 MB come attributo statico con disponibilità pari al 100
- 512 MB di RAM con disponibilità pari al 50

In conclusione, il sistema non gestisce efficientemente la risoluzione delle range query.

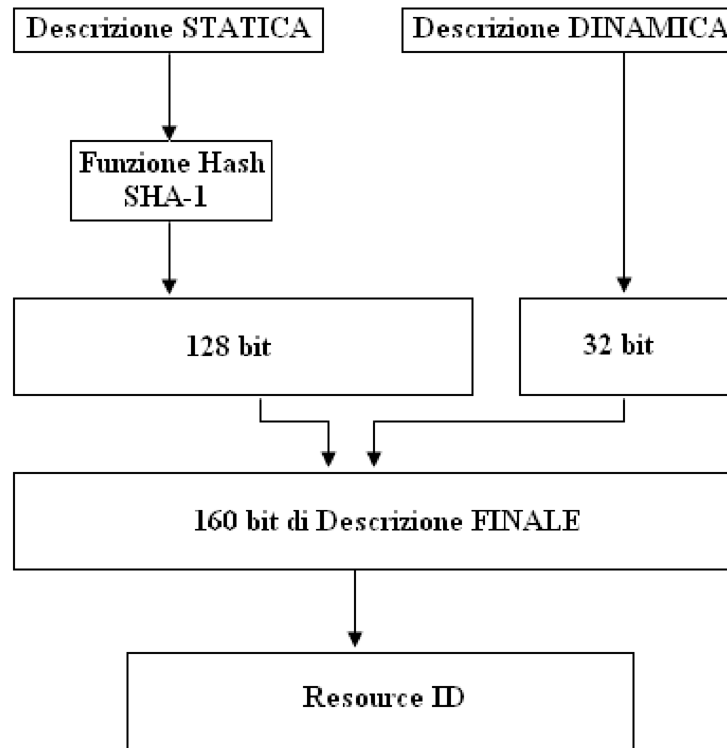


Figura 2.9: Costruzione del ResourceID

2.6.4 SQUID

Squid è un sistema P2P per la scoperta di risorse, il cui obbiettivo è quello di gestire le query complesse e garantire che venga trovata l'informazione, se è presente nel sistema [1, 2].

La sua architettura si basa su una Distributed Hash Table, ma differisce da una vera e propria DHT per la modalità con cui le informazioni vengono mappate nello spazio degli indirizzi. In sistemi esistenti, come Chord e CAN, quest'operazione viene realizzata tramite l'uso di una funzione hash che assegna ai dati un identificatore. Come risultato, le informazioni sono distribuite casualmente tra i peer, senza considerare nozioni di località. Diversamente, Squid tenta di preservare la località utilizzando lo **Hilbert Space Filling Curves (HSFC)**. Lo HSFC è una funzione definita come segue:

$$f : N^d \rightarrow N$$

e mappa uno spazio *d-dimensionale* in uno spazio a *1-dimensione*.

Ogni dato condiviso viene descritto utilizzando un'insieme di parole chiave (*keywords*); quest'ultime formano uno spazio multidimensionale e ogni dato può essere visto come un punto al suo interno. Lo HSFC viene utilizzato per mappare lo spazio multidimensionale in uno *spazio degli indici* a 1 dimensione.

Squid utilizza Chord per realizzare la sua rete di overlay. Gli identificatori dei nodi sono generati casualmente, il loro valore appartiene all'intervallo $[0, 2^m - 1]$ e vengono ordinati su un anello modulo 2^m. I dati vengono assegnati al primo nodo il cui *ID* risulta essere uguale o maggiore all'indice del dato, quest'ultimo calcolato tramite la funzione HSFC.

Squid processa una query in due passi fondamentali, ovvero, traduce la query in un'area significativa nello spazio degli indici ed interroga i nodi corrispondenti per identificare il dato. Gli autori propongono anche un'ottimizzazione nel routing per limitare il numero di nodi coinvolti nella risoluzione di una richiesta.

Conclusioni. In [1, 2] viene presentato un sistema per la scoperta di informazioni che supporta query complesse e che garantisce che tutti gli elementi presenti nel sistema che hanno un match con la query vengano trovati, con costi limitati nel numero dei messaggi e dei nodi coinvolti nella risoluzione di una query. Gli esperimenti effettuati dimostrano la scalabilità del sistema.

2.6.5 Conclusioni

In questo paragrafo sono stati presentati dei sistemi P2P che sfruttano la tecnologia DHT per risolvere le problematiche legate alla scoperta di risorse nelle Griglie. Ognuno di questi, tramite gli esperimenti effettuati, riesce a dimostrare di raggiungere delle buone prestazioni. Ma l'aspetto importante da sottolineare è che, come si evidenzia in [7], i sistemi che sfruttano una rete di overlay organizzata secondo una DHT, non rappresentano la soluzione migliore in presenza di dati altamente dinamici, come ad

esempio il carico di lavoro della CPU. Infatti, nella maggior parte dei sistemi visti [1, 2, 10, 11], ogni volta che il valore della risorsa cambia, occorre re-indicizzare l'oggetto sulla rete.

2.7 P2P e Griglie: Sistemi Non Strutturati

In questo paragrafo, vengono descritti dei sistemi P2P non strutturati, presenti in letteratura, per risolvere le problematiche legate alla scoperta di risorse nelle Griglie.

2.7.1 A Grid Information Service based on Peer-to-Peer

L'obiettivo principale di questo sistema [5] è quello di presentare un Grid Information Service (GIS) che si basa su tecnologie Peer-to-Peer e Routing Index (RI) [9]. Questo sistema nasce dalla necessità di superare i limiti del Globus MDS-GT2 [24], la cui architettura risultava centralizzata e quindi includeva tutte le problematiche del caso, ovvero i server diventano potenziali colli di bottiglia o punti di fallimento.

I peer sono organizzati in cluster e alcuni nodi, chiamati Super-peer, hanno la funzionalità di server. All'interno dei cluster si adottano politiche di ridondanza dei super-peer, per due motivi fondamentali:

1. le repliche mantengono la copia dei dati, in questo modo se all'interno di un cluster, un super-peer fallisce, il sistema continua a funzionare.
2. Il carico di lavoro può essere condiviso dai due super-peer.

Per contro abbiamo che i costi nelle comunicazioni aumentano.

1. Quando un nodo si unisce al cluster oppure quando si ha un aggiornamento, deve inviare K-messaggi se ho una rete di ridondanza K.
2. 2. Esistono $O(K^2)$ connessioni tra due super-peer di ridondanza K.

Il sistema è formato da due entità fondamentali, gli **Agent** che sono responsabili di pubblicare le informazioni dei nodi ai super-peer, e gli **Aggregator** che sono in esecuzione su quest'ultimi e si preoccupano di collezionare i dati, rispondere alle query per inoltrarle agli altri super-peer.

Gli Agent pubblicano le informazioni rese disponibili dagli **Information Provider (IP)**. Gli IP, schedulati periodicamente dagli Agent, interrogano la risorsa e memorizzano l'informazione sotto forma di **Service Data Elements (SDE)**. Esiste un Information Provider per ogni risorsa pubblicata da un nodo.

Il **Routing Indices (RI)** [9] viene utilizzato per migliorare le performance del routing e per prevenire fenomeni di saturazione della rete a causa di un elevato numero di messaggi. La tecnica RI decide a quale nodo inviare la query, tenendo conto della disponibilità che, il potenziale destinatario, ha del dato richiesto. Il sistema implementa **Hop Count Routing Indices (HRI)** [9], che considera il numero di hop necessari per raggiungere l'informazione. Quando un nodo effettua una query, questa viene inviata al super-peer del cluster di appartenenza che si preoccuperà di selezionare il miglior vicino, altrimenti, se necessario, viene inoltrata ad altri super-peer.

Conclusioni. I test effettuati mostrano che questo sistema scala efficientemente [5], ma come tutti i sistemi P2P non strutturati, si evidenziano due limiti importanti:

1. l'algoritmo potrebbe fallire nella ricerca delle risorse;
2. l'algoritmo potrebbe interrogare molti Aggregator rispetto a quelli strettamente necessari.

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

In [7] vengono presentati due sistemi P2P basati su tecnologie **Routing Index (RI)** [9]. Entrambi hanno come obiettivo principale quello di raggiungere un buon trade-off tra la gestione efficiente del routing delle query e la necessità di limitare i messaggi relativi ad aggiornamenti quando si verificano cambiamenti nei valori degli attributi condivisi. Il primo sistema, utilizza un albero come rete di overlay, mentre il secondo, che nasce come evoluzione del primo, si basa su una rete gerarchica a due livelli, dove la struttura ad albero viene mantenuta nel livello più basso della gerarchia.

La prima soluzione prende il nome di **Tree Vector** e mantiene uno spanning tree su tutto l'insieme dei peer. Ogni nodo della rete gestisce, per ogni attributo che caratterizza la risorsa gestita, un **bitmap index**. Quest'ultimo viene utilizzato per rappresentare la risorsa locale, ma anche come descrizione delle risorse presenti in ogni vicino. Grazie alla loro semplicità, questi indici possono essere facilmente aggiornati se il valore di qualche attributo cambia nel tempo. Quando un nodo P riceve una query da un suo vicino P_{IN}, la query viene inoltrata ai rimanenti vicini che hanno un potenziale match. Questo tipo di informazione, P la ricava dal bitmap index. L'aggiornamento viene eseguito quando un nodo P nota un cambiamento. Se il bitmap index del nuovo valore è identico al bitmap index del vecchio valore, rimane tutto invariato. Altrimenti, viene costruito un nuovo indice e spedito a tutti i vicini.

Malgrado le buone performance che questa prima soluzione riesce a raggiungere, mantenere un albero bilanciato può risultare difficile da gestire, soprattutto in presenza di frequenti inserimenti e cancellazioni dalla rete di overlay, da parte dei peer. Inoltre, i nodi vicini alla radice risultano inondati di richieste. 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, si definiscono due indici:

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 quel link esterno.

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.

Conclusioni. In [7] vengono presentati due sistemi che gestiscono efficientemente i frequenti aggiornamenti di una risorsa, senza congestionare la rete per informare tutti gli altri. Inoltre, la seconda soluzione, oltre a mantenere le buone prestazioni che ha il primo, mette in evidenza la semplicità di gestione della rete di overlay. D'altra parte però, il Rounting Index potrebbe risultare meno preciso in quanto non si ha più una rete strutturata.

2.7.3 Conclusioni

I sistemi P2P non strutturati, presentati in questo paragrafo, tramite l'uso del Routing Index, mostrano di saper gestire efficientemente la distribuzione dell'indice globale tra i peer della rete. Inoltre, il sistema supporta i frequenti aggiornamenti dei valori degli attributi delle risorse, senza la necessità di notificare il cambiamento a tutti i nodi. Chiaramente, l'uso di architetture "meno strutturate" non garantisce l'individuazione di tutti i possibili match per una query.

Capitolo 3

Approcci basati su tessellazione di Voronoi

Negli ultimi anni, le reti P2P strutturate come Chord [20] o Pastry [22] hanno riscontrato un grande interesse. Questi sistemi presentano un buon grado di efficienza e utilizzano funzioni hash per garantire il bilanciamento del carico. Ai nodi vengono assegnati degli **identificatori** in uno spazio uniformemente popolato. Una funzione hash viene applicata al contenuto di un file e restituisce la sua **chiave**, che è usata per organizzare la risorsa nel medesimo spazio degli identificatori dei nodi. Questi sistemi forniscono meccanismi di ricerca esatta e non sono in grado di risolvere query per range di valori. Ciò dipende dall'uso del **consistent hashing**, grazie al quale le risorse sono distribuite casualmente tra i peer, ma viene persa ogni nozione di località [2].

Ultimamente sono stati proposti approcci alternativi, tra questi i più interessanti sono quelli basati sui diagrammi di Voronoi.

Il capitolo è così organizzato: nel paragrafo 3.1 vengono descritti i diagrammi di Voronoi e le loro proprietà. Nella sezione 3.2 viene definito il modello dello **small-world** proposto da **Kleinberg**[13, 14]. Nei paragrafi 3.3, 3.4 e 3.5 vengono presentati rispettivamente VoroNet [12], SWAM-V [49] e VON [31], sistemi P2P le cui overlay network sono organizzate secondo una tessellazione di Voronoi. Nella sezione 3.6

vengono descritte altre applicazioni di tali diagrammi in ambito scientifico. Infine, nell'ultimo paragrafo viene spiegato il compass routing e le sue proprietà.

3.1 Diagrammi di Voronoi

I diagrammi di Voronoi [57] 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 [42].

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 punti p_i e p_j (figura 3.1a).

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 punti p_i, p_j e p_k e coincide con il centro del cerchio passante per quei tre punti (figura 3.1b).

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.

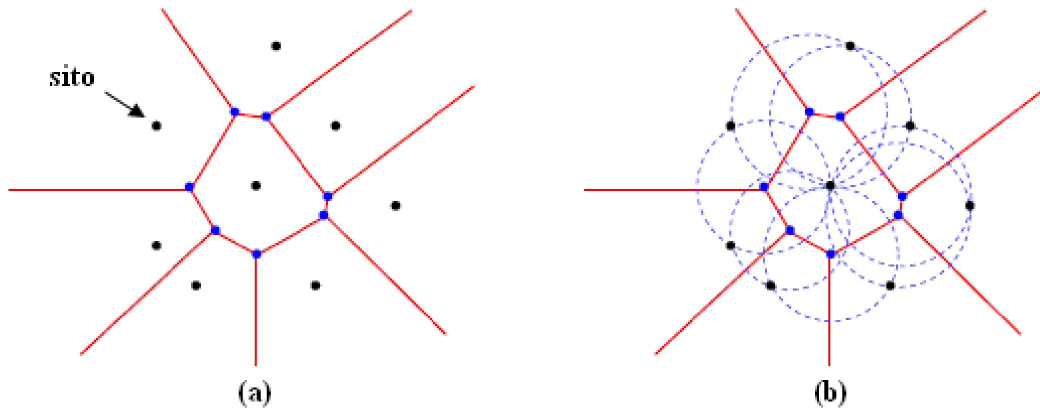


Figura 3.1: Diagramma di Voronoi.

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 [46, 55].

3.1.1 Triangolazione di Delaunay

La definizione della triangolazione di Delaunay si basa sui diagrammi di Voronoi attraverso il principio della dualità. Due punti 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 punti contigui (figura 3.2).

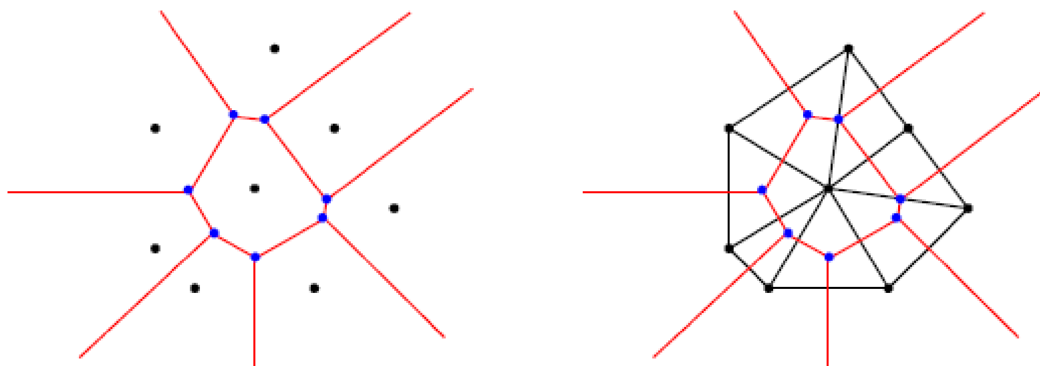


Figura 3.2: Diagramma di Voronoi e triangolazione di Delaunay.

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 (figura 3.3) [56].

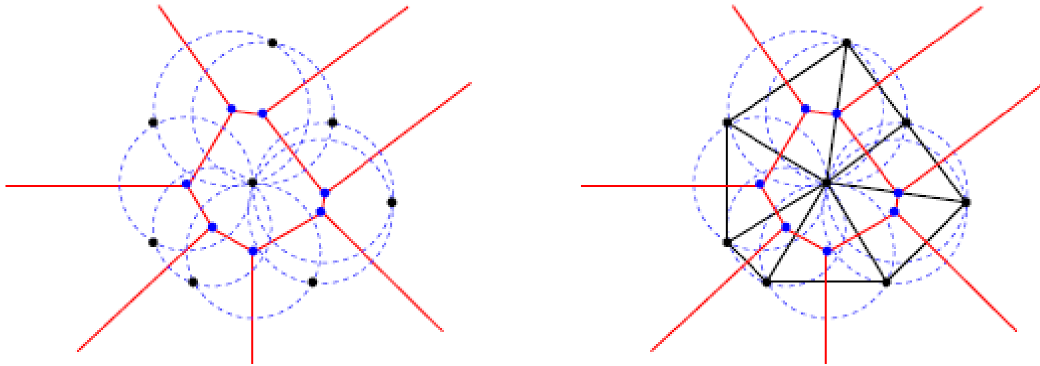


Figura 3.3: Proprietà della triangolazione di Delaunay.

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.

3.2 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[1].

Watts e Strogatz [58] hanno ripreso l'idea di Milgram con lo scopo di definire un routing efficiente in reti P2P. In [58] 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 Kleinberg 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 [13, 14] ogni nodo 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 3.4.

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

La rete VoroNet descritta nel paragrafo successivo si basa sui diagrammi di Voronoi

[1] Questa quantità fu associata alla teoria dei sei gradi di separazione, proposta nel 1929 da dallo scrittore ungherese Frigyes Karinthy in un racconto breve intitolato *Catene* [41].

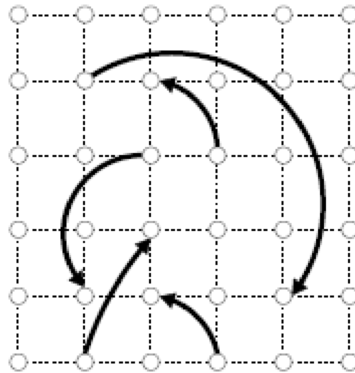


Figura 3.4: Le linee tratteggiate indicano i vicini diretti di un nodo, mentre le frecce in grassetto rappresentano i vicini remoti.

e riprende il modello di Kleinberg con l'obiettivo di realizzare un routing poli-logaritmico nel numero dei nodi partecipanti.

3.3 VoroNet: un overlay basato su Voronoi

VoroNet [12], 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, 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 polilogaritmico nel numero dei nodi partecipanti. Per questa ragione il sistema si ispira al modello dello small-world proposto da Kleinberg [13, 14].

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

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.

[2] Queste assunzioni sono le stesse dell'articolo di VoroNet[12].

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 3.5. I tre insiemi di vicini, descritti sopra, sono necessari per garantire un **routing poli-logaritmico**.

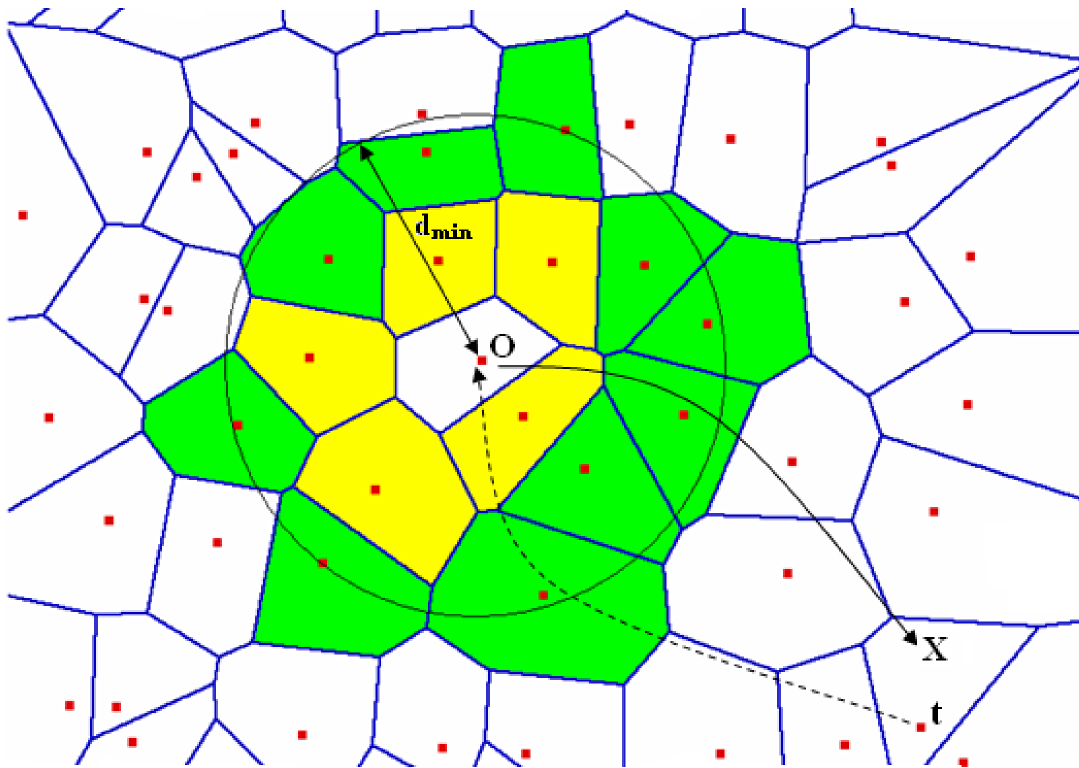


Figura 3.5: Vicini definiti in VoroNet.

Come possiamo vedere in figura 3.5, 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 [12] 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[3].

In conclusione, possiamo dire che la dimensione delle strutture dati memorizzate da ogni nodo[4], 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 nodo è limitata ad un numero costante di vicini. Come conseguenza otteniamo che, un peer, per informare gli altri di avvenuti cambiamenti, invierà un numero limitato di messaggi.

3.3.1 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

[3] Tale dimostrazione è valida soltanto nel caso bi-dimensionale, infatti, algoritmi per costruire i diagrammi di Voronoi in k dimensioni esistono ($k > 2$), ma il numero medio dei vicini non è più limitato. In questo modo è impossibile garantire che tutte le strutture dati siano dell'ordine di $O(1)$ [12].

[4] Possiamo considerare i termini nodo o peer come sinonimi della parola oggetto, in quanto, per semplicità abbiamo fatto l'ipotesi che un nodo fisico può pubblicare un solo oggetto sulla rete.

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 nodi sulla rete.

3.3.2 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.

3.3.3 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.

3.3.4 VoroNet: Conclusioni

VoroNet è un sistema P2P strutturato, la sua particolarità sta nel fatto che indicizza oggetti piuttosto che nodi fisici. **Tale operazione viene eseguita in modo da rispettare criteri di similitudine, ovvero, oggetti con caratteristiche simili risultano vicino nella rete di overlay. In questo modo, VoroNet è in grado di supportare e gestire in maniera efficiente la risoluzione di range query.**

Una caratteristica importante di questo sistema è l'organizzazione dello spazio degli attributi secondo un diagramma di Voronoi, in questo modo VoroNet realizza un routing efficiente memorizzando un numero di vicini che è di ordine $O(1)$ [12].

Infine, come abbiamo visto in questo capitolo, VoroNet si basa sul modello dello small-world proposto da Kleinberg: per ogni peer mantiene sia dei vicini diretti che dei vicini remoti, garantendo che ogni coppia di nodi sia separata da cammini brevi, di lunghezza poli-logaritmica nel numero degli oggetti pubblicati.

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.

3.4 SWAM-V

SWAM-V (Small World Access Methods based on Voronoi diagram) [49] 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 (figura 3.6).

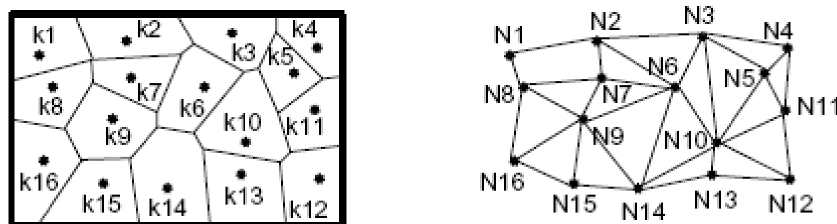


Figura 3.6: Diagramma di Voronoi per lo spazio delle chiavi e Triangolazione di Delaunay corrispondente.

Inoltre, vengono aggiunti dei *random link* in accordo al grafo definito dallo Small World di Kleinberg (figura 3.7).

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.

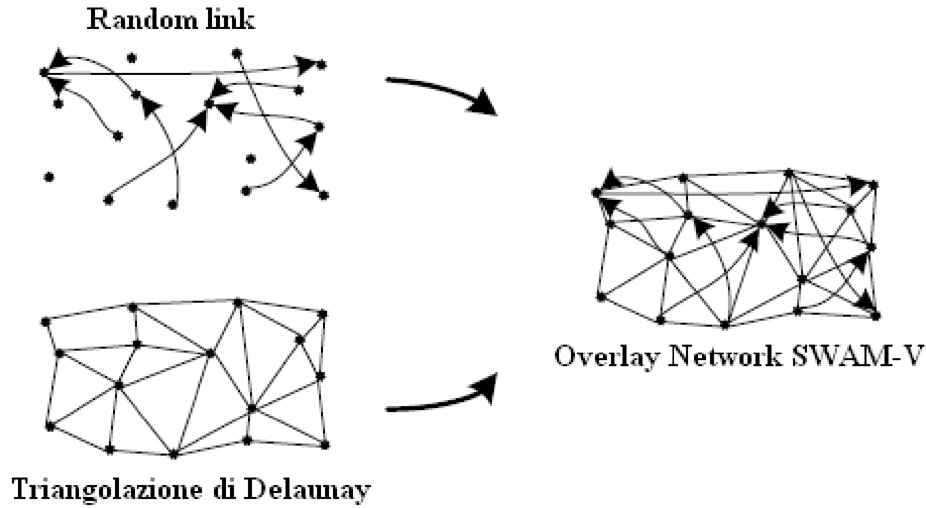


Figura 3.7: Overlay network di SWAM-V.

Definiamo la norma L_p, con $p \in Z^+$, la funzione che misura la "distanza" tra due 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 nodo 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 $L_p(\overrightarrow{k'} - \overrightarrow{q}) = r$ (figura 3.8).

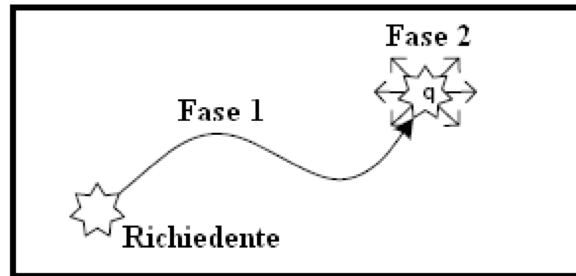


Figura 3.8: Risoluzione di una range query.

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.

3.5 VON

Un **Massively Multiplayer Online Game (MMOG o MMO)** è un gioco per computer che è capace di supportare centinaia o migliaia di giocatori contemporaneamente, connessi tra di loro tramite Internet. Tipicamente, questo tipo di gioco si ambienta in un gigantesco mondo virtuale [30], detto **Networked Virtual Environment (NVE)**.

Gli NVE, rappresentano dei mondi sintetici nei quali ogni utente assume un'identità virtuale (**avatar**) e può interagire con giocatori umani od artificiali. Gli utenti possono effettuare differenti azioni, come spostarsi in una nuova locazione, utilizzare degli oggetti, iniziare conversazioni o sfidare altri giocatori. Tali azioni, vengono codificate in messaggi che sono spediti sulla rete, per aggiornare lo stato dei vicini interessati. Per questa tipologia di sistemi è importante raggiungere caratteristiche come **consistenza**, **scalabilità**, **reattività**, **persistenza**, **affidabilità** e **sicurezza** [31]. Ciò è fondamentale per rendere il gioco attraente e il più reale possibile, fornendo insieme delle tecniche per salvaguardare la privacy dei giocatori.

VON (Voronoi-based Overlay Network) [31], è un sistema che nasce con l'obiettivo di fornire maggiore scalabilità agli NVE, dove per scalabilità, si intende la capacità da parte di un NVE di gestire una grande quantità di utenti senza compromettere la qualità del servizio. In un generico NVE, i partecipanti vengono rappresentati come punti in uno spazio 2D, ed ognuno di essi è caratterizzato da un'**area di interesse (AOI - Area Of Interest)**. Quest'ultimo è un concetto fondamentale, in quanto ogni utente viene influenzato soltanto da nodi od eventi che sono all'interno dell'AOI. Supponiamo che l'area di interesse sia una circonferenza con centro le coordinate dell'utente, e che un nodo si muova con passi discreti. Nasce il seguente problema: come avviene la scoperta dei nodi rilevanti[5], se il nodo è periodicamente in movimento? Gli autori di VON [31] utilizzano i **diagrammi di Voronoi** per risolvere il problema della scoperta dei vicini in maniera completamente distribuita. Ogni nodo mantiene una

[5] I peer che giacciono nell'area di interesse di un nodo.

vista del sistema, ovvero, è connesso ad un'insieme di vicini, che vengono classificati in tre sottoinsiemi. Dato il nodo A, definiamo:

- **AOI Neighbors**, tutti i nodi che giacciono all'interno dell'area di interesse di A;
- **Enclosing Neighbors**, tutti i nodi la cui regione confina con quella di A.
- **Boundary Neighbors**, sono tutti quei nodi che risiedono nell'area di interesse e che hanno almeno un Enclosing Neighbor che giace al di fuori dell'area di interesse di A.

I vicini sopra elencati sono visibili in figura 3.9.

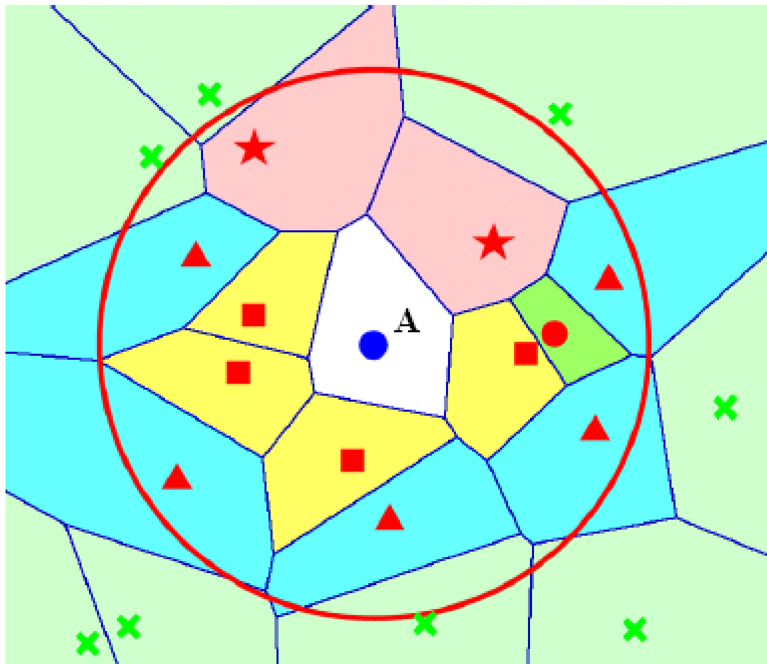


Figura 3.9: La circonferenza in rosso rappresenta l'area di interesse del nodo A. I quadrati sono gli Enclosing Neighbors; i triangoli sono i Boundary Neighbors; le stelle sono sia Enclosing che Boundary Neighbors; le croci rappresentano ii vicini irrilevanti per il nodo A. Tutti gli altri si definiscono AOI-Neighbors (pallino rosso).

A livello di rete, ogni nodo mantiene una connessione diretta con tutti i vicini che appartengono all'area di interesse. Inoltre, la vista del sistema ha una dimensione ridotta e ciò comporta costi bassi per la manutenzione della rete di overlay.

In conclusione possiamo dire che VON riesce a soddisfare i requisiti di:

- **scalabilità**, in quanto ogni nodo memorizza un numero limitato di vicini;
- **reattività**, perché la latenza viene minimizzata grazie alle connessioni dirette tra i nodi.

Inoltre, dimostrano che la scoperta di nuovi vicini comporta l'invio di un numero di messaggi, limitato dalla cardinalità dell'insieme degli AOI Neghbors.

3.6 I diagrammi di Voronoi: altre applicazioni

3.6.1 Astronomia

I diagrammi di Voronoi sono stati applicati in Astronomia per trovare le stelle centrali all'interno delle galassie [32], oppure, per calcolare la dimensione dell'area di quest'ultime [33]. Inoltre, grazie alla tessellazione di Voronoi sono stati individuati cluster di galassie, a partire dall'osservazione della loro densità locale e dei loro vicini [34].

3.6.2 Biologia

I Biologi hanno rappresentato differenti specie di piante utilizzando i diagrammi di Voronoi, descrivendole tramite la loro dimensione e locazione. Inoltre, hanno dimostrato che tali diagrammi sono il metodo che meglio descrive il cambiamento di questi valori durante la fase di crescita delle piante [35]. Infine, è importante sottolineare che la tessellazione di Voronoi è stata utilizzata anche per calcolare il volume e la struttura delle proteine [36, 37].

3.6.3 Chimica

I diagrammi di Voronoi, in Chimica, sono stati utilizzati per definire la struttura degli atomi di ossigeno in uno spazio tri-dimensionale [38], ma anche per sviluppare modelli relativi ai fluidi in uno spazio bi-dimensionale [39].

3.6.4 Geologia

I Geologi hanno simulato il flusso della lava che fuoriusciva da un vulcano, partizionando la sua superficie in numerosi poligoni di Voronoi [41], come si vede in figura 3.10.

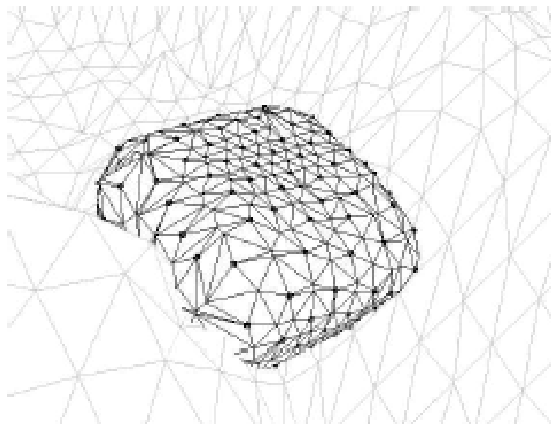


Figura 3.10: Superficie di lava che viene rappresentata tramite la tessellazione di Voronoi.

3.7 Compass Routing

Il **compass routing** [47] è un algoritmo di instradamento che viene applicato sui grafi geometrici[6].

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. Di seguito, descriviamo le sue proprietà in modo formale:

> 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

[6]Un **grafo geometrico** è un grafo nel quale i vertici o i lati sono associati a oggetti o configurazioni geometriche [59].

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.

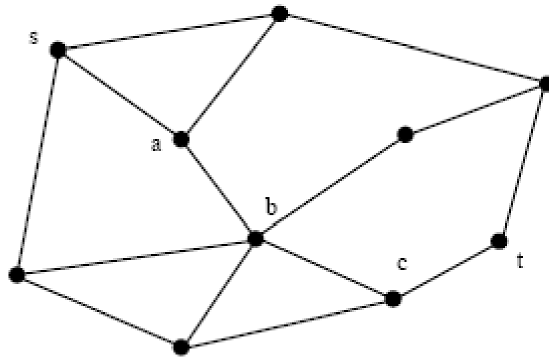


Figura 3.11: Cammino risultante dall'applicazione del compass routing da s a t.

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 3.12.

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 coppia 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

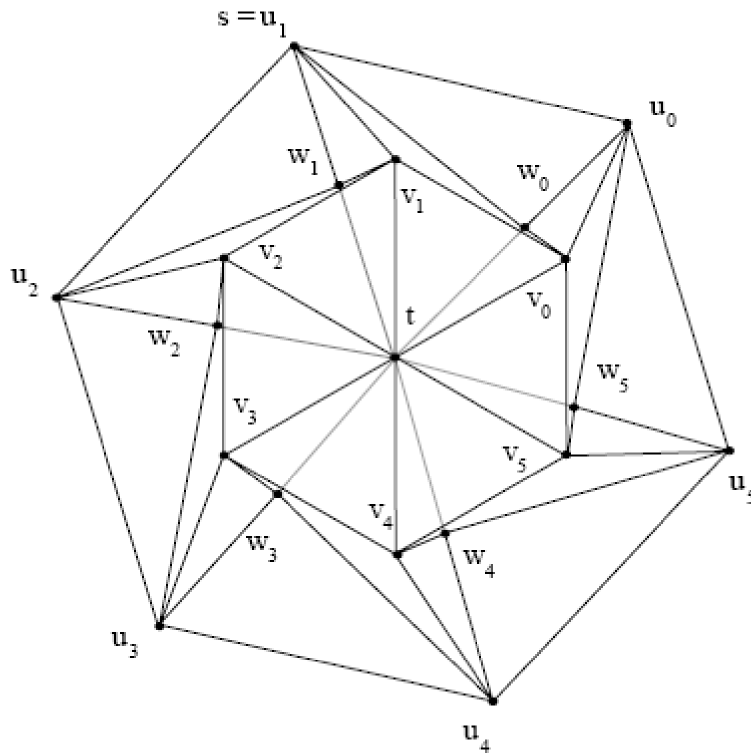


Figura 3.12: Il compass routing non riesce a raggiungere t da u_i, $i = 0, ..., 5$.

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 (figura 3.13).

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 [47].

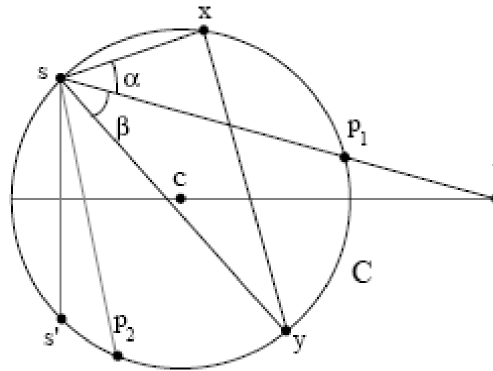


Figura 3.13: Routing sulla triangolazione di Delaunay.

□

Questa tecnica di instradamento permette di costruire uno 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, figura 3.14a.

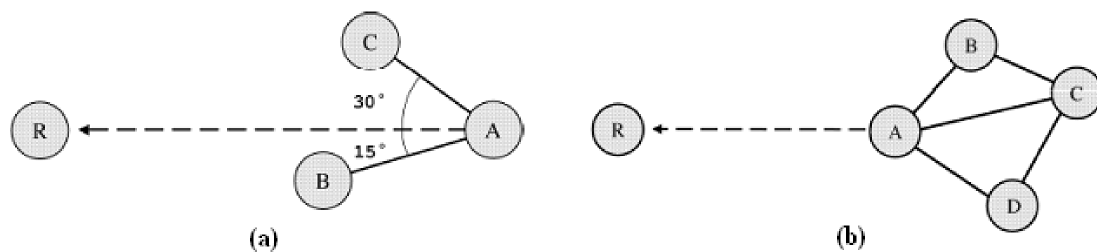


Figura 3.14: Compass Routing.

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 dal momento che, $\angle RCA < \angle RCB$ e $\angle RCA < \angle RCD$, (figura 3.14b). Se ogni nodo effettua il calcolo sopra descritto, viene definito uno spanning tree con radice R [48].

In questo paragrafo abbiamo descritto l'algoritmo di compass routing, le sue proprietà ed i suoi usi. Questo tipo di routing può essere impiegato nella risoluzione di range query su un'overlay network organizzata secondo triangolazione di Delaunay o diagramma di Voronoi. Come descritto in SWAM-V, è necessario eseguire una prima fase in cui viene applicato l'algoritmo di routing greedy ed adottare nella seconda fase l'algoritmo di compass routing per recuperare in maniera più efficiente tutti i match. In questo modo si minimizza il numero di messaggi spediti in rete, aspetto che non è garantito da SWAM-V in quanto sfrutta la tecnica del flooding.

Capitolo 4

VoRaQue: algoritmi e procolli

Nel capitolo precedente, oltre a fare una panoramica sui possibili usi dei diagrammi di Voronoi in ambito scientifico, l'obiettivo principale è stato quello di mostrare sia le funzionalità di VoroNet, che la sua predisposizione a gestire efficientemente sia **query semplici** che **query complesse**.

Gli autori di VoroNet [12] non hanno approfondito le problematiche relative a questo tipo di ricerche, e lo scopo della mia tesi è quello di studiare il problema per trovare una o più soluzioni, affinché il sistema sia in grado di risolvere le range query in maniera efficiente, sia dal punto di vista dello sfruttamento della banda, che del numero di nodi che elaborano la richiesta. Ciò, si potrebbe rivelare utile per superare i limiti relativi al supporto di ricerche complesse, riscontrati in letteratura dai vari sistemi P2P strutturati e non.

Fino ad oggi, le query semplici (o **query esatte**) sono state risolte con tecniche diverse, a partire dai sistemi P2P di prima generazione, come Napster [15], per passare a quelli di seconda generazione, come Gnutella [16, 17, 18], fino ad arrivare alle Distributed Hash Table [20, 21, 22] che si sono rilevate il miglior sistema per risolvere query di questo tipo. Il vantaggio di quest'ultima tecnica è che, data una chiave K, detto N il numero di nodi partecipanti, viene raggiunto il peer che memorizza l'informazione relativa alla chiave in $\log N$ hop ed ogni nodo mantiene una tabella di routing

di dimensione $\log N$.

Una query si definisce complessa se permette di specificare più attributi definendo per ognuno di essi un range di valori. Quest'ultime prendono il nome di **range query multi-attributo**.

In letteratura sono stati sviluppati molti sistemi che si preoccupano di risolvere questo tipo di query, alcuni dei quali sono stati presentati nel capitolo 2. La differenza sostanziale tra le varie soluzioni proposte si riscontra nella tipologia di architettura P2P che viene utilizzata. I sistemi P2P strutturati [1, 2, 6, 10, 11], sfruttano le DHT per i costi logaritmici e la garanzia di recuperare le informazioni all'interno della rete. In questo caso è stato ampiamente dimostrato che tali soluzioni non sono molto efficienti a causa degli elevati costi di manutenzione[1] di cui necessitano.

Diversamente, in sistemi P2P non strutturati [7], viene messa in evidenza la capacità di gestire efficientemente le range query utilizzando la strategia del Routing Index [9], ma con minor costi di gestione della rete. D'altro canto, le soluzioni che adottano architetture "meno strutturate" non garantiscono l'individuazione di tutti i possibili match per una query.

In questo capitolo viene descritto VoRaQue, un supporto P2P per la risoluzione di query complesse ispirato a VoroNet.

[1]Con il termine costi di manutenzione vogliamo indicare il numero di messaggi scambiati tra i nodi che sono necessari per mantenere consistente la topologia dell'overlay network.

4.1 Ipotesi di base

Ogni peer v che implementa un'istanza di VoRaQue mantiene una **visione** locale del sistema, ovvero è connesso ad un'insieme di vicini che vengono classificati in due sottoinsiemi:

- **Voronoi Neighbours VN(v)**: nodi le cui regioni condividono un lato della regione di v;
- **Close Neighbours CN(v)**: nodi che giacciono su una circonferenza di raggio d_{min} con centro le coordinate di v.

In particolare, quest'ultimi vengono utilizzati per garantire la consistenza topologica della rete, inoltre tali vicini vengono impiegati, insieme ai Voronoi Neighbours, quando viene eseguito l'algoritmo di routing greedy.

In VoRaQue non sono implementati i **Long Range Neighbours**, descritti nel capitolo 3, in quanto l'obiettivo di questa tesi è stato lo studio della risoluzione efficiente delle range query sulle reti di Voronoi.

Per tutto il resto del capitolo viene utilizzato il termine **messaggio** per descrivere i pacchetti che vengono scambiati tra i nodi, sia nella risoluzione di query esatte che di range query. Ogni messaggio che viene inviato sulla rete, è in realtà formato da un **header** (figura 4.1), che racchiude le informazioni comuni a tutti i pacchetti, e da un **payload**[2].Il formato dell'header del messaggio è il seguente:

[2]Termine inglese che significa carico utile. E' quella parte dati del pacchetto priva dell'intestazione [45].

ID	FUNC	TTL	$IPPORT_{SEND}$	$IPPORT_{REC}$	HOP

Figura 4.1: Header del pacchetto.

- **ID**: intero che identifica univocamente il messaggio.
- **FUNC**: descrive il tipo del messaggio (**ExactMSG**, **ReplyExactMSG**, **RangeMSG**, **ReplyRangeMSG**, **VoronoiNeighMSG**, **ReplyVoronoiNeighMSG** o **ExpiredTTL**).
- **TTL (Time-To-Live)**: quante volte il pacchetto può essere inoltrato dai nodi prima di essere eliminato dalla rete.
- **IPPORTsend**: se il payload è una richiesta, questo campo contiene l'indirizzo IP e la porta del nodo che ha sottomesso la query al sistema. Ciò significa che il peer che costruirà il messaggio di risposta invierà direttamente la reply al nodo richiedente. In questo modo si evita di utilizzare il **backward routing**[3], implementato nel protocollo di Gnutella [16, 17, 18]. Se il payload rappresenta una risposta, il mittente è proprio il nodo che ha costruito la reply.
- **IPPORTrec**: questo campo contiene l'indirizzo IP e la porta del nodo destinatario. La scelta di quest'ultimo dipende dal tipo di routing che viene utilizzato. Questa operazione sarà descritta in maniera dettagliata nel resto del capitolo.
- **HOP**: quanti nodi hanno già inoltrato il messaggio.

[3]La risposta positiva ad una query viene inoltrata a ritroso lungo lo stesso cammino seguito dalla query.

4.2 Query Esatte

Definiamo con il termine di query esatta una richiesta che permette di specificare un unico valore per ogni coordinata dello spazio degli attributi. In questo tipo di query, è necessario che sia assegnato un valore per ogni dimensione. In altre parole, nello spazio bi-dimensionale abbiamo che:

- $Q = \{R|x_Q = 128 \wedge y_Q = 300\}$ è una query esatta;
- $Q = \{R|x_Q = 128\}$ non è una query esatta, perché $x_Q = 128$ nel piano cartesiano rappresenta una retta.

Nell'istante in cui viene richiesta la risoluzione di una query esatta, viene applicato l'algoritmo di routing greedy dal nodo richiedente al nodo che contiene nella sua regione di Voronoi il punto definito dalla richiesta stessa. Come la tecnica del flooding utilizzata da Gnutella [16, 17, 18], il routing greedy di VoRaQue è limitato dal valore del TTL, personalizzabile dall'utente, che determina il numero massimo di nodi che possono essere attraversati. Durante la risoluzione di una query esatta, se un nodo azzera il valore del Time-To-Live, il protocollo mette a disposizione un pacchetto ad-hoc per informare il peer richiedente che la destinazione non è stata raggiunta.

4.2.1 Query Esatte: ExactMSG

Il payload del messaggio di tipo ExactMSG è il seguente:

Figura 4.2: Payload del messaggio ExactMSG.

- **GEOM** e **RANGEcriteria**: sono due flag che al momento non sono utilizzati ma sono presenti per uniformare il formato del pacchetto rispetto alle range query che invece ne fanno uso.

- **QUERYcoord**: sono le coordinate con cui viene rappresentata la query nello spazio.

4.2.2 Query Esatte: ReplyExactMSG

Il payload del messaggio di tipo ReplyExactMSG è il seguente:

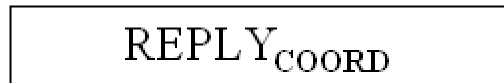


Figura 4.3: Payload del messaggio ReplyExactMSG.

- **REPLYcoord**: sono le coordinate del nodo che è match per la query. Se la query non ha soluzione, il campo sarà vuoto.

4.2.3 Query Esatte: ExpiredTTL

Il messaggio di tipo ExpiredTTL è caratterizzato da un payload vuoto.

4.2.4 Query Esatte: protocollo

Lo spazio degli attributi di VoRaQue viene mappato su un diagramma di Voronoi. In questo modo, ogni nodo viene rappresentato come un punto nello spazio bi-dimensionale. Supponiamo che un utente U richieda al sistema di risolvere la seguente richiesta:

$$Q = \{R | x_Q = 128 \wedge y_Q = 300\} \tag{4.1}$$

che rappresenta la query Q che ricerca la risorsa R di coordinate $(128, 300)$. Da un punto di vista geometrico, la query esatta viene rappresentata come un punto nello spazio degli attributi. Quest'ultimo può coincidere con le coordinate di un peer oppure può giacere all'interno di una regione di Voronoi. Di conseguenza, l'obiettivo di una query esatta è individuare il punto corrispondente nello spazio e restituire una risposta positiva se coincide esattamente con le coordinate di un peer, negativa altrimenti.

Il nodo N di coordinate (x_N, y_N) è un match per la query Q se vale la seguente condizione:

$$x_Q = x_N \wedge y_Q = y_N$$

In caso contrario occorre verificare se il punto definito da Q appartiene alla regione di N. Per eseguire questo test basta applicare la proprietà dei diagrammi di Voronoi, ovvero, un punto q fa parte della cella p_i se e solo se $dist(q, p_j) > dist(q, p_i)$ per ogni p_j con $i \neq j$. In altre parole deve valere la seguente condizione:

$$dist(Q, P) < dist(Q, V) \forall V \in VoronoiNeighbours(P)$$

Ciò significa che la distanza euclidea tra Q e P è più piccola della distanza che separa Q da qualunque vicino di P.

L'algoritmo che viene eseguito da ogni nodo che riceve una query esatta è derivato da queste osservazioni. Se un nodo P riceve una query esatta confronta le proprie coordinate per verificare se è un match per la query. In caso contrario confronta la distanza della query da se stesso e dai propri vicini. Se si individua come nodo a distanza minima significa che la query non ha soluzione.

Per esempio, osservando la figura 4.4 possiamo notare che la query Q non ha esito positivo, in quanto il punto che definisce giace nella regione di Voronoi del nodo P ma non combacia con quest'ultimo.

Iniziamo la descrizione del protocollo con la procedura per la risoluzione di una query esatta: per la trattazione considereremo l'overlay network formata dai nodi ri-

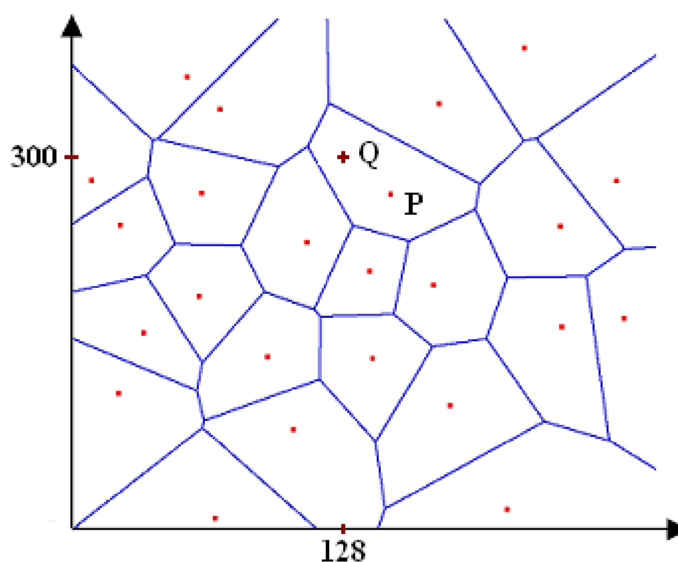


Figura 4.4: Rappresentazione geometrica della query esatta Q.

portati in figura 4.5. La procedura inizia con la sottomissione della richiesta 4.1 da parte di un utente connesso alla rete tramite il nodo N.

Prima di inoltrare il messaggio, VoRaQue verifica se le coordinate del nodo coincidono con il punto definito dalla query. In caso affermativo è il match per Q e la risposta viene restituita all'utente senza la necessità di inviare alcun pacchetto sulla rete. In caso negativo, si verifica se il punto giace nella regione di Voronoi di N. Se quest'ultimo test restituisce esito positivo, non vi è alcun match per la richiesta, ma otteniamo la sua risoluzione senza dover inoltrare il messaggio ad alcun vicino. In caso negativo, N non è in grado di restituire una risposta alla query Q[4], quindi l'applicazione costruisce un messaggio di tipo **ExactMSG** contenente le coordinate del punto da individuare nella rete.

Questo pacchetto viene inviato ad un nodo che viene selezionato secondo una

[4]Quando viene utilizzata la dicitura il "nodo è in grado di risolvere la query", significa che il punto definito dalla query o coincide con le coordinate del nodo oppure appartiene alla sua regione. In altre parole significa che almeno uno dei due test che il peer deve eseguire ogni volta riceve un messaggio di tipo ExactMSG, da esito positivo.

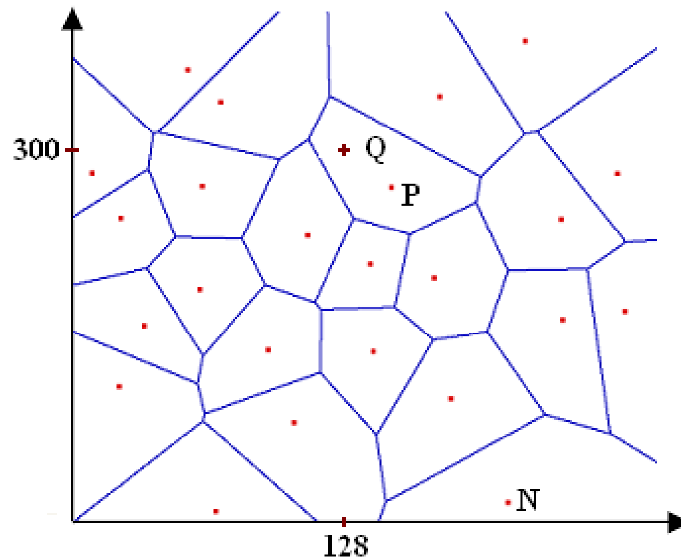


Figura 4.5: Situazione iniziale.

strategia greedy: *il destinatario è il peer le cui coordinate minimizzano la distanza euclidea con il punto definito dalla richiesta e appartiene all'insieme dei **VoronoiNeighbours o CloseNeighbours** del nodo che sta elaborando la query.*

Tutte le operazioni sopra descritte vengono applicate da qualsiasi peer che riceve un messaggio di tipo **ExactMSG** finché non viene raggiunto il nodo in grado di risolvere la query. Quest'ultimo (nodo P di figura 4.5) si preoccupa di restituire la risposta al nodo richiedente tramite il messaggio di tipo **ReplyExactMSG** inserendo le sue coordinate se è un match per Q, altrimenti non viene immesso nessun valore nel payload. Nella figura seguente abbiamo un esempio relativo alla risoluzione di una query Q.

In figura 4.6 le regioni di Voronoi in giallo rappresentano i VoronoiNeighbours, quelle in verde sono sia VoronoiNeighbours che CloseNeighbours. In figura 4.6a il nodo N invia la richiesta di ExactMSG al vicino N_1 poiché le coordinate di N_1 minimizzano la distanza euclidea con il punto Q. In figura 4.6b N_1 invia la query a N_2 in quanto risulta essere il nodo più vicino alla destinazione.

In figura 4.7a il nodo N_2 invia la richiesta al vicino P. Quest'ultimo verifica che il

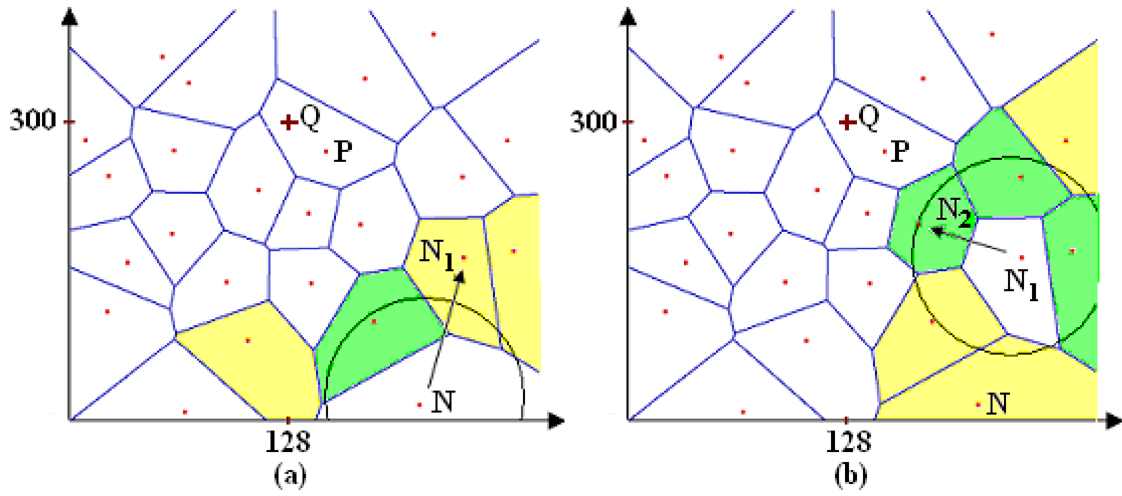


Figura 4.6: Risoluzione di query esatta.

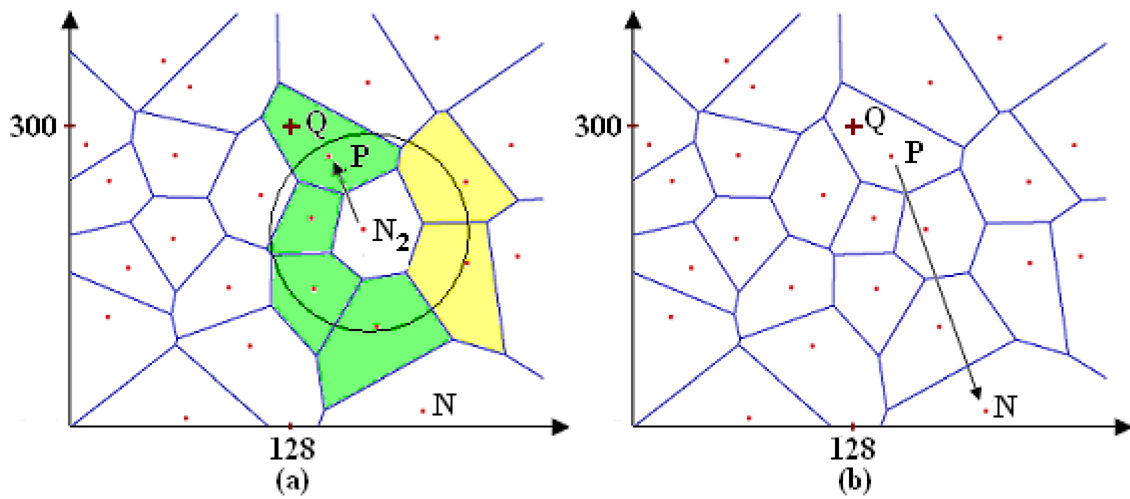


Figura 4.7: Simulazione di una risoluzione di query esatta.

punto Q non coincide con le sue coordinate ma appartiene alla sua regione di Voronoi. Quindi, costruisce un messaggio di ReplyExactMSG, contenente una risposta negativa, che viene inviato direttamente al richiedente (figura 4.7b).

Durante l'esecuzione del routing greedy, può accadere che il Time-To-Live raggiunga il valore zero senza che il pacchetto sia giunto al nodo in grado di risolvere la query. In questo caso, il peer che interrompe la propagazione della query, costruisce un messaggio di tipo **ExpiredTTL** e lo invia al nodo che ha generato la richiesta per informare l'utente che è necessario aumentare il valore del TTL per raggiungere la destinazione. Utilizzare un TTL personalizzabile permette al protocollo di essere più adattabile alle esigenze dell'utente.

4.3 Range Query

Definiamo con il termine range query una richiesta in cui almeno una coordinata dello spazio degli attributi è un **range di valori**. In altre parole, nello spazio bi-dimensionale abbiamo che:

- $Q = \{R \in \{R_1, ..., R_N\} | 128 \leq x_Q \leq 256 \wedge 100 \leq y_Q \leq 300\}$ è una range query;
- $Q = \{R \in \{R_1, ..., R_N\} | x_Q = 128\}$ è una range query che equivale a $Q = \{R \in \{R_1, ..., R_N\} | x_Q = 128 \wedge y_{min} \leq y_Q \leq y_{max}\}$ dove y_{min} e y_{max} rappresentano rispettivamente il valore minimo e massimo definito nello spazio degli attributi.

Il protocollo per la risoluzione delle range query ha l'obiettivo di recuperare in maniera efficiente tutti i match di una richiesta. Ciò viene realizzato utilizzando l'algoritmo di compass routing che è stato descritto nel capitolo precedente. Questa tecnica viene utilizzata per costruire uno spanning tree per eseguire un multicast efficiente sulla triangolazione di Delaunay che è il duale dei diagrammi di Voronoi. Ricordiamo che l'implementazione dell'algoritmo di compass routing richiede che un nodo P determini i suoi figli nello spanning tree. Quest'ultimi sono selezionati tra i suoi vicini di Voronoi che individuano anche gli archi della triangolazione di Delaunay. In realtà, piuttosto che capire se il vicino V è figlio di P nello spanning tree, è necessario verificare se P è padre di V. Come analizzato nel capitolo 3, per eseguire questo controllo occorre conoscere tutti i vicini di V e individuare se, tra questi, P è il vicino che forma con V l'angolo più piccolo con la radice dello spanning tree.

Ogni istanza di VoRaQue ha una visione del sistema che si limita ai vicini ai quali è connesso. Per questo motivo, quando il nodo P deve verificare se è il padre di V, deve inviare a quest'ultimo un messaggio per conoscere i suoi vicini di Voronoi. Il protocollo per la risoluzione delle range query permette di eseguire un caching sull'insieme dei Voronoi Neighbours di ogni vicino confinante con P, in modo da evitare di richiedere queste informazioni ogni volta che P riceve dalla rete una range query.

Inoltre, viene applicato un caching sui risultati restituiti dal compass routing. Queste tecniche vengono utilizzate per velocizzare l'elaborazione di una range query.

Una funzionalità messa a disposizione dal protocollo è la ricerca incrementale dei risultati. Come per la risoluzione delle query esatte, il TTL è personalizzabile dall'utente. Quando un peer azzera tale valore, si preoccupa di costruire il messaggio di risposta, con tutti i risultati fino ad ora raccolti, da inviare direttamente al nodo richiedente. Quest'ultimo estrae dal pacchetto i match e le informazioni che individuano il peer che ha interrotto la ricerca. In questo modo è possibile ottimizzare il protocollo perché per ottenere ulteriori risultati possono essere contattati direttamente i nodi che hanno arrestato la ricerca.

4.3.1 Range Query: RangeMSG

Il payload del messaggio di tipo RangeMSG è il seguente:

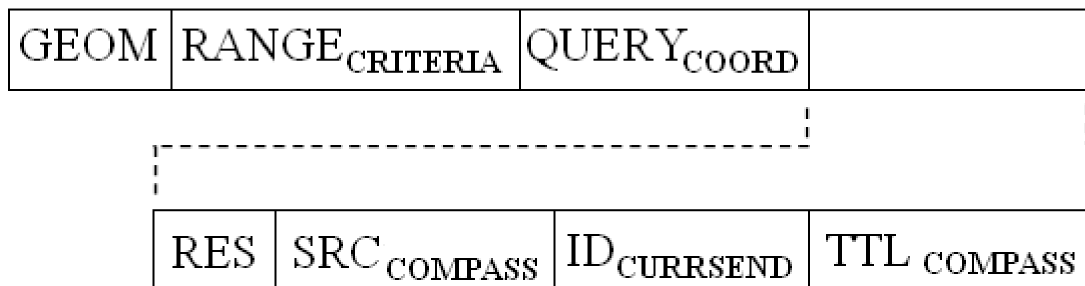


Figura 4.8: Payload del pacchetto RangeMSG.

- **GEOM**: specifica se si sta applicando il routing greedy o il compass.
- **RANGEcriteria**: è un flag che contiene informazioni varie, ad esempio se si desidera raggiungere il centro dell'area di interesse della query prima di iniziare la fase di compass routing.
- **QUERYcoord**: sono le coordinate con cui viene rappresentata la query nello spazio.

- **RES**: è una lista che contiene le informazioni di tutti i nodi attraversati dal messaggio. Ogni elemento è caratterizzato da un indirizzo ip, una porta e le coordinate del peer nello spazio.

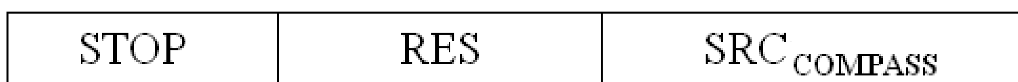
- **SRCcompass**: sono le coordinate del nodo che rappresenta la radice dello spanning tree.

- **IDcurrsend**: questo campo indica l'identificatore associato al nodo che ha inoltrato il messaggio durante la fase di compass routing. Quando un peer elabora una richiesta, recupera la lista dei vicini di Voronoi la cui regione interseca l'area di interesse della query. Con quest'informazione può cancellare da tale lista il nodo che gli ha inoltrato la richiesta.

- **TTLcompass**: rappresenta il valore del Time-To-Live selezionato dall'utente. Il nodo che sarà la radice dello spanning tree inizializza il campo TTL dell'header con questo valore.

4.3.2 Range Query: ReplyRangeMSG

Il payload del messaggio di tipo ReplyRangeMSG è il seguente:



STOP	RES	$SRC_{COMPASS}$

Figura 4.9: Payload del pacchetto ReplyRangeMSG.

- **STOP**: indica la possibilità di riprendere la ricerca.

- **RES**: è una lista che contiene le informazioni di tutti i nodi attraversati dal messaggio. Ogni entrata è caratterizzata da un indirizzo ip, una porta e le coordinate del peer nello spazio.

- **SRCcompass**: sono le coordinate del nodo che è la radice dello spanning tree. Questo campo è utile se la ricerca può proseguire.

4.3.3 Range Query: VoronoiNeighMSG

Il payload del messaggio di tipo VoronoiNeighMSG è il seguente:

Figura 4.10: Payload del pacchetto VoronoiNeighMSG.

- **IDcompass**: è l'identificatore associato al compass routing che il peer sta eseguendo.

4.3.4 Range Query: ReplyVoronoiNeighMSG

Il payload del messaggio di tipo ReplyVoronoiNeighMSG è il seguente:

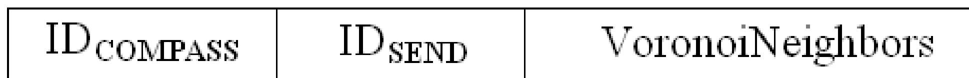


Figura 4.11: Payload del pacchetto ReplyVoronoiNeighMSG.

- **IDcompass**: è l'identificatore associato al compass routing. E' lo stesso valore che è presente nel messaggio per la richiesta dei vicini di Voronoi.

- **IDsend**: riporta l'identificatore del nodo che ha costruito il messaggio di reply.

- **VoronoiNeighbours**: contiene la lista dei vicini di Voronoi.

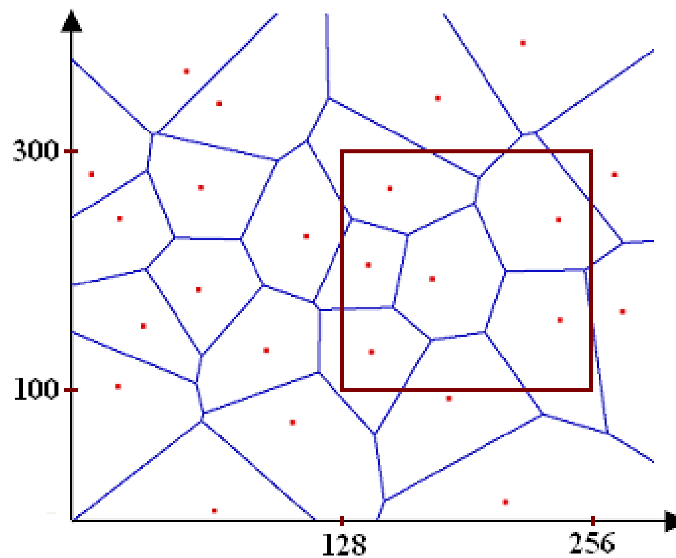


Figura 4.12: Rappresentazione geometrica della range query Q.

4.3.5 Range Query: protocollo

Supponiamo che un utente U richieda al sistema di risolvere la seguente richiesta:

$$Q = \{R \in \{R_1, ..., R_N\} | 128 \leq x_Q \leq 256 \wedge 100 \leq y_Q \leq 300\} \tag{4.2}$$

Il cui significato nello spazio bi-dimensionale è indicato in figura 4.12. In questo caso la query non rappresenta un punto, ma definisce un'area che chiameremo *Area di Interesse (AOI - Area Of Interest)*.

L'obiettivo del routing è quello di recuperare tutti i nodi la cui regione di Voronoi interseca per almeno un punto l'Area di Interesse della query. La fase di ricerca è così organizzata: applicazione dell'algoritmo di routing *Greedy* per raggiungere l'*AOI* e successivamente esecuzione del *Compass Routing* [47, 48], che consente di costruire un albero di multicast che permette di recuperare i nodi che appartengono all'area di interesse della query minimizzando il numero di messaggi spediti.

Il risultato è ben visibile in figura 4.13.

Il protocollo per la risoluzione delle range query sfrutta due tecniche di routing,

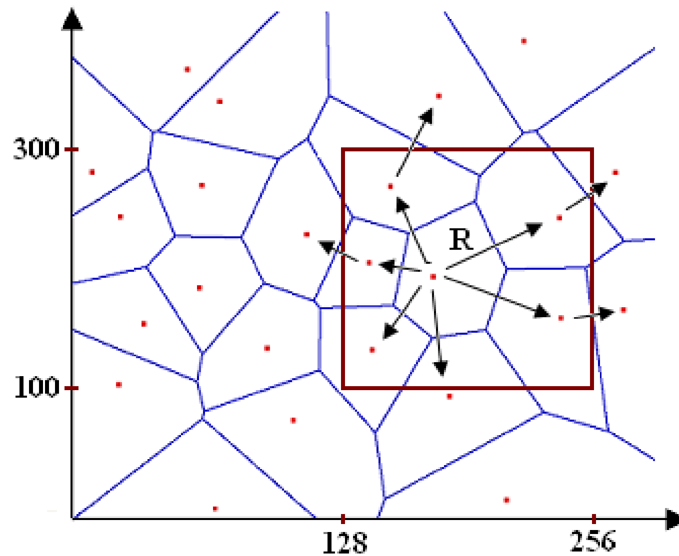


Figura 4.13: Costruzione dello spanning tree con radice R.

come conseguenza abbiamo che il protocollo permette di associare un Time-To-Live diverso per ogni algoritmo di instradamento. Occorre fare una distinzione:

1. se il nodo che sottomette la query al sistema coincide con la radice dello spanning tree, il campo **TTL** dell'header viene inizializzato con il valore del Time-To-Live selezionato per l'algoritmo di compass routing.

2. Se il nodo che sottomette la query non coincide con la radice dello spanning tree, il campo **TTL** dell'header viene inizializzato con il valore del Time-To-Live selezionato per il routing greedy. In questo caso viene eseguito tale algoritmo di instradamento per raggiungere il nodo radice dello spanning tree. Quest'ultimo, quando riceve il messaggio di tipo RangeMSG, verifica che può iniziare la fase di compass routing, ma prima deve modificare il campo TTL dell'header con il valore del Time-To-Live scelto per il secondo algoritmo di instradamento. Quest'informazione è reperibile dal payload di RangeMSG (campo **TTLcompass**) che era stato inizializzato dal nodo che aveva generato la query.

Nel capitolo precedente, abbiamo preso in esame SWAM-V che affronta il proble-

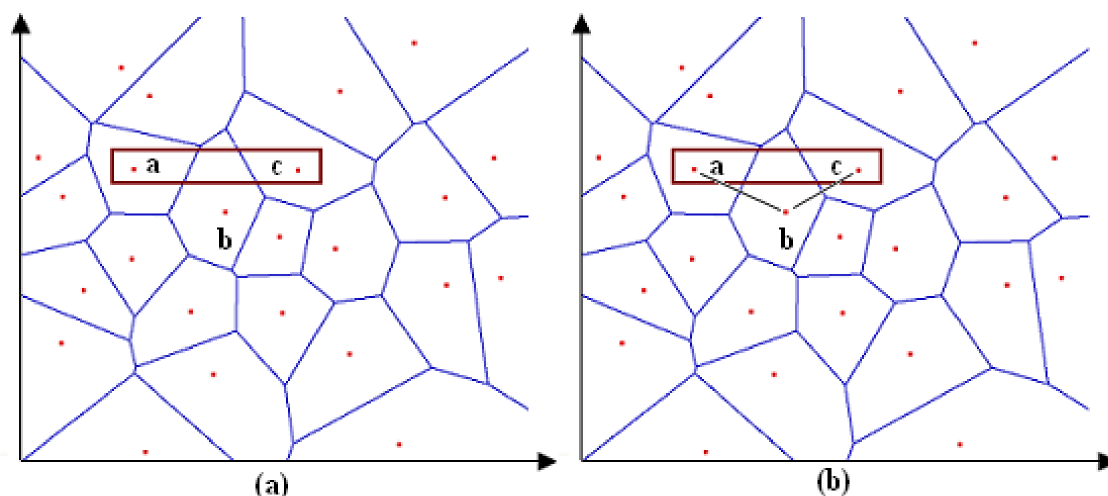


Figura 4.14: Risoluzione di una range query.

ma relativo alla risoluzione di range query sulle reti di Voronoi, ma in quel caso la tecnica adottata per recuperare i potenziali match nell'area di interesse descritta dalla query avviene tramite l'uso della tecnica di *flooding* [49] che si rivela efficace, ma non efficiente dal punto di vista dei messaggi spediti sulla rete.

L'obiettivo del routing è quello di recuperare tutti i nodi la cui regione di Voronoi interseca ***per almeno un punto*** *l'AOI della query.* Questa scelta è stata fatta per evitare la perdita di potenziali match in prossimità dei bordi dell'area di interesse. Osserviamo la figura 4.14.

Possiamo vedere che, se lo spanning tree dell'area di interesse della query di figura 4.14a fosse ottenuto applicando il compass routing considerando soltanto i nodi le cui coordinate soddisfano la condizione della query, si potrebbe verificare la perdita di potenziali match infatti il nodo a non può raggiungere il nodo c perchè dovrebbe passare da b, ma le coordinate di quest'ultimo punto giacciono al di fuori dell'area di interesse.

In realtà lo spanning tree viene costruito sui nodi la cui regione di Voronoi interseca per almeno un punto l'area di interesse della query, in questo modo, se a fosse la

radice dell'albero distribuito raggiungerebbe il nodo c passando per b, che a sua volta è un match. Da notare che vale anche il caso speculare, ovvero, se c fosse la radice dell'albero distribuito raggiungerebbe il nodo a passando per b (figura 4.14b).

Per questo motivo, i controlli che un peer deve effettuare per verificare se almeno un punto della sua regione di Voronoi interseca quella descritta da una range query sono diversi rispetto a quelli eseguiti durante la risoluzione di una richiesta semplice. Tali controlli sono elencati di seguito.

E' necessario che sia verificato almeno uno di essi affinché il nodo sia considerato un risultato per la richiesta. Consideriamo il peer P di coordinate (x_P, y_P) e la query $Q = \{R \in \{R_1, ..., R_N\} | x_{MIN} \leq x_Q \leq x_{MAX} \wedge y_{MIN} \leq y_Q \leq y_{MAX}\}$. P è un match per Q se:

1. le coordinate del nodo appartengono all'area di interesse definita dalla richiesta, ovvero deve valere la condizione: $x_{MIN} \leq x_P \leq x_{MAX} \wedge y_{MIN} \leq y_P \leq y_{MAX}$ (figura 4.15a).

2. Esiste almeno un vertice della regione di Voronoi del peer P appartenente all'area di interesse della query, ovvero, se $\exists V \in VertexRegion(P)$ tale che $x_{MIN} \leq x_V \leq x_{MAX} \wedge y_{MIN} \leq y_V \leq y_{MAX}$, dove VertexRegion rappresenta l'insieme dei vertici che delimitano la regione di Voronoi di P(figura 4.15b).

3. Esiste almeno un lato della regione di P che interseca almeno un lato dell'area di interesse della query (figura 4.16a).

4. Se nessuna delle condizioni precedenti si è verificata, per ogni vertice dell'area di interesse, si calcola la distanza euclidea che separa il punto P da tale vertice. In questo modo possiamo recuperare il punto più lontano dell'AOI. Se quest'ultimo giace all'interno della regione di Voronoi di P, allora l'area definita dalla query è completamente contenuta nella regione di P (figura 4.16b).

Un nodo è un match per una range query se almeno uno dei controlli descritti sopra ha esito positivo.

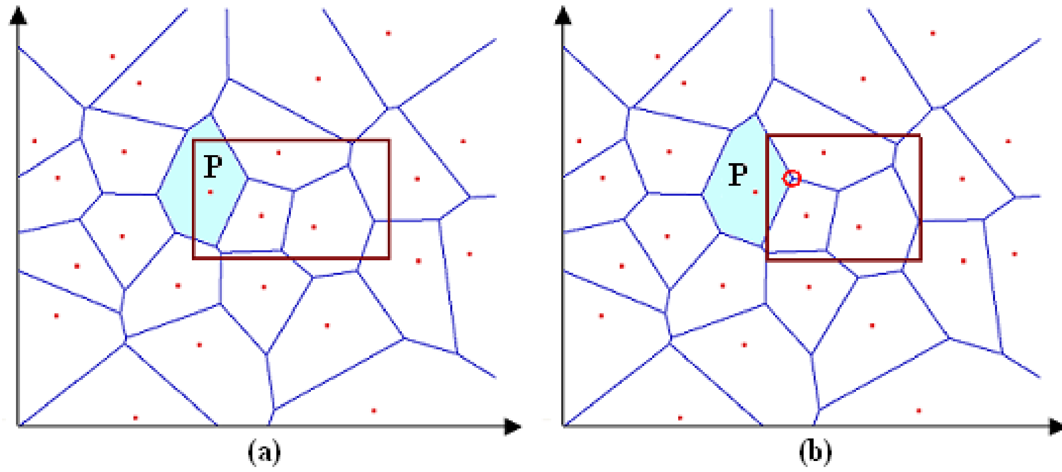


Figura 4.15: Controlli per verificare se un nodo è un match. Il rettangolo rappresenta l'area di interesse della query.

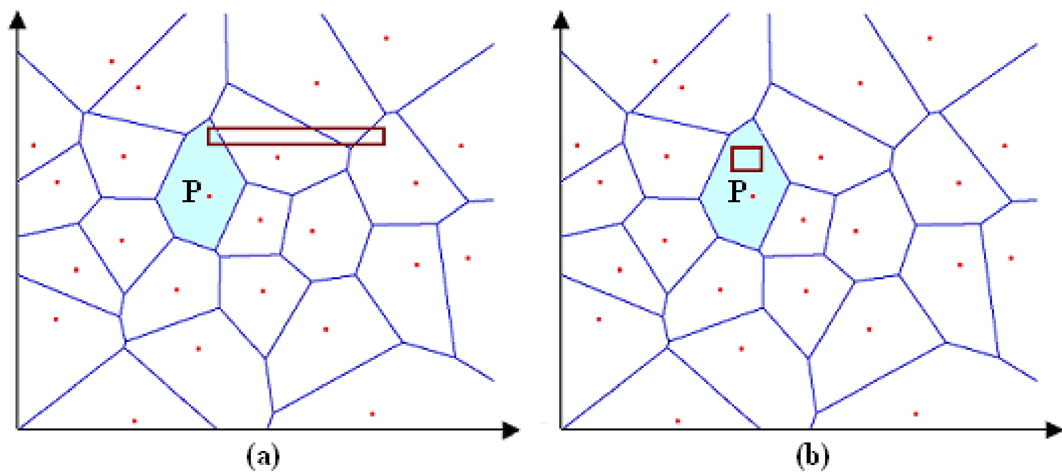


Figura 4.16: Controlli per verificare se un nodo è un match. Il rettangolo rappresenta Q.

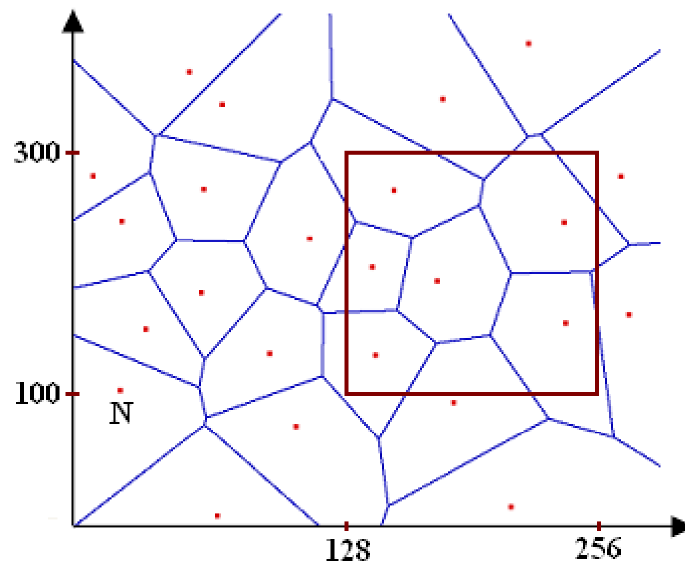


Figura 4.17: Situazione iniziale.

Infine, è importante indicare un'ulteriore differenza tra il protocollo per la risoluzione di query esatte e quello per la risoluzione di range query. Per le prime, non appena viene individuato il peer in grado di risolvere la richiesta, viene costruito il messaggio di reply. Invece, per le seconde la risposta viene spedita quando il Time-To-Live del compass raggiunge il valore zero o quando non vi sono ulteriori nodi nell'area di interesse che possono essere raggiunti con un hop. Questo verrà spiegato in maniera più dettagliata nei prossimi paragrafi.

La procedura inizia con la sottomissione della richiesta 4.2 da parte di un utente connesso alla rete tramite il nodo N (figura 4.17).

La prima fase del protocollo riguarda il raggiungimento di un nodo che appartenga all'AOI individuata dalla query. Per questa ragione, prima di inoltrare il messaggio, l'applicativo verifica se la regione di Voronoi di N interseca per almeno un punto l'area di interesse della query Q. In caso negativo, viene costruito il messaggio di tipo **RangeMSG** e si applica il routing greedy per raggiungere l'AOI. Da notare che il nodo ricevente viene selezionato secondo questo criterio: *il destinatario è il peer*

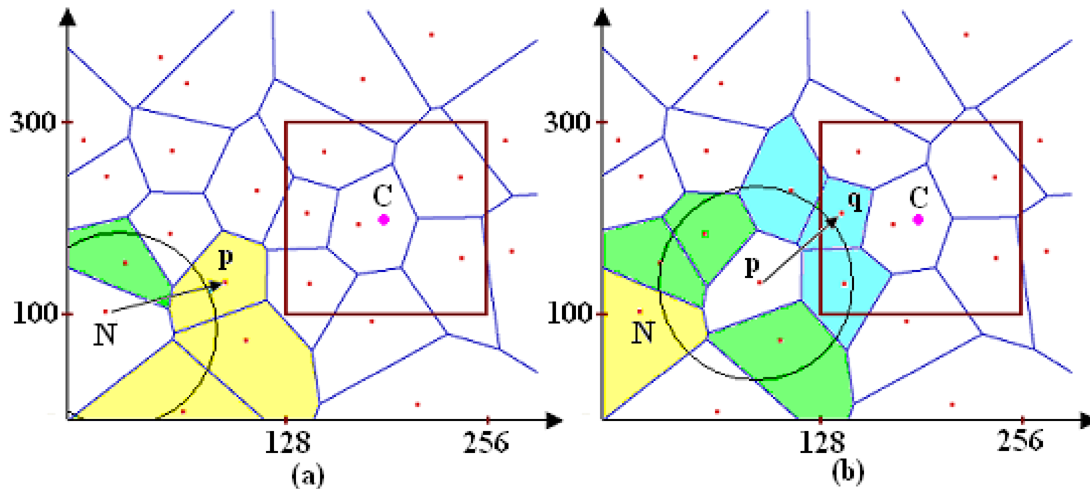


Figura 4.18: Applicazione dell'algoritmo di routing greedy per raggiungere l'area di interesse.

le cui coordinate minimizzano la distanza euclidea con il punto centrale dell'area di interesse ed appartiene all'insieme dei ***VoronoiNeighbours o CloseNeighbours*** *del nodo che sta elaborando la query.*

In caso affermativo, ovvero se la regione di N interseca per almeno un punto l'AOI, si può passare subito alla seconda fase, che prevede l'esecuzione del compass routing per recuperare tutti i possibili match, oppure si può decidere di continuare ad applicare il routing greedy per raggiungere comunque il centro dell'area di interesse. Questa scelta viene lasciata all'utente. Se la query fosse completamente contenuta nella regione di N, quest'ultima verrebbe subito risolta senza dover inviare alcun messaggio sulla rete.

Come possiamo vedere in figura 4.17, il nodo N non interseca l'area di interesse della richiesta. In questo caso, viene applicato l'algoritmo di routing greedy per raggiungere il primo match come mostrato nella figura 4.18.

In figura 4.18 le regioni colorate in giallo rappresentano i VoronoiNeighbours mentre quelle in verde sono sia Voronoi che CloseNeighbours. Le regioni colorate in celeste rappresentano il generico vicino che è match per la query, mentre C è il centro

dell'AOI. In figura 4.18a il nodo N ha quattro vicini, ma tra questi seleziona p e gli invia la richiesta di RangeMSG. Quest'ultimo inoltra la query al peer q (figura 4.18b).

Finché viene applicato l'algoritmo di routing greedy, il protocollo per la risoluzione delle range query mantiene lo stesso funzionamento di quello descritto all'inizio di questo capitolo per le query esatte.

Il protocollo per la risoluzione di una range query può essere eseguito in due diverse modalità. Nella prima la generazione dello spanning tree avviene non appena si raggiunge un nodo che appartiene all'area di interesse della query. Nella seconda modalità l'albero viene creato a partire dal nodo centrale dell'AOI.

Supponiamo che il protocollo venga eseguito considerando la prima modalità. In questo caso il nodo q diventa la radice dello spanning tree ed inizia l'esecuzione del compass routing.

Per prima cosa, a partire dall'insieme dei VoronoiNeighbours, q recupera i vicini la cui regione interseca per almeno un punto l'area di interesse della query. Fatto questo, inoltra ad ognuno di essi il messaggio di tipo **RangeMSG**, inserendo il suo ip, la sua porta e le sue coordinate nel campo **RES**, che rappresenta i match fino ad ora incontrati (figura 4.19). Inoltre, q inserisce le sue coordinate nel campo **coordSRCCompass**, in modo che il generico peer che riceve la richiesta possa recuperare facilmente la radice dello spanning tree per eseguire il compass routing, che ricordiamo, include il calcolo degli angoli formati con la radice. E' importante sottolineare che il nodo q, non appena ha inoltrato il messaggio ai vicini, rimane in attesa di ricevere **nuove** richieste dalla rete.

A questo punto, prendiamo in esame la computazione eseguita dal generico peer P quando riceve una richiesta di tipo **RangeMSG** in fase di compass routing. Ricordiamo che l'implementazione dell'algoritmo di compass routing richiede che P determini i suoi figli nello spanning tree. Quest'ultimi sono selezionati a partire dai suoi vicini di Voronoi che individuano anche gli archi della triangolazione di Delaunay. In realtà piuttosto che capire se il vicino V è figlio di P nello spanning tree è necessario veri-

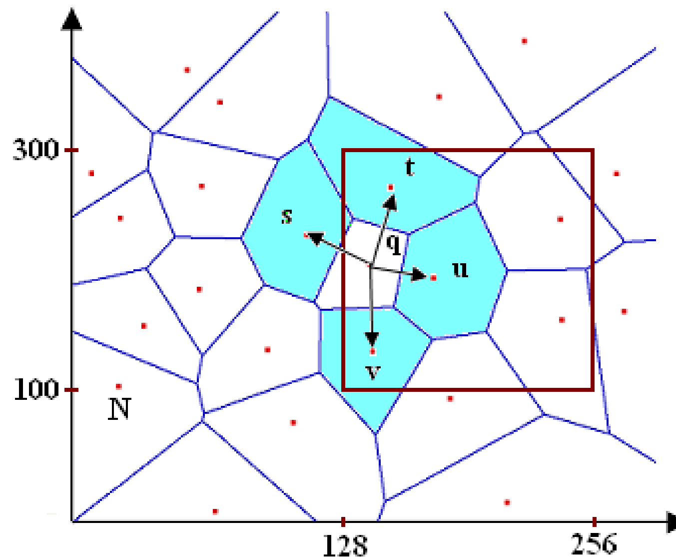


Figura 4.19: Prima fase del Compass Routing. Sono colorate in celeste le regioni di Voronoi dei vicini di q che intersecano l'area di interesse definita dalla query.

ficare se *P è padre di V*. Come analizzato nel capitolo 3, per eseguire questo controllo occorre conoscere tutti i vicini di V e individuare se tra questi P è il vicino che forma con V l'angolo più piccolo con la radice dello spanning tree.

Consideriamo il nodo u di figura 4.19. Per prima cosa, a partire dall'insieme dei VoronoiNeighbours, u recupera i vicini la cui regione interseca per almeno un punto l'area di interesse della query. Fatto questo, invia ad ognuno di essi un messaggio di tipo **VoronoiNeighMSG** per ottenere i loro vicini di Voronoi (figura 4.20a). Tutto ciò viene fatto per poter calcolare l'angolo ed applicare le decisioni locali definite dal Compass Routing.

Uno alla volta, i vicini interpellati risponderanno con un messaggio di tipo **ReplyVoronoiNeighMSG** includendo nel campo **voronoiNeighbours** le informazioni relative ai vicini e in particolar modo le loro coordinate.

In figura 4.20a sono colorate in celeste le regioni di Voronoi dei vicini di u che intersecano l'area di interesse definita dalla query. In figura 4.20b, il nodo w invia un

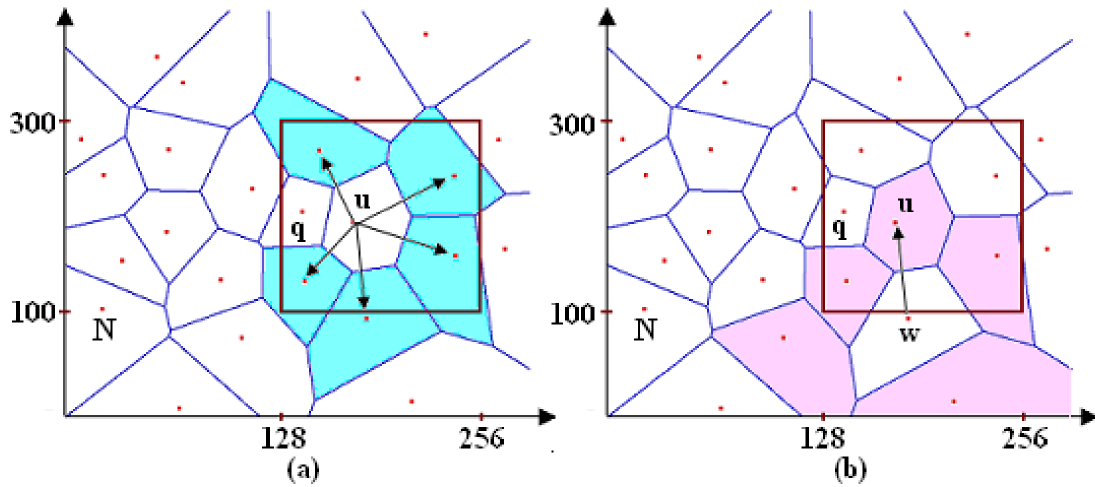


Figura 4.20: Richiesta dei vicini di Voronoi ai vicini di Voronoi.

messaggio di tipo **ReplyVoronoiNeighMSG** ad u includendo i suoi vicini di Voronoi (regioni colorate in viola). Da notare che vengono inviati anche i vicini di Voronoi di w che non intersecano l'area di interesse.

Supponiamo che u riceva come prima risposta la reply proveniente dal nodo w (figura 4.20b). u esegue la seguente procedura:

1. estrae le coordinate della sorgente del Compass q dal campo coordSRCCompass del messaggio RangeMSG e calcola il vettore $\overrightarrow{w-q}$.

2. $\forall n \in VoronoiNeighbours(w)$, calcola il vettore $\overrightarrow{w-n}$ e ottiene l'angolo compreso tra tale vettore e quello determinato al punto 1 sfruttando la seguente formula: $\theta = \arccos \frac{\overrightarrow{a} - \overrightarrow{b}}{|a| \cdot |b|}$, dove a e b sono i due vettori appena calcolati.

3. Recupera il nodo che forma l'angolo più piccolo con il vettore $\overrightarrow{w-q}$ e verifica se questo peer coincide con se stesso. In caso affermativo, u rappresenta il padre per w e gli spedisce il messaggio di **RangeMSG**. Altrimenti, il nodo non esegue nessuna operazione.

Nel caso specifico, il nodo che descrive l'angolo più piccolo con il vettore $\overrightarrow{w-q}$ è v (figura 4.21).

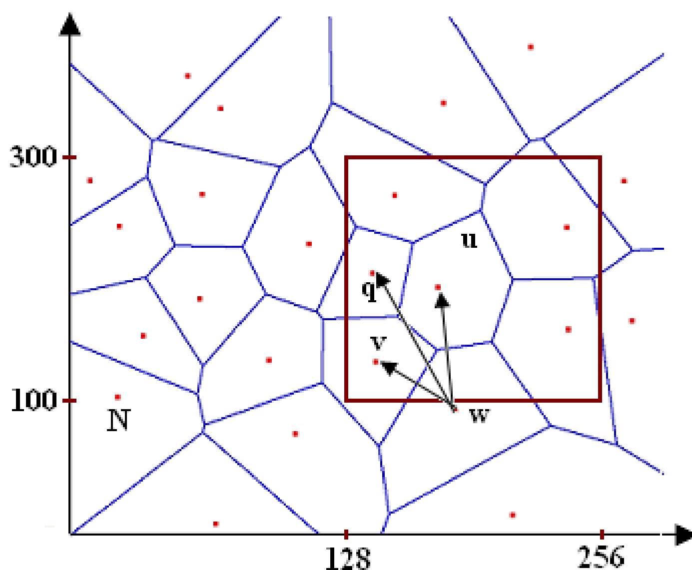


Figura 4.21: L'angolo minore è compreso tra i nodi v, w e q.

Supponiamo che u riceva un messaggio di ReplyVoronoiNeighMSG dal vicino z (figura 4.22a). Anche in questo caso, u esegue la procedura sopra descritta, ma stavolta il vicino che forma l'angolo più piccolo con il vettore $\overrightarrow{z-q}$ è proprio u (figura 4.22b). Quindi, quest'ultimo è il padre di z e può inoltrargli il messaggio di RangeMSG.

In conclusione, se ogni nodo che riceve un messaggio di tipo RangeMSG esegue le computazioni sopra descritte, viene costruito uno spanning tree che permette di eseguire un multicast efficiente.

Nella figura 4.23a viene mostrato l'albero che è stato costruito al termine del compass routing.

Se la query fosse stata eseguita in modalità da *raggiungere il centro di AOI*, sarebbe stato necessario eseguire un altro passo del routing greedy per raggiungere tale punto, che di fatto appartiene alla regione di Voronoi di u. Lo spanning tree risultante viene mostrato in figura 4.23b.

In figura 4.23a, la sorgente è q poiché l'albero viene costruito a partire dal primo match individuato. In figura 4.23b, la sorgente è u perché è stato richiesto di

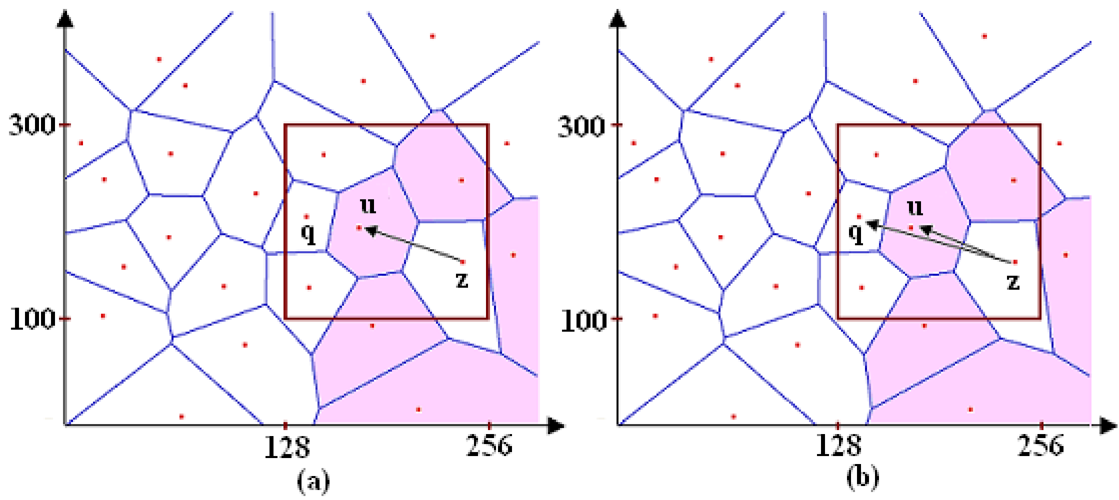


Figura 4.22: Ricezione di un messaggio di ReplyVoronoiNeighMSG con individuazione dell'angolo minore. Le regioni in viola rappresentano i vicini del nodo z.

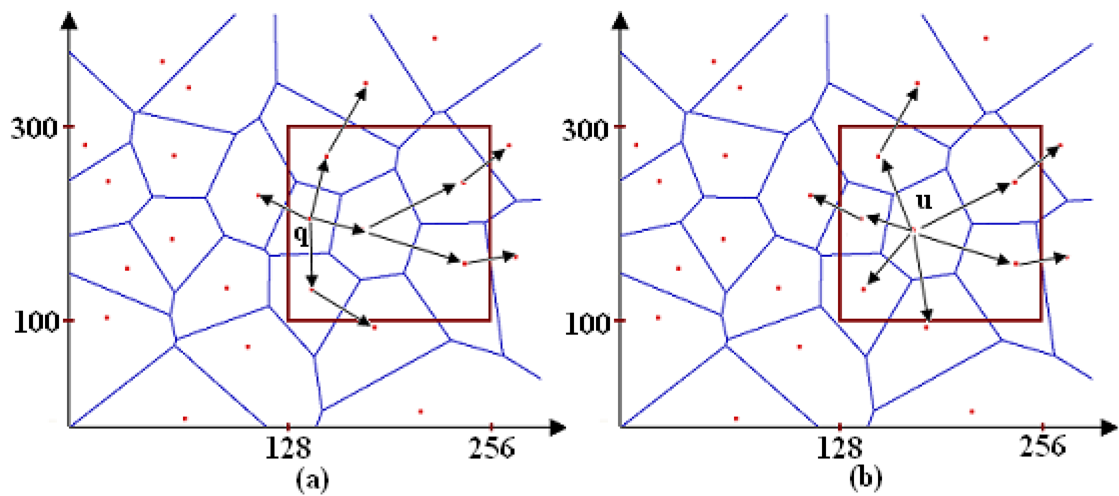


Figura 4.23: Spanning tree per la query $Q = \{R \in \{R_1, ..., R_N\} | 128 \leq x_Q \leq 256 \wedge 100 \leq y_Q \leq 300\}$**.**

raggiungere il centro dell'area di interesse della query.

4.3.6 Range Query: Ricerca incrementale dei match

Come nel protocollo per la risoluzione delle query esatte, anche per le range query l'utente può impostare il valore del TTL ed incrementarlo successivamente, in modo da individuare ulteriori match. Per ottimizzare il protocollo è opportuno determinare i peer da cui proseguire la ricerca e contattarli direttamente. A questo scopo ogni nodo che azzera il TTL di un messaggio di richiesta, costruisce una reply e la invia direttamente al peer che ha generato la query. Quest'ultimo, tra tutte le risposte ricevute, recupera le informazioni di quei nodi dai quali è possibile riprendere la ricerca. Questo procedimento viene implementato tramite lo **stop bit** che è un'informazione booleana che risiede nel pacchetto di **ReplyRangeMSG**. Lo stop bit viene utilizzato per informare il nodo che ha generato la richiesta sulla possibilità di proseguire nella ricerca dei risultati qualora quest'ultimi non fossero sufficienti. In sostanza, lo stop bit può assumere due valori:

1. **true**: la ricerca non può proseguire in questa direzione;
2. **false**: la ricerca può proseguire. In questo caso sarà possibile estrarre dal messaggio l'indirizzo ip e la porta del mittente per poter inviare successivamente la query direttamente a questo.

Supponiamo che un utente U richieda al sistema di risolvere la richiesta sottostante:

$$Q = \{R \in \{R_1, ..., R_N\} | x_Q \leq 256 \wedge y_Q \leq 400\} \tag{4.3}$$

Supponiamo che U sia connesso alla rete tramite il nodo N (figura 4.24a). Partendo dal presupposto che, come il TTL del routing greedy, il Time-To-Live del Compass sia personalizzabile da parte dell'utente, supponiamo che N sia al centro dell'area di interesse della query e che **TTLcompass** sia uguale a 1. Osserviamo la situazione di figura 4.24.

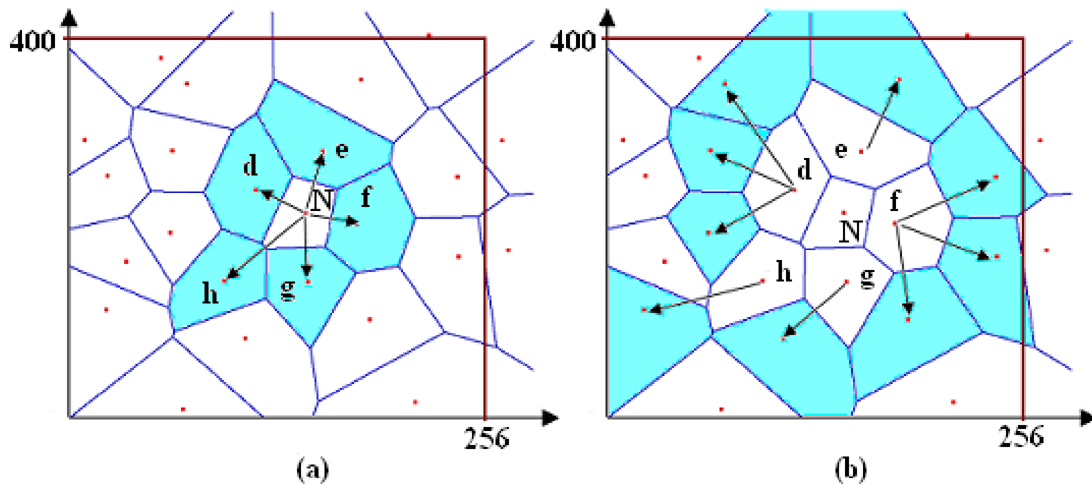


Figura 4.24: Simulazione Compass Routing.

In figura 4.24a, abbiamo che N, essendo la radice dello spanning tree, invia la richiesta di RangeMSG a tutti i suoi vicini (regioni colorate in celeste) in quanto anch'essi sono match per la query 4.3. Prima di inoltrare la richiesta, N inizializza il valore del campo TTL dell'header del messaggio con il valore di TTLcompass (uguale a 1). A loro volta, i peer d, e, f, g, h elaborano il messaggio, decrementano il valore del Time-To-Live, che adesso è uguale a zero, inseriscono le loro informazione nel campo **RES** ed inviano il pacchetto RangeMSG ai vicini secondo i criteri del Compass Routing (figura 4.24b).

In figura 4.25 abbiamo che i peer che ricevono la richiesta inviata da d, e, f, g, h (regioni di Voronoi colorate in verde) trovano il valore del TTL uguale a zero e si limitano a costruire il pacchetto di ReplyRangeMSG da restituire direttamente ad N, senza inserire le loro informazioni, ma settando il campo **STOP** della reply a false.

A questo punto, N riceve una lista di risultati per la query 4.3 che comprende i peer d, e, f, g, h. Se non sono sufficienti, l'utente può decidere di proseguire la ricerca ed in questo caso il messaggio di **RangeMSG** viene direttamente inviato a tutti i peer che hanno spedito il messaggio di **ReplyRangeMSG** con lo stop bit a false (figura 4.26a). Nella richiesta viene settato un nuovo valore per il TTL, ancora una volta

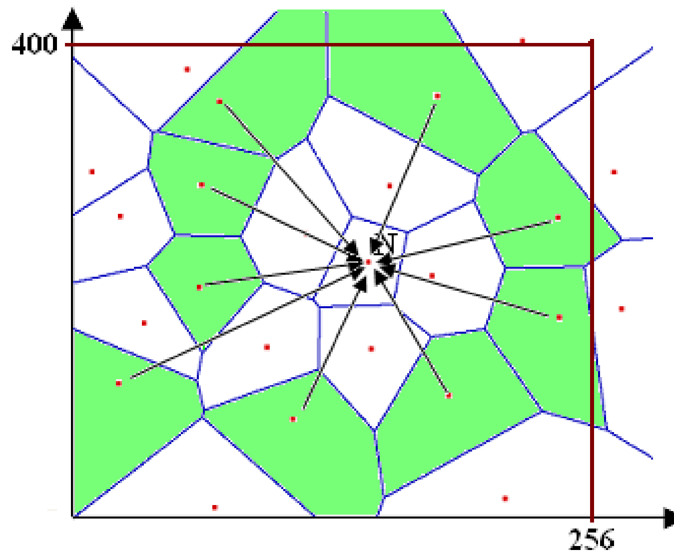


Figura 4.25: Simulazione Compass Routing.

scelto dall'utente.

In figura 4.26a N riprende la ricerca dai nodi che gli hanno risposto. In figura 4.26b, il nodo con la regione colorata di verde spedisce un messaggio di ReplyRange-MSG, inserendo le sue informazioni nel campo **RES**, ma impostando lo stop bit a true poiché non ha vicini a cui spedire. Tutti gli altri elaborano la richiesta e la inoltrano ai vicini (zone colorate in celeste).

In figura 4.27 viene descritta l'ultima fase. Tutti i peer evidenziati in verde spediscono ad N il messaggio di ReplyRangeMSG, dopo aver inserito le loro informazioni e dopo aver impostato lo stop bit a true. Adesso non è più possibile proseguire la ricerca.

In generale, il compass routing per una richiesta termina quando tutti i nodi che stanno elaborando la query non hanno vicini a cui spedire.

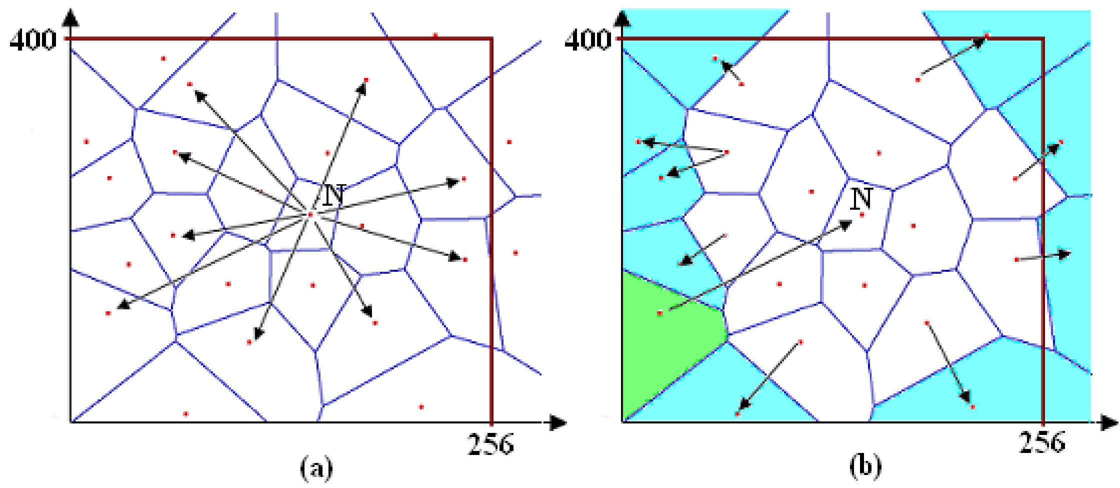


Figura 4.26: Simulazione Compass Routing.

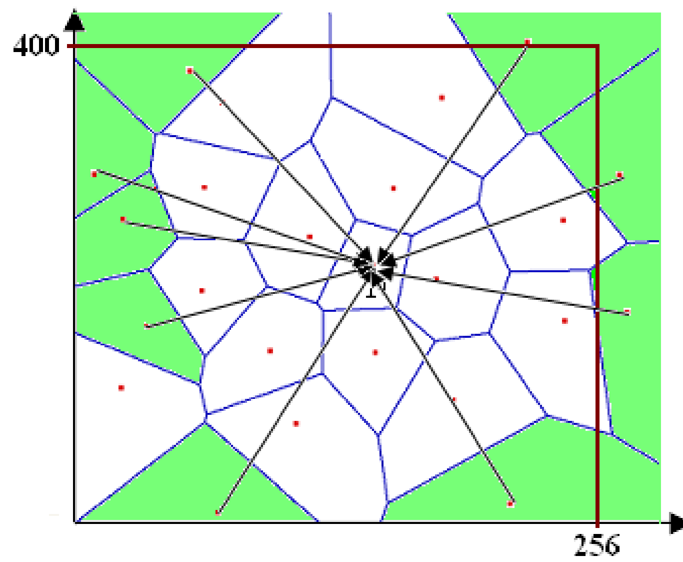


Figura 4.27: Simulazione Compass Routing.

4.3.7 Range Query: Caching

Il protocollo utilizza tecniche di caching per ottimizzare l'elaborazione di VoRaQue durante la risoluzione di una range query. Vengono impiegate due tipologie di cache:

- la prima viene utilizzata per memorizzare i vicini di Voronoi dei peer confinanti con un nodo P. In questo modo si evita di spedire le richieste di **VoronoiNeighMSG** ogni volta che P riceve una nuova range query dalla rete. Le informazioni contenute in questa cache hanno associato un valore che indica la "freschezza" del dato e l'intervallo di validità dipende dal **churning**[54] dell'overlay network. In altre parole, se la rete è soggetta ad un elevato numero di operazioni di inserimento e cancellazione dei nodi, i dati in cache possono essere validi per un breve intervallo di tempo a causa dei frequenti cambiamenti. Diversamente, se l'overlay network risulta più "statica", le informazioni memorizzate possono valere per periodi più lunghi.

- La seconda cache viene utilizzata per memorizzare le informazioni relative all'*i-simo compass routing* che VoRaQue sta elaborando. In altre parole, quando un peer P riceve una richiesta di range query dalla rete, la prima cosa che deve fare è stabilire la relazione padre-figlio. Se la cache definita al punto 1 non contiene nessun dato, P deve inviare le richieste per ottenere i vicini di Voronoi dei nodi confinanti per eseguire i calcoli descritti nel compass routing. Spediti tali pacchetti, P **non rimane in attesa di ricevere le relative risposte dalla rete, perchè lo stato dell'elaborazione dell'i-esimo compass routing viene salvato in cache**. In questo modo, P può elaborare qualsiasi messaggio indipendentemente dall'ordine in cui vengono ricevuti.

  Ogni volta che P riceve un pacchetto di tipo **ReplyVoronoiNeighMSG**, estrae dal campo **IDcompass** del payload, l'identificatore associato all'i-esimo compass routing, grazie a questa informazione recupera dalla cache i dati corrispon-

denti e ripristina lo stato relativo all'elaborazione dell'i-esima range query e può procedere nella sua risoluzione.

L'i-esimo compass routing termina quando P ha eseguito il calcolo dell'angolo su tutti i suoi vicini confinanti. Da notare che, conclusa l'elaborazione di una range query, tutte le informazioni associate rimangono in cache per un intervallo di tempo t con lo scopo di velocizzare l'elaborazione di VoRaQue nel caso si possa ricevere nuovamente la medesima query[5].

[5]Due range query sono equivalenti se coincidono le coordinate sia dell'area di interesse che della radice dello spanning tree.

Capitolo 5

VoRaQue: Architettura e Implementazione

In questo capitolo verrà esaminato VoRaQue da un punto di vista architetturale ed implementativo. In particolare verranno descritti i package, verranno mostrate le principali scelte progettuali e le classi definite.

5.1 Package

Iniziamo vedendo la struttura dei package che compongono VoRaQue. Il diagramma[1] seguente mostra quelli creati e le relazioni di dipendenza tra di essi:

- **user**: contiene le classi che interagiscono con l'utente, permettendogli di eseguire delle query che vengono poi inviate al server.
- **server**: in questo package sono presenti le classi che definiscono il comportamento del Server che rappresenta il punto centrale di tutta l'applicazione. Tale

[1] In questo e nei successivi diagrammi delle classi verrà mostrato soltanto un sottoinsieme delle operazioni e delle proprietà effettivamente presenti nelle classi/interfacce, allo scopo di non appesantire troppo i diagrammi stessi. Per maggiori dettagli riferirsi al codice sorgente.

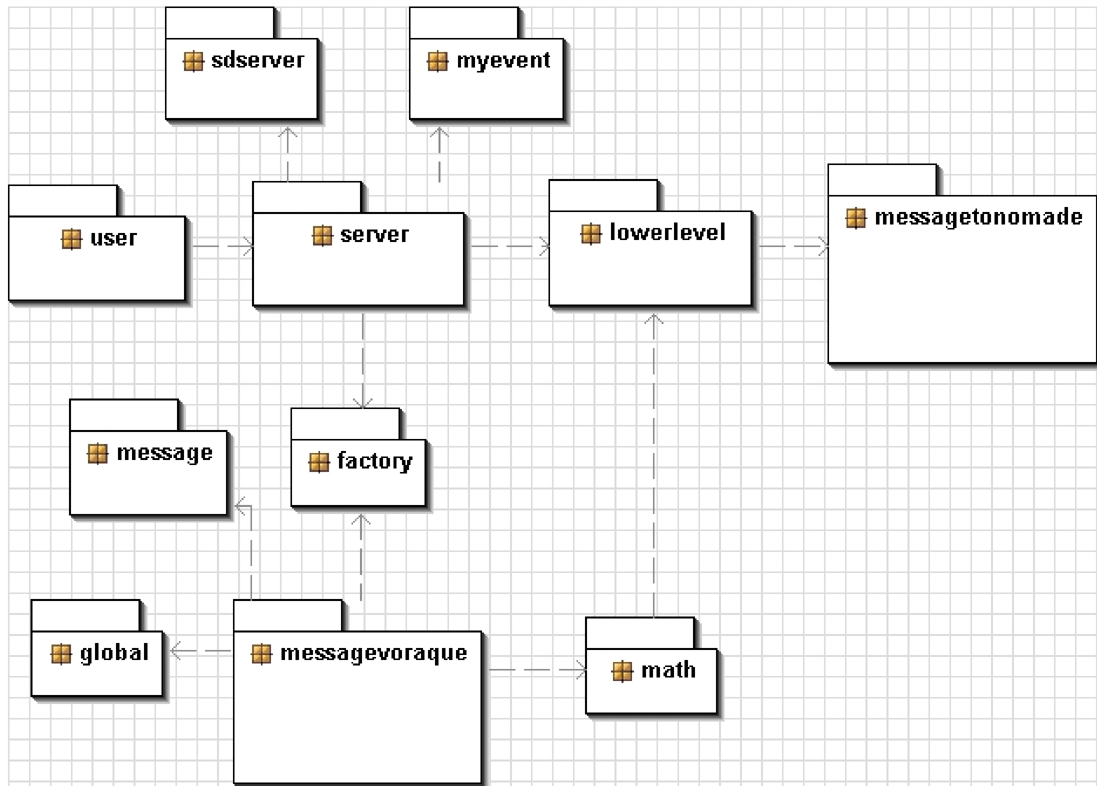


Figura 5.1: La struttura dei Package di VoRaQue.

oggetto può ricevere richieste sia da parte dell'utente che dalla rete e il suo compito è quello di elaborarle.

- **sdserver**: contiene le strutture dati utili per le computazioni effettuate dal **Server**, ad esempio, la lista con tutti gli identificatori dei messaggi elaborati. Tali oggetti verranno descritti in dettaglio nel paragrafo 4.5.

- **myevent**: in questo package viene definita una **coda ad eventi**, ovvero, ogni volta che viene inserito un nuovo oggetto, viene lanciato un evento di tipo **QueueInsertEvent** per informare il **Server** di una nuova richiesta da elaborare.

- **lowerlevel**: in questo package è stata definita l'interfaccia **LowerLevel** che interagisce con le classi che implementano le funzionalità relative alla gestione di una rete P2P la cui overlay network è un diagramma di Voronoi 2D [50]. Il si-

stema prende il nome di **Nomade** e l'autore ha gentilmente concesso l'uso delle classi sviluppate nel suo tirocinio.

- **messagetonomade**: contiene i messaggi che vengono spediti a **Nomade** con i corrispondenti gestori. Tali richieste vengono effettuate sia per ottenere le strutture dati relative ai diagrammi di Voronoi, come la lista dei **Voronoi Neighbours**, sia per inoltrare in rete i messaggi definiti nel protocollo di VoRaQue.

- **factory**: contiene le interfacce che sono definite nel design pattern **Factory Method** [51]. Quest'ultimo è stato utilizzato per creare i gestori associati ad ogni messaggio, in questo modo il protocollo dell'applicazione è facilmente estendibile.

- **message**: contiene l'intestazione e la classe **Message** che deve essere estesa da ogni messaggio.

- **messagevoraque**: questo package incapsula le classi che rappresentano tutti i messaggi definiti nel protocollo con i corrispondenti gestori.

- **global**: contiene le strutture dati e le informazioni relative al nodo che possono essere utili al Server per elaborare le richieste, come ad esempio, l'indirizzo IP, la porta e le coordinate del peer nello spazio.

- **math**: il package math contiene la classe che definisce le operazione matematiche utili in particolar modo quando si esegue il compass routing. Ad esempio, possiamo trovare l'implementazione del calcolo per ottenere gli angoli per stabilire la parentela tra i nodi nello spanning tree.

5.2 Gestione ad eventi del Server

Il Server rappresenta la parte centrale di tutta l'applicazione, è un thread, ma la sua caratteristica più importante è che elabora le richieste ogni volta che viene sollevato un evento di tipo **QueueInsertEvent**. Osserviamo la figura 5.2.

Come possiamo vedere, nel package **myevent** vengono definite tre classi e un'interfaccia:

1. **QueueMessage**: questa struttura dati viene implementata tramite una **LinkedList** e mette a disposizione operazioni **sincronizzate** per cancellare o aggiungere oggetti. La politica di inserimento e cancellazione è tipo *FIFO (First-In First-Out)* [52].

2. La classe **QueueInsertEvent** rappresenta l'evento che viene sollevato ogni volta che viene inserito un nuovo oggetto nella coda di tipo **QueueMessage**. Il costruttore richiama quello della superclasse **EventObject**.

3. L'interfaccia **QueueInsertListener** espone la firma del metodo **queueInserted** e tutte le classi che vogliono gestire l'evento **QueueInsertEvent** dovranno implementare tale interfaccia.

4. La classe **MyQueueEvent** incapsula un oggetto **codaEventi** di tipo **QueueMessage** e una lista **_listeners** di tipo **QueueInsertListener**. Di particolare interesse è il metodo **insertObjectInQueueMessage** poiché inserisce l'oggetto passato come parametro in **codaEventi**, fatto questo invoca il metodo **_fireQueueInsertEvent** che crea una nuova istanza di **QueueInsertEvent** e per ogni listener che si è registrato, viene invocato il metodo **queueInserted**. In figura 5.3 mostriamo il codice che descrive le operazioni.

In conclusione, come evidenziato in figura 5.2, il Server mantiene una variabile privata **queueMsgToServer** di tipo **MyQueueEvent** e viene definita una classe interna **ServerListener** che implementa l'interfaccia **QueueInsertListener**. Quest'ultima

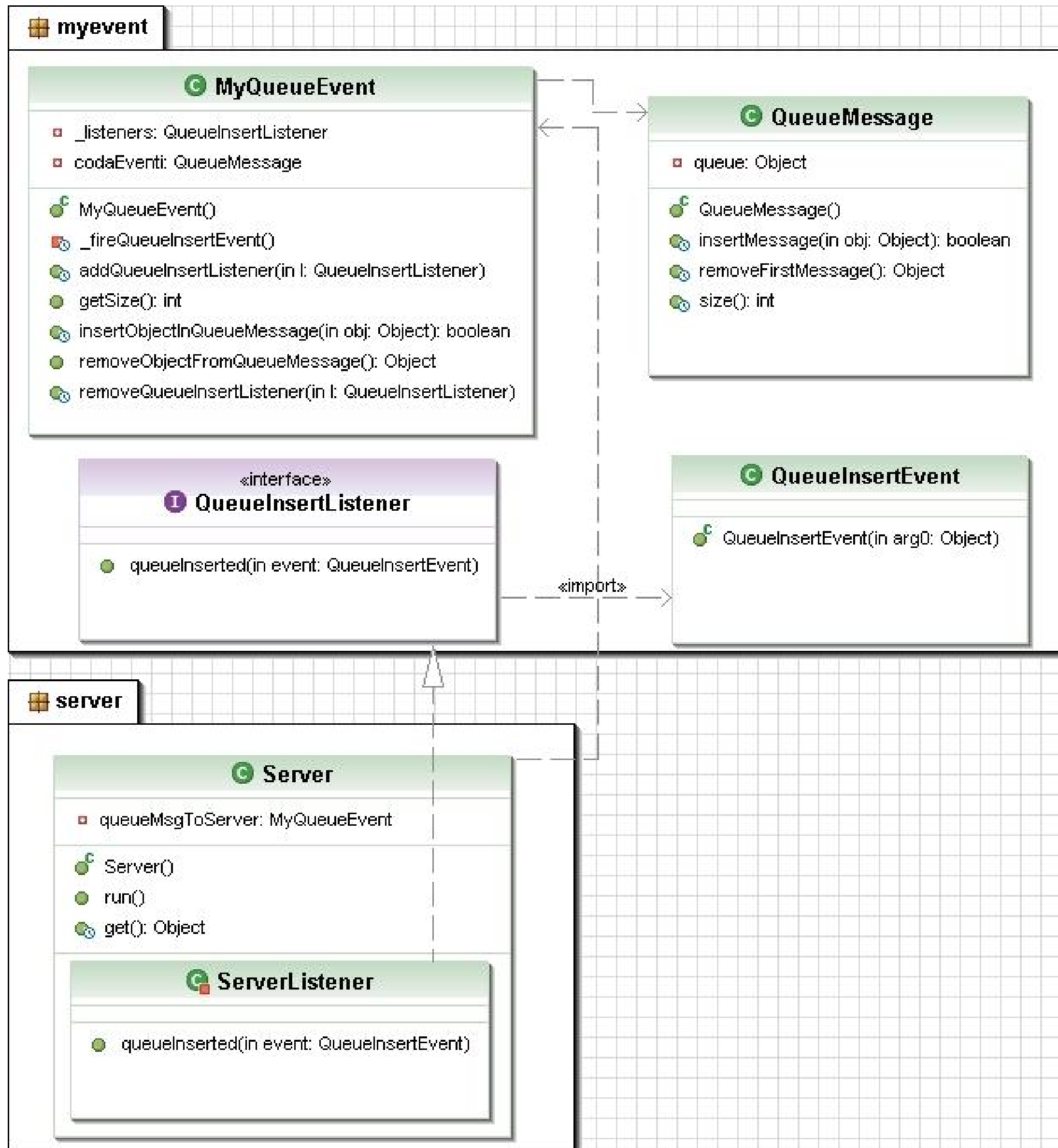


Figura 5.2: Implementazione del Server con coda ad eventi.

```
public class MyQueueEvent {

private QueueMessage codaEventi;
private List<QueueInsertListener> _listeners = new
                                 ArrayList<QueueInsertListener>();

public MyQueueEvent()
  {
     codaEventi = new QueueMessage();
  }

public synchronized boolean insertObjectInQueueMessage(Object obj)
  {
     if(this.codaEventi.insertMessage(obj))
     {
           _fireQueueInsertEvent();
           return true;
     }
     else
           return false;
  }

public synchronized void
                      addQueueInsertListener(QueueInsertListener l)

  {
      _listeners.add( l );
  }


public synchronized void
                   removeQueueInsertListener(QueueInsertListener l)
  {
      _listeners.remove( l );
  }

private synchronized void _fireQueueInsertEvent()
  {
     QueueInsertEvent mood = new QueueInsertEvent(this);
     Iterator listeners = _listeners.iterator();
     while( listeners.hasNext() )
      ((QueueInsertListener)listeners.next()).queueInserted(mood);
  }

public Object removeObjectFromQueueMessage()
  {
     return codaEventi.removeFirstMessage();
  }

public int getSize()
  {
     return codaEventi.size();
  }
}
```

Figura 5.3: Codice MyQueueEvent

viene dichiarata e inizializzata nel metodo **run** e il riferimento viene registrato nella lista dei listener di **queueMsgToServer**. Inoltre, sempre nel **run**, il Server invoca iterativamente il metodo **sincronizzato get** per verificare se la coda di tipo **MyQueueEvent** ha almeno un oggetto al suo interno, in caso negativo il thread viene deschedulato. In questo modo il Server rimane nello stato di **attesa** finché non viene lanciato l'evento che corrisponde all'inserimento di un oggetto in coda.

Quando il Server passa nello stato di **esecuzione**, rimuove il primo oggetto che trova in **queueMsgToServer** e inizia la fase di elaborazione della richiesta. Mentre il Server elabora un oggetto, possono essere generati altri eventi di tipo **QueueInsertEvent**. Da notare che, **il Server non interrompe** l'elaborazione corrente per elaborare il nuovo oggetto, ma continua ad eseguire tutti i calcoli e soltanto nell'istante in cui ha terminato l'elaborazione, può estrarre la nuova richiesta da **queueMsgToServer**.

Un'implementazione alternativa per il Server può essere definita mediante l'approccio basato su **multi-threading** [53], ovvero, ogni volta che si riceve una richiesta viene istanziato un nuovo thread dedicato all'elaborazione di quel messaggio. Una soluzione di questo tipo comporta la gestione di operazioni in mutua esclusione nel caso di modifiche a strutture dati condivise. In realtà si è preferito implementare il Server con una coda ad eventi sia per la sua semplicità, ma in particolar modo perché, nell'istante in cui inizia la fase di elaborazione di una richiesta vengono eseguite molte istruzioni di calcolo vero e proprio – ad esempio, i controlli per verificare se un nodo è un match per una range query, vedere paragrafo 4.3.5 – e un'organizzazione di tipo multi-threading non sarebbe sfruttata e si ricondurrebbe comunque all'esecuzione di un singolo thread alla volta.

Fino a questo momento abbiamo parlato di *oggetti inseriti nella coda ad eventi del Server* ma non è stato specificato chi esegue l'inserimento e cosa viene immesso. Osserviamo la figura 5.4.

Come abbiamo detto sopra, il Server mantiene l'istanza di tipo **MyQueueEvent** che rappresenta la coda ad eventi. Da notare che l'inserimento di oggetti in essa

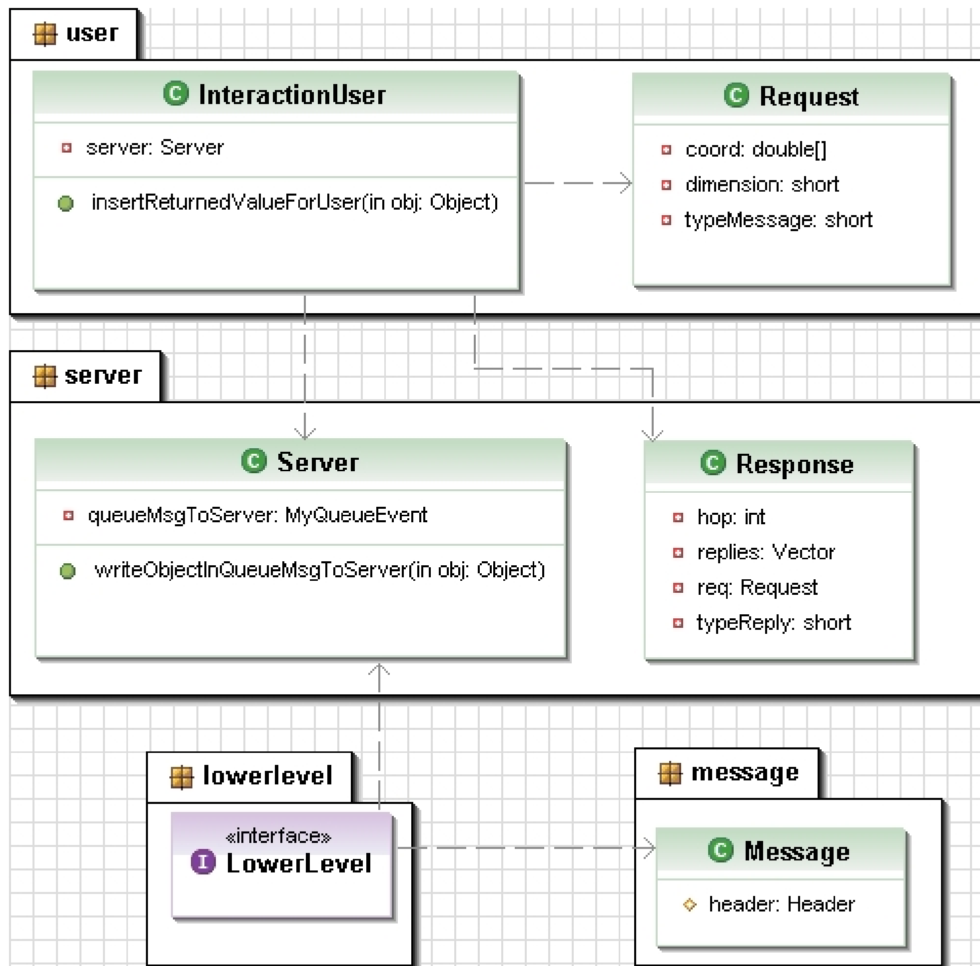


Figura 5.4: Diagramma delle classi InteractionUser - Server - LowerLevel.

avviene tramite il metodo pubblico **writeObjectInQueueMsgToServer(Object obj)** (figura 5.4). La classe **InteractionUser** mantiene il riferimento al Server e ogni volta che l'utente sottomette una query al sistema viene creato un nuovo oggetto **Request** che contiene le coordinate della richiesta, il numero delle dimensioni del diagramma di Voronoi e la tipologia di interrogazione che è stata effettuata, ovvero query esatta o range query. L'operazione successiva è l'invocazione del metodo pubblico per inserire l'oggetto **Request** nella coda del Server. L'altra entità che interagisce con l'istanza di tipo **MyQueueEvent** appartiene al package **lowerlevel** ed è la classe che andrà ad estendere l'interfaccia contenuta in esso. In questo caso, vengono inseriti oggetti di tipo **Message**, ovvero i messaggi descritti nel protocollo di VoRaQue provenienti dalla rete.

5.3 Elaborazione di una richiesta

Nel paragrafo precedente abbiamo visto come la classe Server gestisce le richieste in arrivo dalla rete o dall'utente. La fase successiva è l'elaborazione dell'oggetto estratto dalla coda, ma prima osserviamo la figura 5.5.

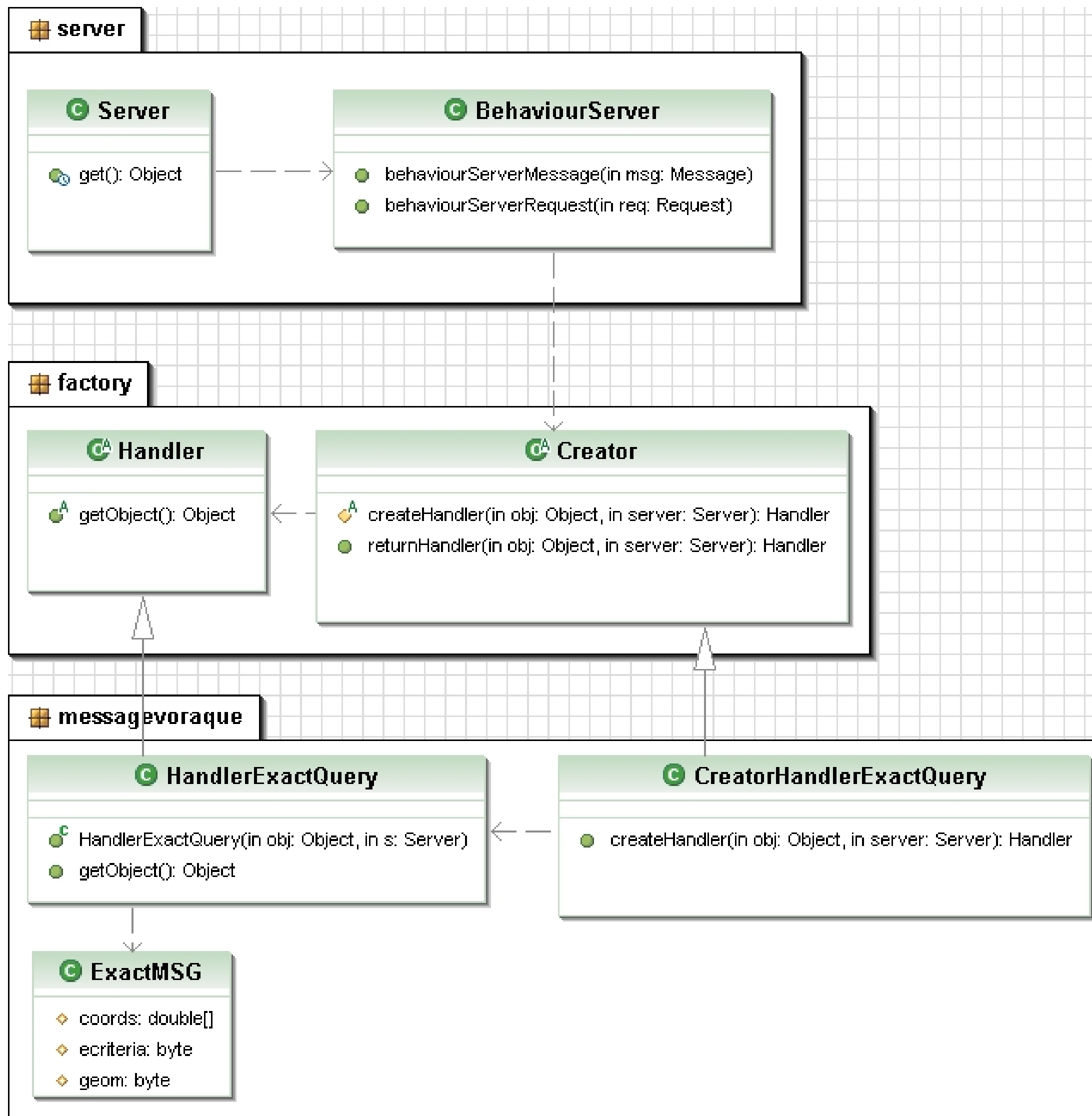


Figura 5.5: Elaborazione di un messaggio.

Come possiamo vedere in figura 5.5 viene utilizzato il design pattern **Factory**

Method [51] che definisce un'interfaccia (**Creator**) per ottenere una nuova istanza di un oggetto (**Handler**) delegando ad una classe derivata (**CreatorHandlerExactQuery**) la scelta di quale classe istanziare (**HandlerExactQuery**). Nel caso specifico stiamo considerando una richiesta di query esatta.

Il Server, dopo aver estratto dalla coda un oggetto, ne verifica il tipo e invoca il metodo corrispondente della classe **BehaviourServer** (figura 5.5). Entrambi creano un **Creator** appropriato a seconda del valore numerico della richiesta. La risoluzione da parte del Server di una query esatta avviene sempre tramite l'istanziazione di **HandlerExactQuery** indipendentemente dalla sua provenienza. Inoltre, soltanto le richieste di query esatta e range query possono giungere sia dalla rete, sotto forma di oggetti di tipo **Message**, che direttamente dall'utente, sotto forma di istanze di tipo **Request**.

Creato il gestore associato alla richiesta, il Server esegue le operazioni definite al suo interno e nel caso specifico, se proviene dall'utente avremo 2 possibilità:

1. la query viene risolta e può essere restituita un risultato evitando di inviare il pacchetto in rete.
2. Viene costruito un messaggio di tipo **ExactMSG** per essere spedito.

Se è un oggetto proveniente dalla rete si possono verificare 2 casi:

1. se il messaggio di tipo ExactMSG non può essere risolto[2], il pacchetto viene inoltrato al nodo vicino.
2. Se il nodo è in grado di risolvere la richiesta, costruisce un messaggio di tipo **ReplyExactMSG** da spedire direttamente al richiedente.

In generale, se al termine dell'elaborazione di una richiesta generica viene restituito un oggetto di tipo Message, questo sarà sicuramente inoltrato in rete. In tutti gli

[2]Un messaggio di tipo **ExactMSG** non viene risolto, se il nodo che lo elabora verifica che non contiene il punto definito dalla query nella sua regione di Voronoi, quindi non è in grado di restituire ne una risposta positiva ne negativa. (vedere capitolo 4).

altri casi, il risultato elaborato viene restituito direttamente all'utente. Da notare che l'utente interagisce con l'applicazione tramite un'interfaccia grafica, ma questa viene gestita unicamente dall'**Event Dispatcher Thread (EDT)**. Ciò significa che il Server non può direttamente restituire all'utente i risultati ottenuti, ma è necessario che deleghi il compito ad EDT. Quest'operazione viene fatta utilizzando il metodo **invokeLater** della classe **java.swing.EventQueue**. In questo modo viene affidato ad Event Dispatcher l'esecuzione di un **Runnable** e viene restituito immediatamente il controllo al thread corrente. Di seguito si mostra lo snippet di codice relativo all'uso del metodo invokeLater in figura 5.6.

```
java.awt.EventQueue.invokeLater(new Runnable() {
      public void run() {
            if(obj.getClass().equals(Response.class))
                  panel.writeResultResponse((Response)obj);

            if(obj.getClass().equals(String.class))
                  panel.writeResultString((String)obj);
      }
  });
```

Figura 5.6: Uso del metodo invokeLater.

La variabile panel rappresenta l'oggetto dell'interfaccia grafica sul quale EDT va a scrivere. La risposta può essere incapsulata in una stringa oppure in una istanza di **Response**, che contiene tutte le informazioni relative ai risultati ottenuti (lista di indirizzi ip e porta, coordinate del match e gli hop necessari, figura 5.4). E' importante sottolineare che **Event Dispatcher** può eseguire il **Runnable** soltanto dopo aver consumato tutti gli eventi pendenti.

5.4 Nomade e lowerlevel Package

Nomade [50] è un sistema P2P che gestisce una overlay network definita come un diagramma di Voronoi 2D e nasce nell'ambito degli ambienti virtuali distribuiti. Questo sistema si ispira alle ricerche che hanno portato alla definizione di VON (Voronoi-based Overlay Network) [31] e condivide con quest'ultimo gran parte dei concetti, anche se il protocollo di comunicazione è stato sostanzialmente riprogettato.

L'idea che sta alla base di Nomade è quella di associare ad ogni nodo una coppia di coordinate appartenenti ad un piano. Tali coordinate corrispondono alla posizione del relativo **avatar**[3] all'interno del mondo virtuale e cambiano in funzione degli spostamenti di quest'ultimo. Ad ogni nodo viene inoltre associata un'area di interesse[4] che rappresenta il raggio visivo/raggio d'azione dell'avatar nel mondo virtuale. Il punto chiave di Nomade è che ogni nodo dovrebbe interagire esclusivamente con i peer le cui coordinate giacciono all'interno della propria area di interesse attraverso delle connessioni dirette con essi. Così facendo si limita il numero di messaggi scambiati da ogni nodo ed il numero di connessioni che questo deve gestire, garantendo perciò una considerevole scalabilità.

Inoltre ogni nodo gestisce un proprio diagramma di Voronoi in cui è presente un sito per il nodo stesso ed un sito per ogni suo vicino sull'overlay network. Tale diagramma consente di classificare i vicini come **boundary neighbours** e/o **enclosing neighbours**. I primi agiscono come delle "vedette", consentendo la scoperta di nuovi vicini in seguito ad uno spostamento, mentre i secondi permettono di mantenere connessa l'overlay network e possono trovarsi anche al fuori dell'area di interesse.

[3]Un avatar identifica il personaggio controllato dal giocatore o una immagine che lo identifica. [60]

[4]L'area di interesse di Nomade ha una funzionalità completamente diversa dall'area di interesse definita da VoRaQue. Nel primo caso rappresenta la visione del mondo di un avatar. Nel secondo caso, identifica una superficie descritta dalla query nello spazio bi-dimensionale che contiene i risultati per la richiesta.

5.4.1 Nomade API

In questo paragrafo vedremo come utilizzare Nomade a livello applicativo [50].

Attivazione di un nodo. L'attivazione di un nodo avviene istanziando la classe Nomade, il cui costruttore prevede i seguenti parametri:

- **mapWidth**: specifica la larghezza della mappa del mondo virtuale (che corrisponde poi alla larghezza del diagramma di Voronoi).
- **mapHeight**: specifica l'altezza della mappa del mondo virtuale (che corrisponde poi all'altezza del diagramma di Voronoi).
- **aoiRadius**: indica la dimensione del raggio dell'area di interesse del nodo.
- **nicAddr**: indica a Nomade su quale interfaccia di rete si vuole effettuare il binding del socket.
- **logToConsole**: Se impostato a true, l'attività di logging avviene su console, altrimenti avviene sul file "nomade.log".

Si noti che i primi tre parametri devono essere gli stessi per ogni istanza della stessa applicazione. Una volta attivato un nodo è possibile ottenere le informazioni relative ad esso attraverso i seguenti metodi:

- **getIPAddress()**: restituisce l'indirizzo IP associato al nodo (lo stesso specificato nel costruttore) sotto forma di stringa.
- **getPortNumber()**: restituisce il numero di porta associato al nodo. Viene generato casualmente da Nomade nell'intervallo [49152, 65535] (porte non riservate).
- **getNodeId()**: restituisce l'identificatore associato al nodo.

L'interazione tra Nomade e l'applicativo avviene scambiando messaggi attraverso una coppia di code sincronizzate. Con i metodi *getInputQueue()* e *getOutputQueue()* si ottengono rispettivamente le code da usare per ricevere ed inviare messaggi. E' importante che ogni volta che si vuole inviare un messaggio si provveda ad istanziare un nuovo oggetto, anziché limitarsi a modificare i campi di un messaggio preesistente.

Associazione ad una overlay network esistente. Dopo aver attivato un nuovo nodo occorre richiedere esplicitamente l'associazione ad una overlay network esistente. Per fare ciò basta inserire un messaggio di tipo **JoinRequestV1** nella coda di output. Nel messaggio deve essere specificata la posizione iniziale da associare al nodo e l'indirizzo IP ed il numero di porta di un qualsiasi altro nodo già attivo sulla overlay network a cui vogliamo associarci. Non esiste alcun 'ack' esplicito per avvertire l'applicazione del fatto che l'associazione è avvenuta con successo, ma si può desumere dalla ricezione di uno o più messaggi di tipo **NewAvatarV1** che notificano la scoperta di nuovi vicini. Se non viene ricevuto alcun messaggio NewAvatarV1 nel giro di qualche secondo, si consiglia di provare ad inviare nuovamente un **JoinRequestV1** modificando la posizione iniziale. Se le coordinate iniziali specificate nella richiesta giacciono al di fuori dei confini della mappa, Nomade risponderà con un messaggio di tipo **ErrorMessageV1** avente come codice di errore **POSIZIONE_FUORI_MAPPA**.

Keep alive e spostamento. Una volta associati ad una overlay network occorre inviare periodicamente un messaggio di tipo **PositionUpdateV1** in cui si deve specificare la posizione corrente dell'avatar. Al momento dell'invio è possibile anche incapsulare nel **PositionUpdateV1** del payload applicativo da recapitare ai vicini. Anche in questo caso è possibile ricevere come risposta un **ErrorMessageV1** contenente uno dei seguenti codici di errore:

- **POSIZIONE_FUORI_MAPPA**: indica che la posizione specificata giace al di fuori dei confini della mappa.

- **POSIZIONE_OCCUPATA**: indica che si è tentato lo spostamento in un punto già occupato da un altro avatar.

- **VELOCITA_ECCESSIVA**: se la distanza di spostamento è eccessiva. Inquesta versione di Nomade non è possibile percorrere, con un solo spostamento, una distanza superiore al 70% del raggio dell'area di interesse.

In tutti e tre i casi il PositionUpdateV1 viene ignorato, conservando la posizione precedente all'invio. Al fine di determinare quale sia questa posizione, in **ErrorMessageV1** è stato inserito il campo **currentPosition** che la contiene. Il campo **requestedPosition** indica invece la posizione richiesta che ha scatenato l'errore.

Creazione di una nuova overlay network. Per creare una nuova overlay network basta iniziare direttamente ad inviare i **PositionUpdateV1** omettendo la **JoinRequestV1**.

Scoperta dei vicini. Ogni volta che viene scoperto un nuovo vicino, si riceve un messaggio di tipo **NewAvatarV1**. Esso contiene l'identificatore del nodo appena scoperto, la posizione del relativo avatar all'interno del mondo virtuale ed un eventuale payload applicativo da esso inviato. Si noti che **NewAvatarV1** non fa distinzione tra le scoperte causate dall'inserimento di un nuovo nodo nell'overlay network, e quelle causate dallo spostamento di nodi già esistenti.

Spostamento di un vicino. Ogni volta che un nodo vicino modifica la sua posizione, si riceve un messaggio di tipo **PositionUpdateV1** contenente l'identificatore del nodo che si è spostato, la sua nuova posizione ed un eventuale payload applicativo. Si noti che questi messaggi non vengono ricevuti periodicamente, ma solo in seguito ad un effettivo spostamento e/o alla presenza di payload.

Perdita di vicinanza. Ogni volta che un vicino diventa non più tale, si riceve un messaggio di tipo **ExitNotificationV1**. Questo contiene l'identificatore 'dell'ex vicino' ed un flag booleano, chiamato **leaving**, che indica se il nodo oggetto del messaggio ha lasciato la rete o si è semplicemente spostato in un punto troppo distante (per mantenere la relazione di vicinanza). Si noti che in seguito alla massiccia uscita di nodi dall'overlay network può capitare di perdere la connettività. Al fine di individuare una simile situazione, si consiglia di conteggiare il numero totale di **NewAvatarV1** e di **ExitNotificationV1**, prestando attenzione che la loro differenza sia maggiore di zero.

Uscita dall'overlay network. Per lasciare l'overlay network basta inserire nella coda di output un messaggio di tipo **LeaveRequestV1** (non contiene alcun campo). Si noti che l'invio di tale messaggio non da luogo automaticamente alla disattivazione del nodo. Questo può risultare utile ad esempio quando si vogliono effettuare dei grandi spostamenti (es. per simulare un teletrasporto). Basta infatti richiedere l'uscita dalla rete con una **LeaveRequestV1** e poi richiedere dopo qualche secondo un nuovo inserimento (per mezzo di una **JoinRequestV1**). Tuttavia questo espediente non permette di modificare l'identificatore del nodo, per cui si consiglia di effettuare il reinserimento dopo un periodo di tempo sufficientemente lungo e possibilmente specificando una posizione iniziale ben distante da quella precedente.

Disattivazione di un nodo. Per disattivare un nodo basta invocare il metodo **close()** della classe Nomade. Tale metodo può bloccare per qualche secondo. Un nodo disattivato non può più essere riattivato.

5.4.2 Interazione VoRaQue - Nomade

Passiamo ora ad esaminare come ogni istanza di VoRaQue comunica con i nodi vicini (figura 5.7).

Come si nota in figura, l'interazione tra l'applicazione di alto livello e Nomade avviene inserendo/prelevando oggetti, che chiameremo messaggi, in/da due **code sin-**

Figura 5.7: Interazione VoRaQue - Nomade - Rete

cronizzate. Inoltre, viene impiegato un socket UDP per inviare (ricevere) messaggi a (da) nodi remoti. E' importante sottolineare che Nomade mette a disposizione per l'applicazione di alto livello le API per invocare funzionalità utili per gestire l'overlay network, quali l'inserimento, la cancellazione e la gestione del keep alive [50]. Come specificato sopra, queste operazioni possono essere invocate da VoRaQue inviando nella coda di output i messaggi definiti nel protocollo di Nomade. Le classi implementate nel package lowerlevel si occupano dell'integrazione tra le due applicazioni e si preoccupano anche di convertire i dati che vengono scambiati tra VoRaQue e Nomade. Osserviamo la figura 5.8.

L'interfaccia **LowerLevel** mette a disposizione del Server una lista di metodi. La loro implementazione si trova nella classe **LowerLevelWrapper**, che ha il compito di convertire la richiesta proveniente dall'applicazione di alto livello in una richiesta che possa essere soddisfatta da Nomade. Ad esempio, supponiamo che VoRaQue invochi il metodo per la join. In questo caso, il LowerLevelWrapper si preoccupa di creare e mandare in esecuzione il thread **VonNodeThread** che a sua volta istanzia un oggetto di tipo Nomade e inserisce nella sua coda di output la richiesta di inserimento nella rete di overlay sotto forma di un messaggio di **JoinRequestV1**, che è definito nel protocollo di Nomade. Il thread aspetta quindi la conferma[5] dalla coda in input. Al termine, il LowerLevelWrapper esegue la conversione del risultato restituito da Nomade nel tipo di risultato atteso da VoRaQue.

In figura 5.8 viene definito anche un package **messagetonomade**. Come abbiamo

[5] In realtà, come definito nelle API di Nomade, la conferma si desume dalla ricezione di messaggi di tipo NewAvatarV1.

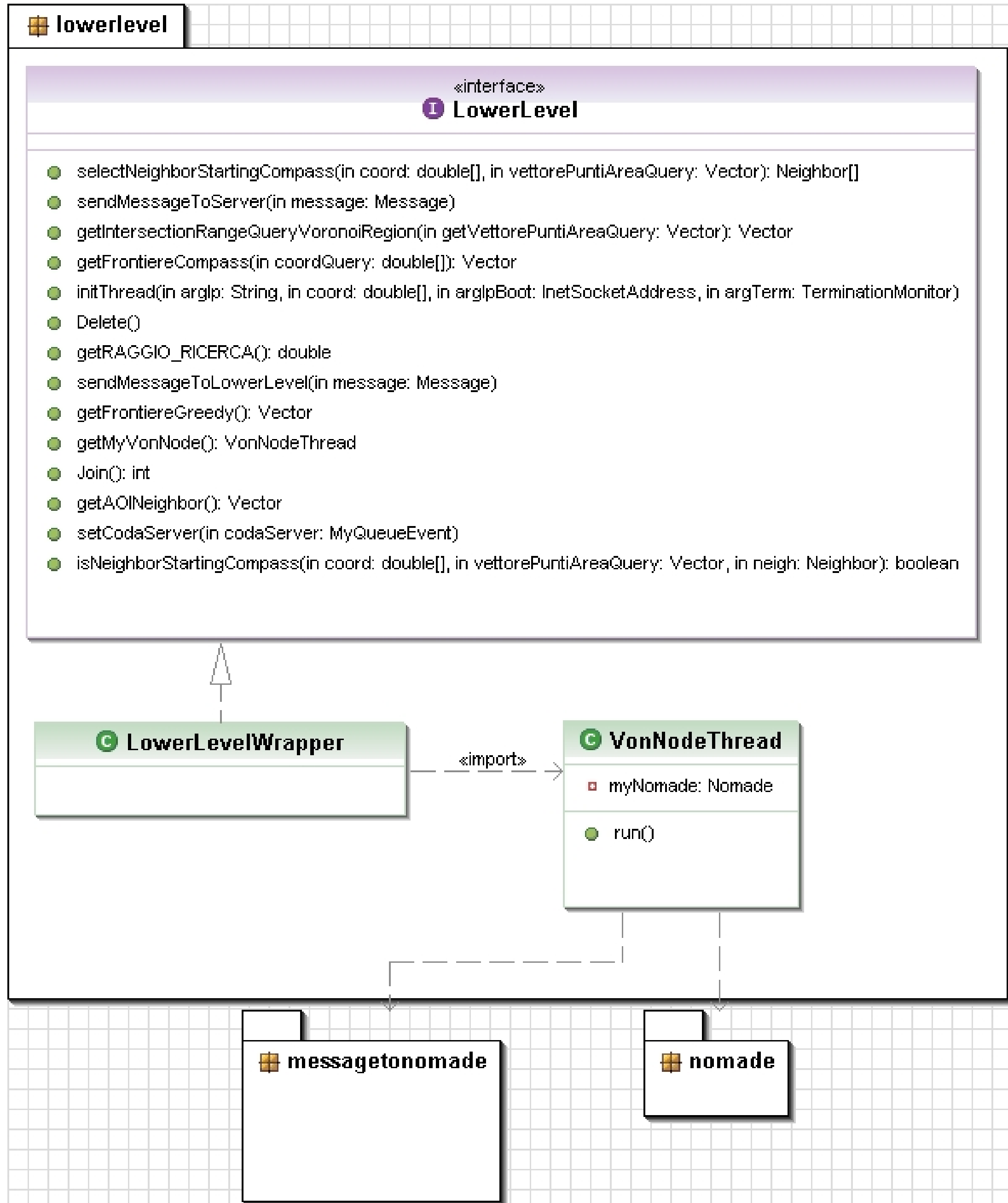


Figura 5.8: Integrazione VoRaQue - Nomade

visto all'inizio del capitolo, in esso sono implementati tutti i messaggi che vengono spediti all'applicazione di basso livello con i corrispondenti gestori. Per fare questo, si è scelto di incapsulare gli oggetti di tipo Message dell'applicazione di alto livello in messaggi che potessero essere manipolati da Nomade per essere inoltrati sull'overlay network. Osserviamo il diagramma delle classi di figura 5.9.

Come possiamo vedere, nel package messagetonomade di VoRaQue sono definiti due messaggi, **ApplicationMessage** e **ApplicationMessageRequest**, che hanno come variabile privata un oggetto Message, e due gestori, **ApplicationMessageHandler** e **ApplicationMessageRequestHandler**. Ognuno di essi implementa la corrispondente interfaccia definita in Nomade. Da notare che, quest'ultimo utilizza un **dispatcher** per elaborare i messaggi scegliendo il gestore appropriato, affinché ciò sia possibile, è necessario eseguire un'operazione di registrazione messaggio-gestore.

Ma vediamo come queste classi vengono utilizzate. Nell'istante in cui il Server invoca il metodo **sendMessageToLowerLevel** (figura 5.9) per spedire un pacchetto in rete, il LowerLevelWrapper crea un'istanza di ApplicationMessageRequest e inizializza la sua variabile privata **message** (figura 5.9) con il messaggio del Server. Fatto questo, viene invocato VonNodeThread che inserisce nella coda di output il messaggio ApplicationMessageRequest. Il dispatcher invoca il gestore corrispondente (ApplicationMessageRequestHandler) che costruisce il pacchetto ApplicationMessage e lo invia tramite il socket UDP.

L'istanza di Nomade che riceve dalla rete questo messaggio, via dispatcher invoca ApplicationMessageHandler, il quale, mantenendo il riferimento all'oggetto di tipo LowerLevel, estrae l'oggetto Message dal pacchetto ricevuto e viene passato come parametro al metodo **sendMessageToServer** (figura 5.9) per inserirlo nella coda del Server.

Abbiamo visto come l'interazione VoRaQue - Nomade permetta la spedizione e la ricezione dei messaggi definiti nel capitolo 4. L'applicazione di basso livello mantiene le strutture dati per la gestione del diagramma di Voronoi del nodo corrente. Queste

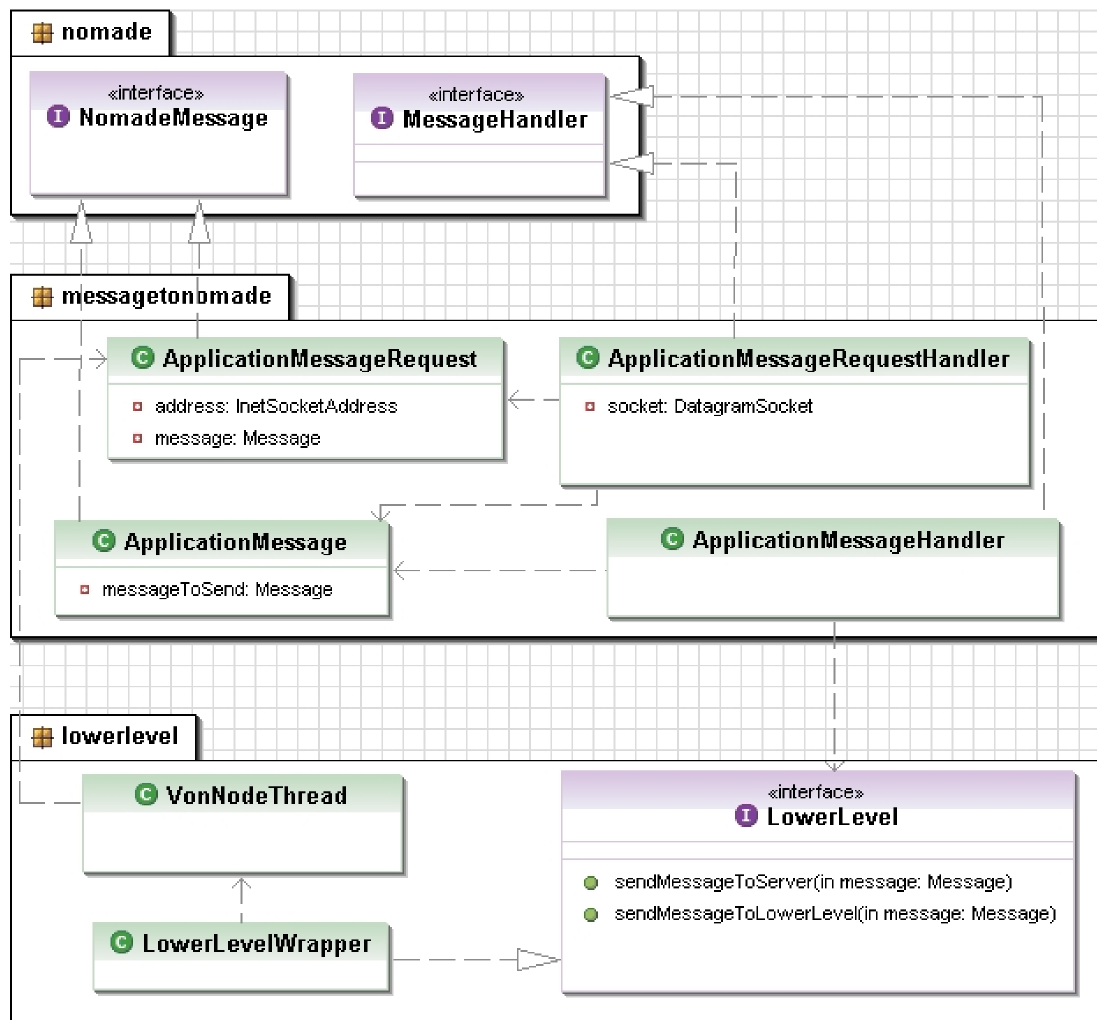


Figura 5.9: Messaggio di tipo ApplicationMessage.

informazioni sono utili per l'elaborazione dei messaggi da parte del Server, in particolar modo quando devono essere elaborate richieste di query esatte o range query che richiedono la conoscenza dei **Voronoi Neighbours e/o Close Neighbours** (vedere capitolo 4). Tali strutture dati non possono essere accedute direttamente, ma anche in questo caso viene inviata una richiesta a Nomade tramite la coda in output e si attende la risposta sulla coda in input. Osserviamo la figura 5.10.

Nel package messagetonomade sono definiti i messaggi **VoronoiNeighborMessage** e **VoronoiNeighborReply** e il gestore associato al primo. Ognuno di essi implementa la corrispondente interfaccia definita in Nomade.

La classe VoronoiNeighborMessage mantiene una variabile booleana che sta ad indicare se sono richiesti soltanto i vicini confinanti oppure sia i Voronoi Neighbours che i Close.

Nell'istante in cui il Server invoca il metodo **getFrontiereGreedy** dell'interfaccia LowerLevel, il LowerLevelWrapper crea un'istanza di VoronoiNeighborMessage e viene invocato VonNodeThread che inserisce nella coda di output il messaggio. Il dispatcher invoca il gestore corrispondente (**VoronoiNeighborMessageHandler**), il quale crea la reply VoronoiNeighborReply che restituisce una *copia* della lista dei vicini. Da notare che in questo caso viene definito un gestore in meno perché è VonNodeThread che si preoccupa di estrarre dal messaggio la struttura dati e restituirla al Server. Inoltre, quest'ultimo esegue un'**attesa attiva**, ovvero, per poter portare avanti l'elaborazione deve aspettare la risposta da Nomade. Nel caso precedente, nel quale viene inviato un messaggio in rete, non appena il Server ha invocato il metodo **sendMessageToLowerLevel** (figura 5.9), può iniziare a elaborare la nuova richiesta se questa è in coda, altrimenti passa nello *stato di attesa*, in particolare **attesa passiva**.

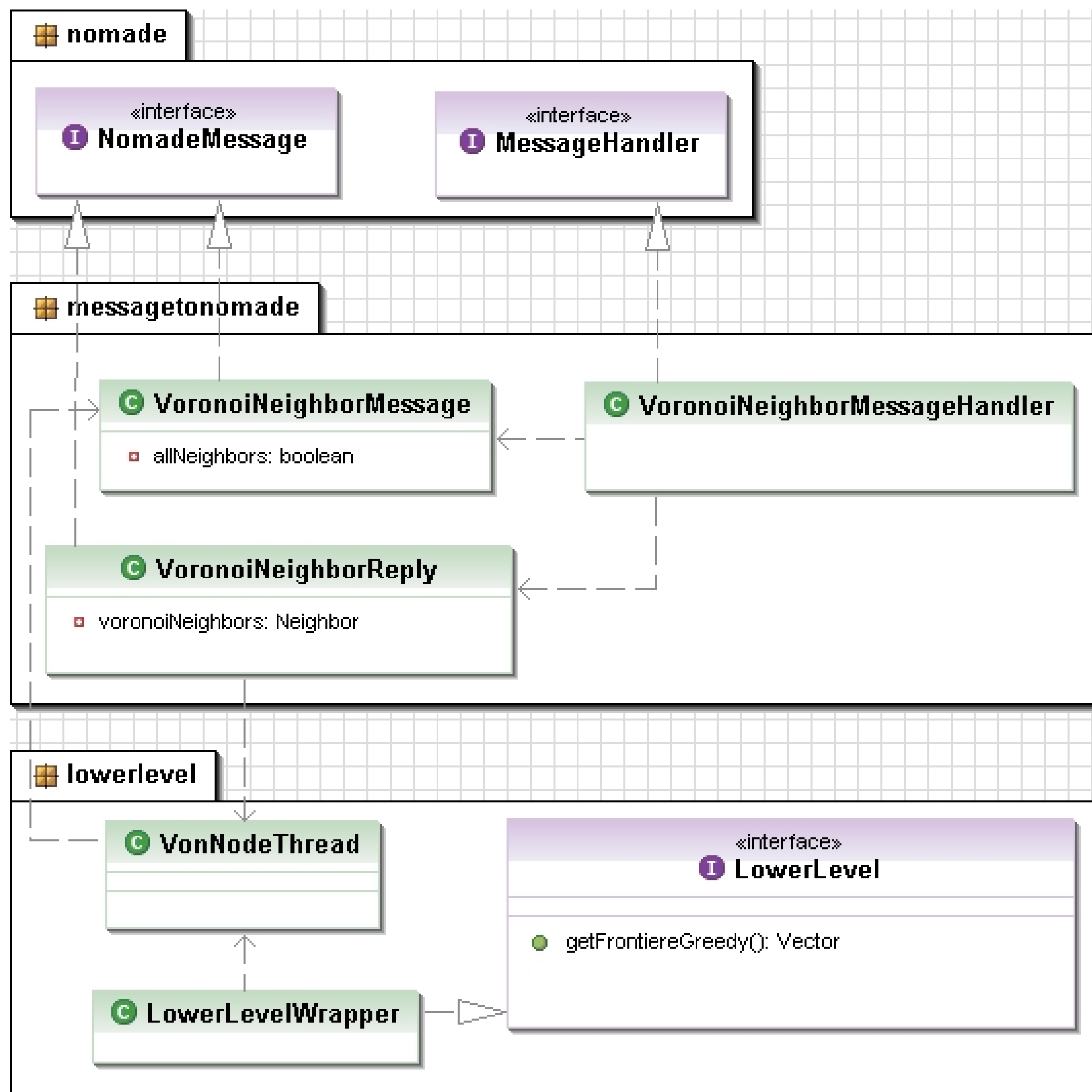


Figura 5.10: Messaggio di tipo VoronoiNeighborMessage.

5.5 Strutture Dati utilizzate

Nei seguenti sottoparagrafi vengono mostrate le strutture dati utilizzate dal Server che sono definite nel package **sdserver**.

5.5.1 QueryStory

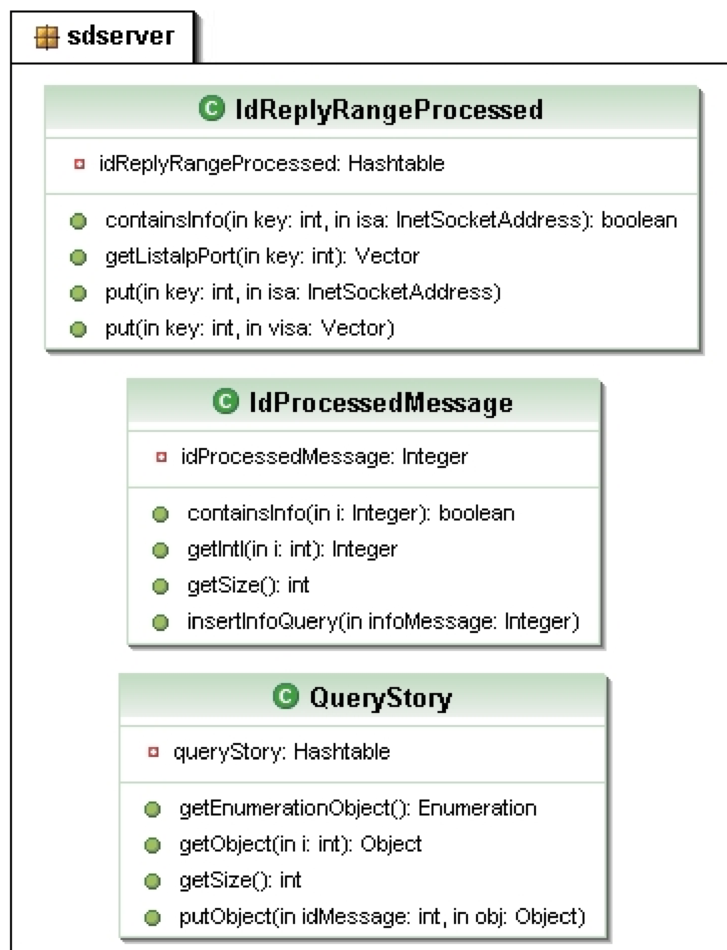


Figura 5.11: Strutture dati in sdserver.

QueryStory (figura 5.11) è una struttura dati che viene utilizzata per tenere traccia di tutte le query che sono state eseguite dall'utente in una sessione e viene implementata utilizzando un oggetto di tipo **java.util.HashTable<Integer, Object>**. Per ogni

richiesta vengono registrati i corrispondenti risultati. Supponiamo che venga invocata la risoluzione di una query esatta. Abbiamo visto che, sottomessa una richiesta da parte dell'utente, la classe InteractionUser costruisce un nuovo oggetto di tipo Request (**Req**) e lo inserisce nella coda del Server (figura 5.4). Quest'ultimo, estratta l'istanza, calcola l'idenfiticatore unico associato al messaggio (**ID**) che diventa anche la chiave della tabella hash. Se il nodo è in grado di risolvere la query senza inviare il pacchetto in rete, il Server costruisce l'oggetto Response (**Res**) (figura 5.4) e QueryStory mantiene la coppia **< ID, Res >**.

Se il nodo non è in grado di risolvere la query, prima di inoltrare il messaggio di ExactMSG, il Server crea l'associazione **< ID , Req >** nella tabella hash. Nell'istante in cui il thread riceve la reply dalla rete, viene costruito l'oggetto Response **Res1**, viene rimosso l'oggetto Req da QueryStory e viene creata la nuova associazione **< ID, Res1 >**. In questo modo siamo in grado mantenere la corrispondenza tra richiesta e risposta.

5.5.2 IdProcessedMessage

IdProcessedMessage (figura 5.11) è una struttura dati che viene utilizzata per tenere traccia di tutti gli identificatori dei messaggi provenienti dalla rete che sono stati già elaborati e viene implementata con un oggetto di tipo **java.util.LinkedList<Integer>**. Ogni volta che il Server riceve un messaggio dalla rete, verifica se l'identificatore del pacchetto è presente in questa struttura dati. In caso affermativo, viene costruita una reply oppure viene restituito un messaggio di errore. In caso negativo, l'intero viene inserito in IdProcessedMessage e il messaggio viene elaborato.

5.5.3 IdReplyRangeProcessed

IdReplyRangeProcessed (figura 5.11) è una struttura dati che viene implementata con un'istanza di **java.util.HashTable<Integer, java.util.Vector<InetSocketAddress»** e viene utilizzata per tenere traccia dei messaggi di tipo **ReplyRangeMSG** provenienti dalla rete che sono stati già elaborati. E' importante sottolineare che questa struttura

dati ha la stessa funzione di **IdProcessedMessage**, ma il motivo per il quale è stata introdotta risale al protocollo per la risoluzione di range query descritto nel capitolo 4. Come abbiamo visto, nella fase in cui viene applicato il compass routing il nodo che ha generato la richiesta può ricevere più di una reply e poiché, *le risposte mantengono lo stesso identificatore della richiesta*, se fosse utilizzata solo IdProcessedMessage, tutti i messaggi di tipo ReplyRangeMSG che vengono ricevuti dopo il primo verrebbero scartati con la conseguente perdita di risultati. Per eliminare questo problema è stata introdotta IdReplyRangeProcessed. Quindi, ogni volta che il Server riceve una ReplyRangeMSG, estrae l'identificatore, recupera dalla tabella hash il vettore corrispondente e verifica se l'indirizzo IP e la porta del mittente sono già presenti nella struttura dati. In caso affermativo viene restituito un messaggio di errore. In caso negativo, vengono inserite le informazione del nodo che ha spedito la risposta in IdReplyRangeProcessed e subito dopo vengono recuperati i dati dei match.

5.5.4 VoronoiNeighNeigh

VoronoiNeighNeigh (figura 5.12), è una struttura dati che viene utilizzata con funzionalità di cache quando si applica il compass routing e viene implementata con una **java.util.HashTable<Integer, VoronoiNeighbor>**. La classe **VoronoiNeighbor** mantiene la lista delle informazioni relative ai nodi vicini (voronoiNeighNeigh) e un parametro che rappresenta la 'freschezza' di tali dati (**now**).

Come abbiamo visto nel capitolo 4, un nodo prima di inoltrare il messaggio di RangeMSG a un vicino P deve stabilire se esiste un rapporto padre-figlio con esso. Questo test viene effettuato richiedendo i vicini di Voronoi di P. VoronoiNeighNeigh viene utilizzata per memorizzare per un certo periodo di tempo t i VoronoiNeighbours dei vicini di Voronoi del peer corrente, allo scopo di ridurre il traffico dei messaggi sulla rete. In sostanza, ogni volta che il Server riceve una richiesta di RangeMSG in fase di compass routing, per prima cosa recupera i suoi vicini di Voronoi, per ognuno di essi estrae l'identificatore associato al nodo e verifica se l'entrata di VoronoiNeigh-

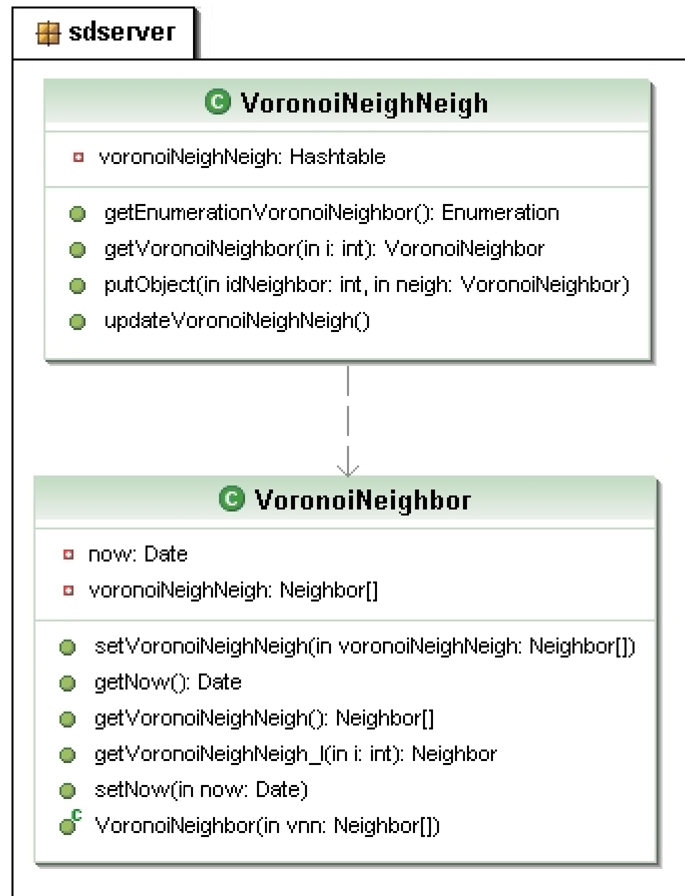


Figura 5.12: VoronoiNeighNeigh.

Neigh non è vuota. In caso affermativo il Server utilizza i vicini nella cache per implementare i controlli richiesti dal compass routing, altrimenti invia messaggi di VoronoiNeighMSG. Da notare che, ogni volta che vengono ricevute delle ReplyVoronoiNeighMSG si aggiorna la tabella hash.

Infine, il periodo di tempo t dipende dal churning [54] registrato da un nodo, ovvero, se un peer verifica un elevato numero di inserimenti e cancellazioni tra i suoi vicini, t dovrà assumere un valore piccolo, al massimo un minuto. In caso contrario, può assumere valori temporali molto più grandi.

5.5.5 MatchRangeQuery

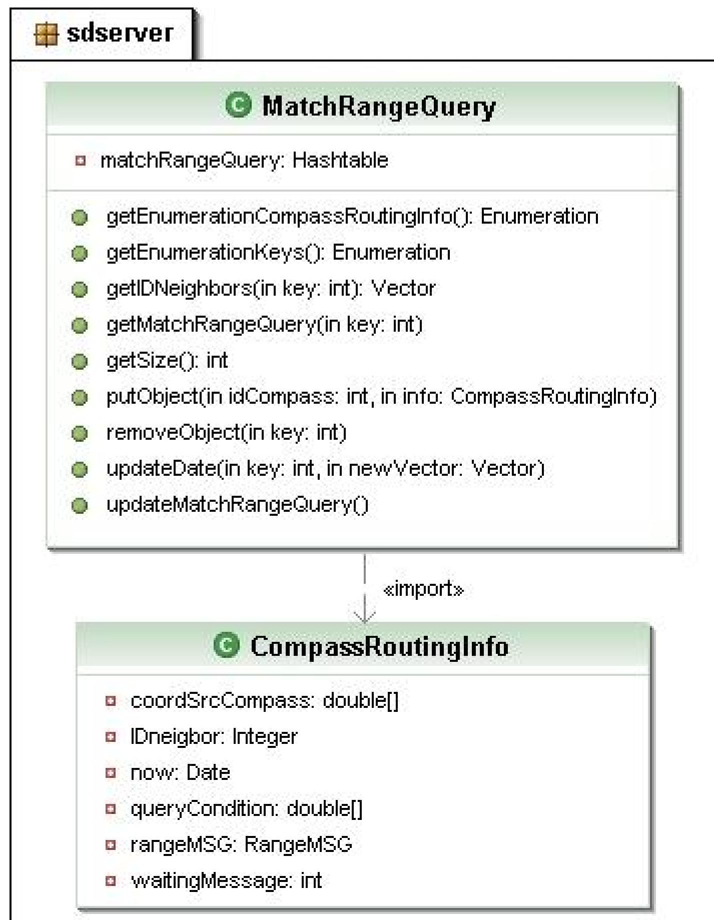


Figura 5.13: MatchRangeQuery.

MatchRangeQuery (figura 5.13), è una struttura dati che viene utilizzata con funzionalità di cache quando si applica il compass routing e viene implementata con una **java.util.HashTable<Integer, CompassRoutingInfo>**. La classe **CompassRoutingInfo** mantiene le informazioni relative all'i-esimo compass routing che il nodo sta eseguendo. In particolare sono memorizzate nella classe, la radice dello spanning tree, la lista degli identificatori dei nodi ai quali è stato inoltrato il messaggio di RangeMSG, la data di ultima modifica dell'oggetto CompassRoutingInfo, le condizioni della query, il riferimento alla richiesta e un valore intero che memorizza il numero di ReplyVoro-

noiNeighMSG che devono essere ancora ricevute. Quando waitingMessage è uguale a zero il compass routing i-esimo è terminato. Questa struttura dati ha due utilizzi importanti:

1. viene utilizzata per permettere al Server di elaborare altre richieste in coda senza dover necessariamente aspettare le risposte dai vicini per ottenere i loro VoronoiNeighbours.

2. Viene utilizzata come cache, ovvero se viene ricevuta la medesima query con la stessa radice per lo spanning tree, è possibile recuperare la lista dei vicini ai quali è stato inoltrato il messaggio di RangeMSG in precedenza. In questo modo viene ridotto il tempo di elaborazione impiegato dal Server.

Infine, è importante sottolineare che vengono applicati controlli per verificare la *freschezza* delle informazioni contenute in MatchRangeQuery. In particolare, l'entrata i-esima viene memorizzata per un periodo di tempo t_1 per attendere la terminazione compass routing. Successivamente, a compass routing terminato, l'entrata i-esima viene memorizzata per un periodo di tempo t_2 per stabilire un intervallo temporale di validità dell'informazione in cache.

Capitolo 6

Risultati sperimentali

In questo capitolo vengono descritti gli esperimenti effettuati per dimostrare l'efficacia e l'efficienza di VoRaQue.

6.1 Interazione con il sistema

Questo paragrafo mostra come interagire con VoRaQue, in particolare vengono evidenziati i passi fondamentali da eseguire e come richiedere al sistema la risoluzione di una query. Ogni volta che viene avviato l'applicativo è necessario inserire l'indirizzo IP e la porta del nodo di bootstrap[1], l'indirizzo IP e le coordinate della nuova istanza di VoRaQue. Al termine di questa prima fase di inizializzazione, occorre avviare l'operazione di inserimento nell'overlay, utilizzando il bottone evidenziato in rosso in figura 6.1.

Se l'operazione di connessione ha avuto esito positivo, nel panel **Info** vengono visualizzate le informazioni del nodo corrente, che sono lo **stato corrente**, l'**indirizzo ip**, la **porta**, l'**identificatore del nodo** e le **coordinate nello spazio**. A questo punto è possibile sottomettere delle richieste al sistema. Osserviamo la figura 6.2.

[1] Il nodo di bootstrap è un nodo già attivo con coordinate (0,0) che serve esclusivamente come punto di ingresso nell'overlay network.

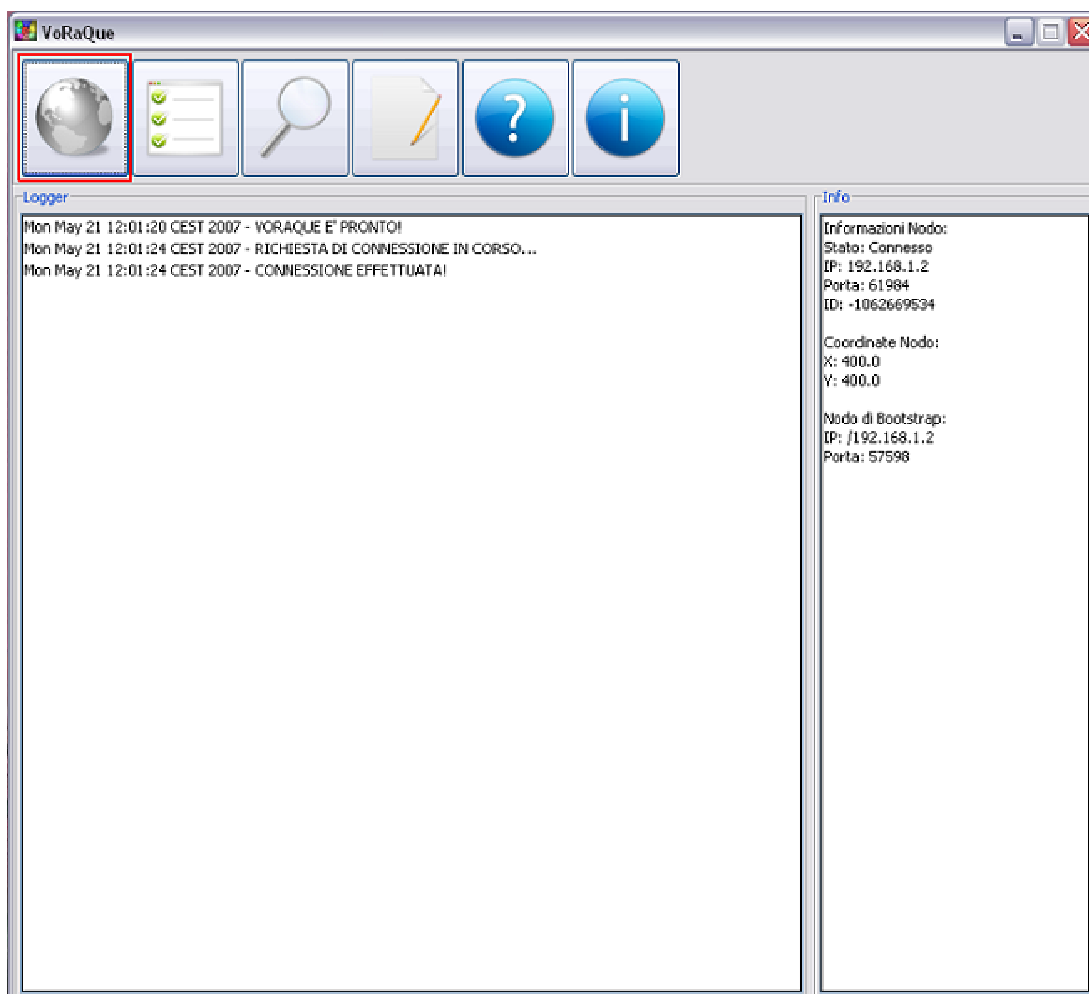


Figura 6.1: Schermata principale: il tasto evidenziato in rosso invia la richiesta di connessione e cancellazione dalla rete.

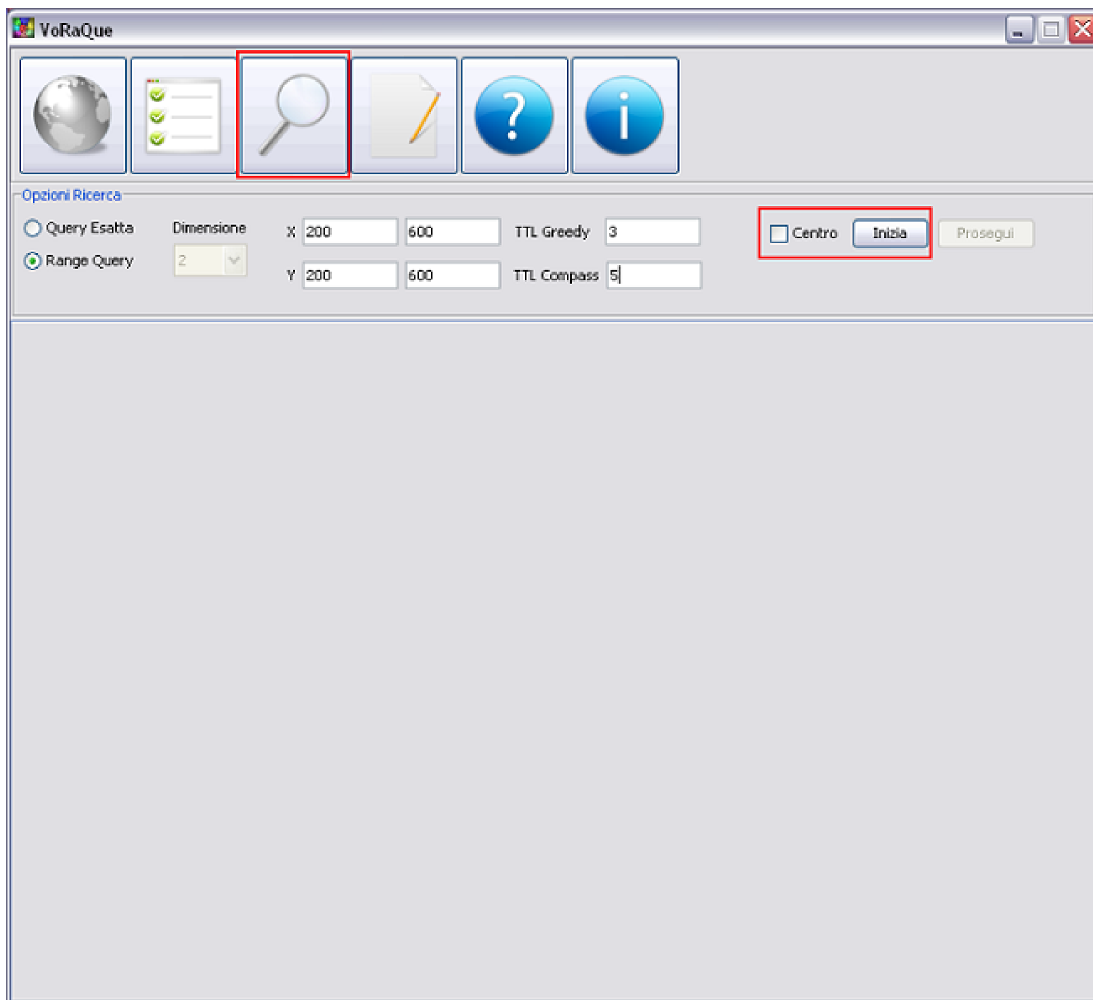


Figura 6.2: Schermata sottomissione query: per accedere a questa schermata occorre premere il bottone evidenziato in rosso.

L'interfaccia permette all'utente di selezionare la tipologia di ricerca che intende richiedere, che in questo caso è una range query. E' necessario inserire le coordinate dell'area di interesse della richiesta ed impostare il valore del Time-To-Live sia per il routing greedy che per il compass. Infatti, come abbiamo visto nel capitolo 4, la risoluzione di una range query comprende due fasi. Nella prima viene applicato l'algoritmo di routing greedy per raggiungere l'area di interesse, eseguito se il nodo che sottomette la query non si trova all'interno dell'area delimitata dalla query stessa. Nella seconda fase, viene applicato l'algoritmo di routing compass per recuperare in maniera efficiente tutti i possibili match per una query. Entrambi gli algoritmi di routing sono limitati da TTL e per rendere il protocollo il più adattabile possibile alle esigenze dell'utente, si è scelto di lasciare a quest'ultimo l'impostazione di tale limite. La seconda fase può iniziare in due momenti diversi:

1. non appena viene individuato un nodo la cui regione di Voronoi interseca per almeno un punto l'area di interesse della query.

2. Quando viene raggiunto il peer la cui regione di Voronoi contiene il punto centrale dell'area di interesse della richiesta.

Questa scelta viene lasciata all'utente tramite l'opzione centro (figura 6.2). La motivazione di questa funzionalità viene spiegata in dettaglio nel paragrafo 6.2.4. Inseriti tutti i valori la richiesta può essere sottomessa al sistema tramite il tasto **Inizia**. Successivamente, l'applicazione riceverà dalla rete le risposte che contengono tutti i match e di ognuno di essi verranno visualizzate le informazioni principali, ovvero indirizzo IP, porta e coordinate nello spazio degli attributi (figura 6.3).

Una funzionalità associata alla risoluzione delle range query è la possibilità di riprendere una ricerca da dove questa si è fermata se i risultati ottenuti non sono sufficienti per l'utente[2]. Per far questo basta premere il tasto **Prosegui** (figura 6.3).

[2]Tale funzionalità viene implementata con lo **stop bit** di cui abbiamo parlato nel capitolo 4.

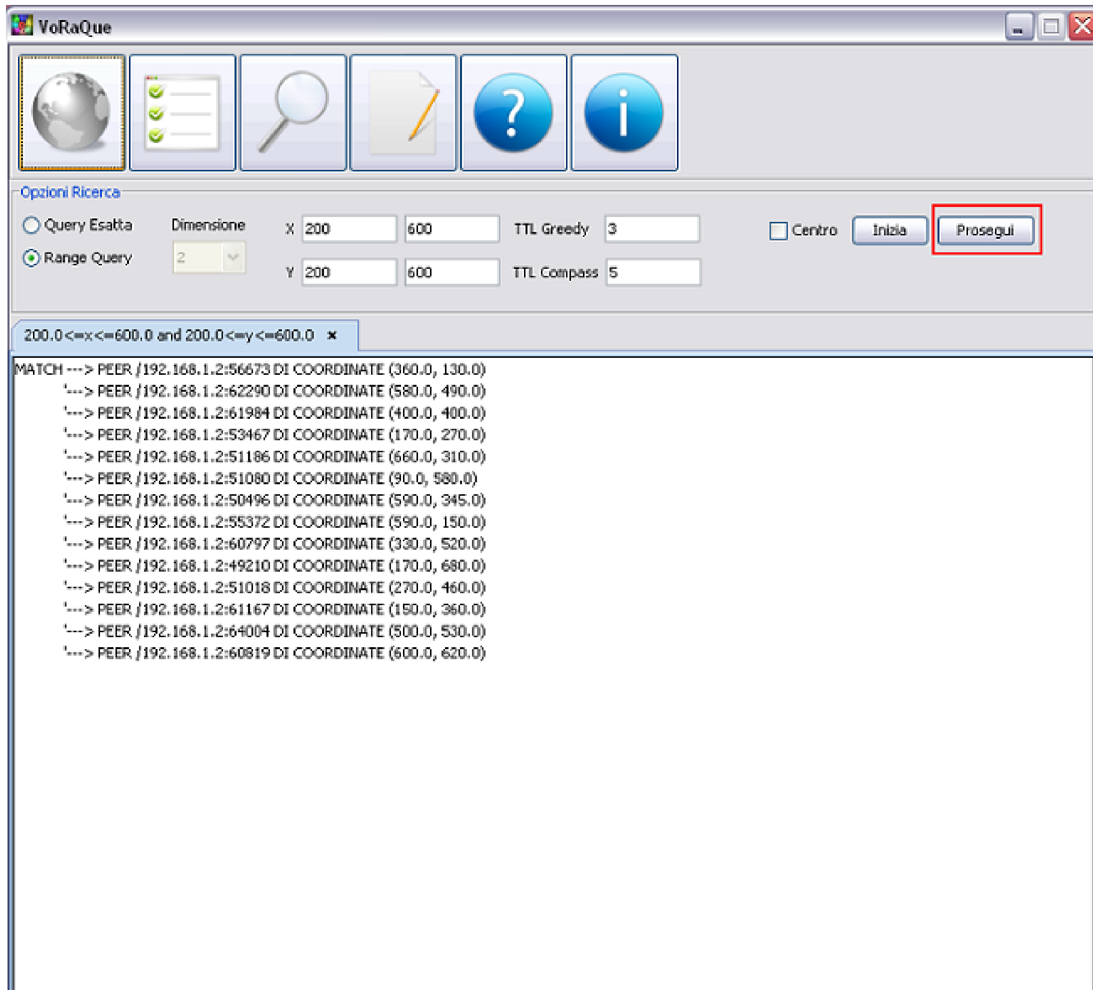


Figura 6.3: Lista dei risultati ottenuti.

6.2 Prove eseguite

In questo paragrafo vengono descritti gli esperimenti eseguiti ed i risultati ottenuti.

6.2.1 Ambiente di prova

La simulazione è stata effettuata in una piccola rete locale composta da due elaboratori aventi le seguenti caratteristiche:

Calcolatore 1 Processore: Pentium 4 con frequenza di clock pari a 3.06 GHz Capacità memoria principale: 512 MB Sistema Operativo: Windows XP Home Edition SP2

Calcolatore 2 Processore: Pentium 3 con frequenza di clock pari a 1,133 GHz Capacità di memoria principale: 384 MB Sistema Operativo: Windows XP Professional Edition.

I pc sono stati collegati tramite cavo LAN incrociato.

Le prove mirano a valutare l'efficienza del sistema nella risoluzione di una range query quando viene applicato il compass routing.

6.2.2 Valutazioni analitiche

Nei primi test eseguiti si è misurato il numero medio di messaggi spediti da ogni nodo:

$$NumMSG = \frac{\sum_{i=1}^{N} MessaggiSpediti(i)}{N} \tag{6.1}$$

dove N rappresenta il numero dei peer coinvolti nella risoluzione di una range query.

Il numero dei messaggi spediti comprende:

1. n richieste di vicini di Voronoi (**VoronoiNeighMSG**).
2. m risposte a m richieste di vicini di Voronoi (**ReplyVoronoiNeighMSG**).

3. r messaggi relativi alla range query inoltrati ai figli (**RangeMSG**) oppure spedizione di una risposta (**ReplyRangeMSG**) direttamente al nodo richiedente.

Il protocollo per la risoluzione di una range query prevede che, la prima volta che viene ricevuto un messaggio di tipo RangeMSG, il nodo P invii la richiesta ai suoi vicini di Voronoi per conoscere i relativi VoronoiNeighbours. Tale richiesta viene inviata a tutti tranne al nodo che ha inoltrato il pacchetto RangeMSG a P. In media vengono spedite n richieste di tipo **VoronoiNeighMSG**, dove

$$n = VoronoiNeighbours(P) - 1 \tag{6.2}$$

D'altra parte, il nodo P invia il pacchetto di **ReplyVoronoiNeighMSG** a tutti i suoi vicini tranne che ai suoi figli nello spanning tree. In altre parole, viene applicato il criterio sopra descritto, ovvero i figli di P non spediscono al padre la richiesta dei vicini di Voronoi e di conseguenza P non deve costruire nessuna risposta. In media vengono spedite m richieste di tipo **ReplyVoronoiNeighMSG**, dove

$$m = VoronoiNeighbours(P) - Son(P) \tag{6.3}$$

e $Son(P)$ rappresenta il numero medio di figli nello spanning tree per il nodo P. Al termine, individuati i figli nell'albero, vengono spedite r copie di RangeMSG, dove

$$r = Son(P). \tag{6.4}$$

La letteratura ricorda che nello spazio bi-dimensionale *il numero medio* di VoronoiNeighbours per una regione è pari a 6 [12, 50].

Osservazione 1. *m + r = VoronoiNeighbours(P) = 6.*

Dimostrazione.
$m = VoronoiNeighbours(P) - Son(P)$ e $r = Son(P)$
$m = VoronoiNeighbours(P) - r$

$m + r = VoronoiNeighbours(P)$

□

Osservazione 2. *Se un nodo F non ha figli, ovvero è una foglia per lo spanning tree, m = VoronoiNeighbours(F) = 6.*

Dimostrazione. Come descritto sopra, un nodo V invia il pacchetto di ReplyVoronoi-NeighMSG a tutti i suoi vicini tranne che ai suoi figli nello spanning tree. Poiché V è una foglia, per definizione non ha figli. Essendo $m = VoronoiNeighbours(V) - Son(V)$ e $Son(V) = 0$, otteniamo $m = VoronoiNeighbours(V) = 6$.

□

Osservazione 3. *In assenza di cache, in media NumMSG è limitato superiormente da* $2\,VoronoiNeighbours - 1$.

Dimostrazione.
$NumMSG = n + m + r$
$NumMSG = VoronoiNeighbours - 1 + VoronoiNeighbours - Son + Son$
$NumMSG = 2\,VoronoiNeighbours - 1$

□

VoRaQue utilizza tecniche di caching per memorizzare i vicini di Voronoi dei suoi vicini di Voronoi e in questo modo un peer evita di inviare richieste di tipo VoronoiNeighMSG per ogni messaggio di range query giunto dalla rete.

Osservazione 4. *In presenza di cache, in media NumMSG è limitato superiormente da r = VoronoiNeighbours - 1.*

Dimostrazione. $NumMSG = n + m + r$, ma in presenza di cache n e m sono pari a zero. Inoltre, nel caso ottimo, tutti i vicini di un nodo P, tranne il peer che gli ha spedito il messaggio di RangeMSG, sono suoi figli nello spanning tree. In questo caso, avremo che il numero di richieste di range query da inoltrare è pari a $VoronoiNeighbours(P) - 1$.

□

6.2.3 Valutazione del traffico

E' stato impostato un intervallo di tempo T di validità delle informazioni presenti in cache[3]. E' stata configurata un'overlay network equamente distribuita tra i due host e successivamente è stata eseguita l'applicazione con l'interfaccia grafica sul calcolatore 1. Per eseguire questo esperimento, ogni istanza di VoRaQue restituisce il numero di messaggi spediti nella risoluzione di una query dopo un periodo di tempo t.

Richiesta al sistema la risoluzione della query $2 \leq x, y \leq 800$, al termine dell'intervallo t sono stati raccolti i valori restituiti da ogni peer. Ogni nodo nella propria cache ha le informazioni relative ai vicini di Voronoi dei confinanti, quindi il calcolo del compass routing viene eseguito in locale. E' stata nuovamente richiesta la risoluzione della query $2 \leq x, y \leq 800$ e al termine del periodo t sono stati raccolti nuovamente i dati. Questo esperimento è stato ripetuto per verificare la correttezza dei risultati. In questo caso la query descriveva sempre la stessa area di interesse, ma veniva variata la dimensione della rete (22, 35, 50, 85, 100 nodi). Di seguito sono mostrati i grafici relativi alla prima misurazione. In essi è riportata anche la varianza, calcolata come:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{M}(x_i - \mu)^2}{M - 1}} \tag{6.5}$$

dove M rappresenta il numero di nodi coinvolti nella risoluzione della query, x_i è il numero medio di messaggi spediti all'i-esimo peer e μ è il numero medio di messaggi spediti.

Il grafico di figura 6.4 mostra che il numero medio di messaggi spediti da ogni nodo aumenta con la dimensione della rete. Una caratteristica fondamentale dei diagrammi di Voronoi è che il numero medio di confinanti per una regione è pari a 6 [12, 50] e come osservato sopra, il numero medio di messaggi spediti da ogni nodo è limitato superiormente da $2VoronoiNeighbours - 1$, che è uguale a 11.

[3] Con il termine cache intendiamo la struttura dati **VoronoiNeighNeigh** descritta nel capitolo precedente. La sua funzione è quella di memorizzare i vicini di Voronoi dei peer confinanti del nodo corrente per velocizzare la risoluzione delle range query.

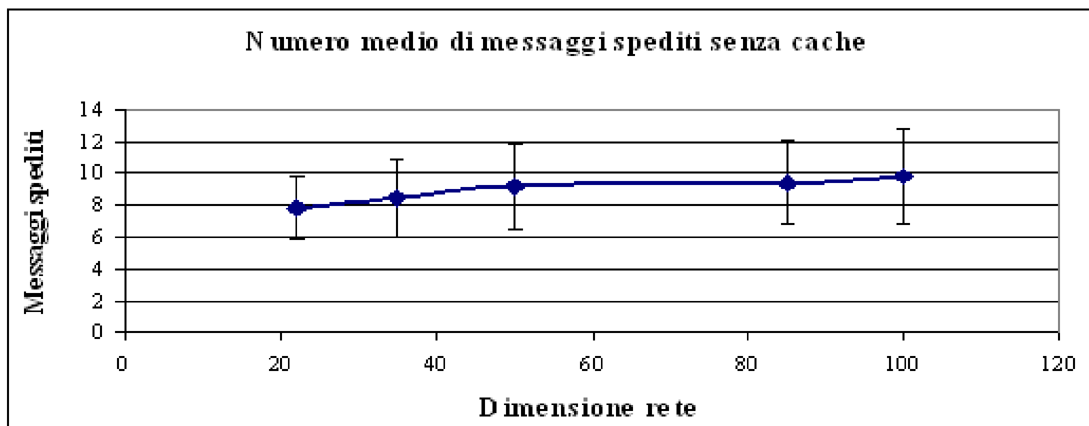


Figura 6.4: Numero medio di messaggi spediti da ogni nodo in assenza di informazioni in cache.

La situazione descritta in figura 6.5 rappresenta il caso in cui il nodo che riceve una range query può stabilire la relazione padre-figlio tra i nodi dello spanning tree con *informazioni locali* già *presenti in cache*. In questo caso, il messaggio spedito può essere di due tipi:

1. messaggio di reply inviato direttamente al nodo richiedente.

2. Messaggio di RangeMSG che viene inoltrato a uno o più nodi vicini.

Il numero medio di messaggi spediti da ogni nodo tende a $1,5$.

6.2.4 Valutazione del numero di hop

Il test descritto in questo paragrafo utilizza il medesimo ambiente di prova, ma in questo caso è stata misurata la **latenza** in termini di numero di hop necessari per recuperare tutti i possibili match di una range query. In questo caso non abbiamo fatto uso di caching, ed è stata configurata un'overlay network che è stata equamente distribuita tra i due calcolatori. Successivamente è stata eseguita l'applicazione con l'interfaccia grafica sul calcolatore 1. La dimensione della rete è fissa, mentre varia la superficie

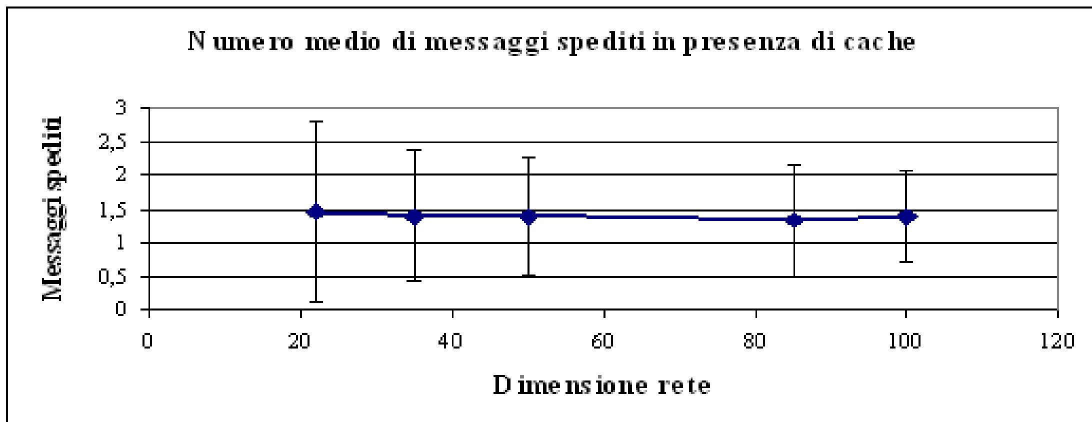


Figura 6.5: Numero medio di messaggi spediti da ogni nodo quando la cache contiene informazioni significative.

dell'area di interesse definita dalla query. Sono state sottomesse in sequenza le seguenti richieste:

- $200 \leq x, y \leq 600$
- $150 \leq x, y \leq 650$
- $100 \leq x, y \leq 700$
- $50 \leq x, y \leq 750$
- $2 \leq x, y \leq 800$.

Il campo **TTLcompass** (vedere figura 6.2) è stato impostato con il valore 1. In questo modo è stato possibile reperire facilmente il numero di nodi recuperati ad ogni passo del compass routing. Iniziata la ricerca, si è provveduto a premere il tasto **Prosegui** finché è stato possibile trovare dei match. L'esperimento è stato eseguito in due modalità:

1. la sequenza delle query è stata richiesta dal nodo di coordinate $(400, 400)$ che si trova nel centro dell'area di interesse.

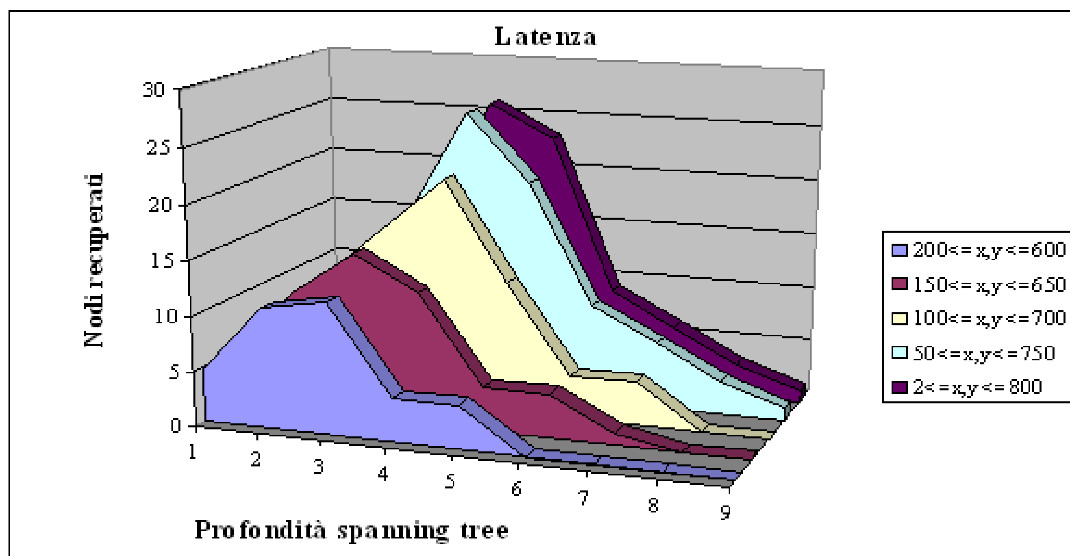


Figura 6.6: Latenza nel caso che il nodo che genera le query si trova al centro dell'area di interesse.

2. La sequenza delle query è stata richiesta dal nodo di coordinate $(184, 52)$ che si trova prima al di fuori, poi su un lato dell'AOI.

L'esperimento è stato ripetuto due volte, per entrambe le modalità, per verificare l'attendibilità dei dati registrati.

Come possiamo vedere nel grafico di figura 6.6, il numero di hop necessari per recuperare tutti i possibili match aumenta con l'aumentare dell'area di interesse della query. Inoltre, è interessante osservare l'andamento a 'campana' della latenza, ovvero, la quantità di nodi recuperati aumenta fino a raggiungere un massimo, dopo di che diminuisce.

La figura 6.7 mostra una rappresentazione grafica dell'esecuzione di una range query. Finché non viene raggiunto l'anello indicato con **Massimo**, il numero di nodi recuperati tende ad aumentare, dopodichè tale quantità tende a diminuire, preché i peer che sono sui bordi non possono inoltrare la query e costruiscono il messaggio di reply, con la conseguente diminuzione sia di direzioni in cui si muove il compass routing, sia

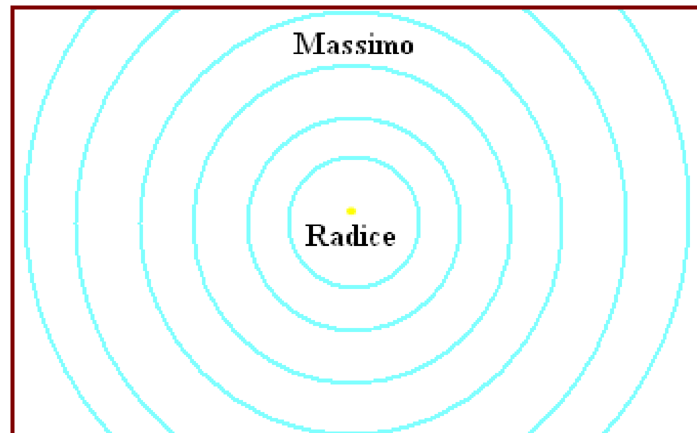


Figura 6.7: Andamento del compass routing ad ogni hop, il rettangolo rappresenta l'area di interesse della query richiesta.

di match recuperati.

Nel grafico successivo (figura 6.8) viene mostrata la latenza nel caso in cui il nodo che genera la richiesta abbia coordinate $(184, 52)$.

Poiché il nodo che sottomette la richiesta al sistema si trova ad un estremo dell'area di interesse della query, per recuperare tutti i possibili match occorrono più hop. Indipendentemente dalla posizione della radice dello spanning tree, la latenza ha sempre un andamento a 'campana'.

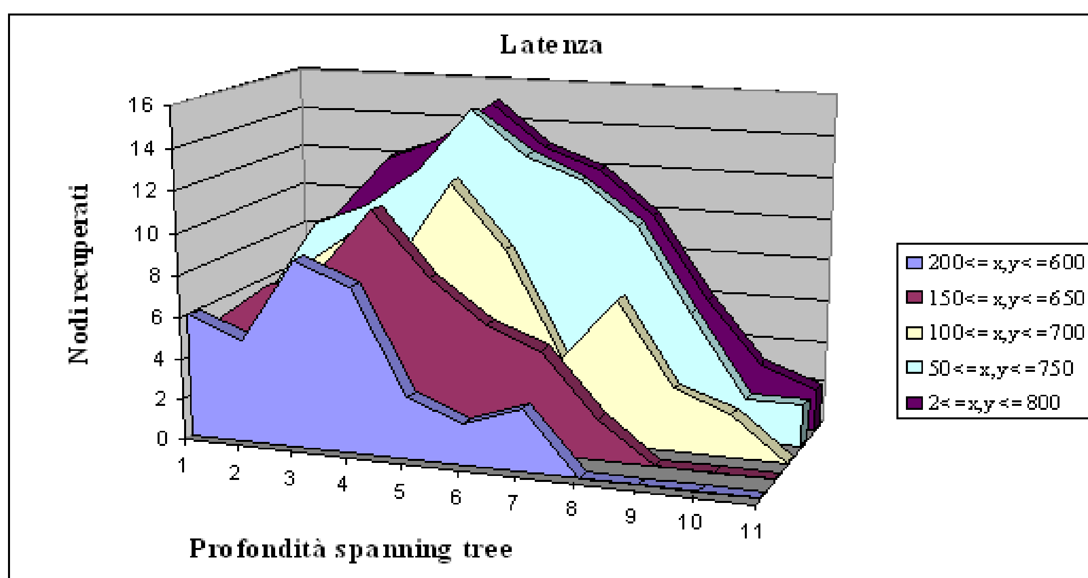


Figura 6.8: Latenza nel caso che il nodo che genera le query si trova in un punto qualsiasi dell'area di interesse.

6.2.5 Valutazione della latenza

L'ultimo test è stato eseguito sulle macchine del Centro di Calcolo del Dipartimento di Informatica dell'Università di Pisa.

In questo esperimento, abbiamo voluto valutare la **latenza** misurata in millisecondi necessari per risolvere una range query in assenza e in presenza di cache[4]. E' stato impostato un intervallo di tempo T di validità delle informazioni presenti in cache ed è stata configurata una rete di 21[5] nodi, uno per ogni macchina del Centro di Calcolo. La dimensione della rete è fissa, mentre varia la superficie dell'area di interesse definita dalla query. Sono state sottomesse in sequenza le seguenti richieste:

- $200 \leq x, y \leq 600$

[4]Con il termine cache intendiamo la struttura dati **VoronoiNeighNeigh** descritta nel capitolo precedente. La sua funzione è quella di memorizzare i vicini di Voronoi dei peer confinanti del nodo corrente.

[5]Un nodo è il bootstrap.

- $150 \leq x, y \leq 650$
- $100 \leq x, y \leq 700$
- $50 \leq x, y \leq 750$.

Questo esperimento è stato ripetuto più volte per verificare la correttezza dei risultati. Nei grafici successivi viene riportata anche la varianza.

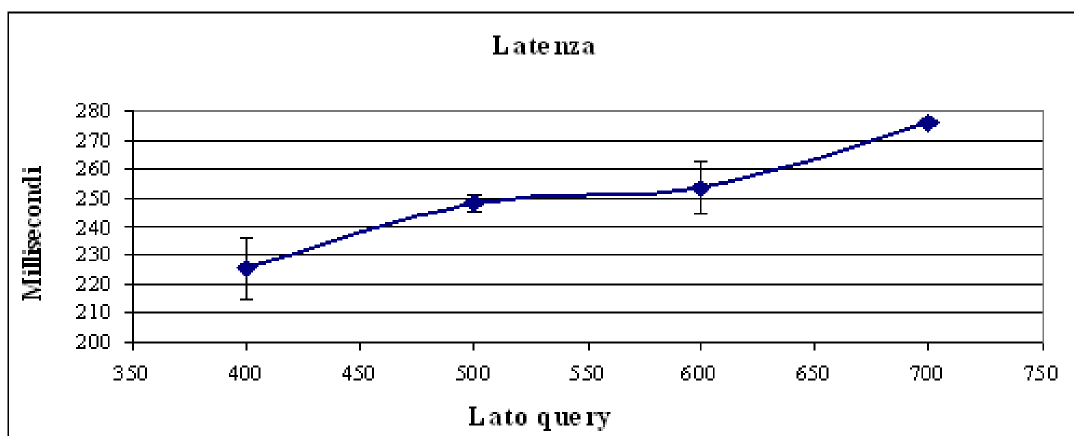


Figura 6.9: Latenza in assenza di informazioni in cache.

Come possiamo vedere nel grafico di figura 6.9, il tempo necessario per risolvere una range query aumenta con la dimensione della superficie descritta dalla richiesta. Di seguito viene mostrato il grafico relativo alla latenza in presenza di cache.

Nel grafico di figura 6.10 otteniamo lo stesso andamento descritto nel grafico precedente: il tempo necessario per risolvere una range query è dipendente dal lato della query. In questo caso, otteniamo dei tempi migliori perchè la relazione padre-figlio definita nel compass routing viene verificata da ogni nodo utilizzando solo le informazioni in cache.

Le prove elencate in questo capitolo dimostrano un'elevata efficienza del protocollo per la risoluzione delle range query implementato in VoRaQue.

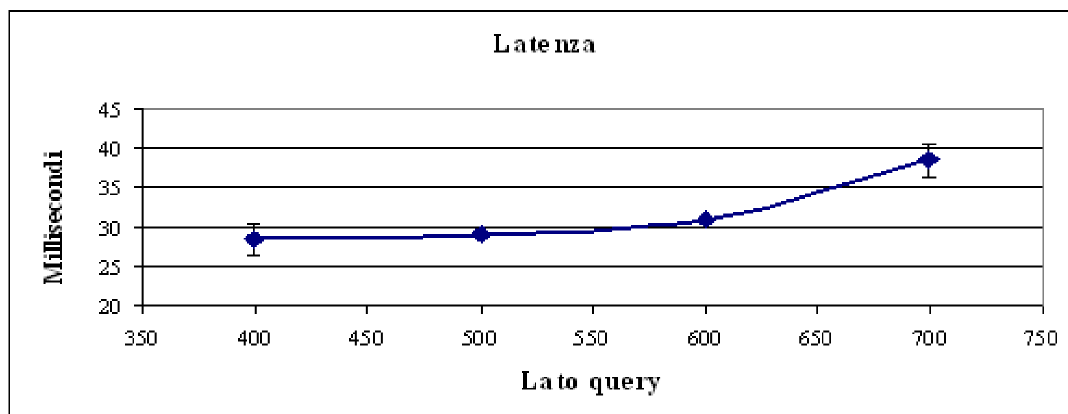


Figura 6.10: Latenza con informazioni contenute in cache.

6.3 Perdita match

Durante la fase di testing per misurare la latenza del sistema si sono verificati dei risultati inattesi. Osserviamo la seguente tabella:

Query sottomesse	Nodi recuperati $(400, 400)$	Nodi recuperati $(184, 52)$
$200 \leq x, y \leq 600$	36	36
$150 \leq x, y \leq 650$	52	52
$100 \leq x, y \leq 700$	73	66
$50 \leq x, y \leq 750$	97	96
$2 \leq x, y \leq 800$	100	100

Come possiamo vedere, quando viene risolta la query $100 \leq x, y \leq 700$ il numero dei nodi recuperati diminuisce in maniera non trascurabile nel caso che la richiesta venga generata dal nodo di coordinate $(184, 52)$. Per capire la mancanza di alcuni match occorre osservare lo spanning tree che viene costruito nei due casi.

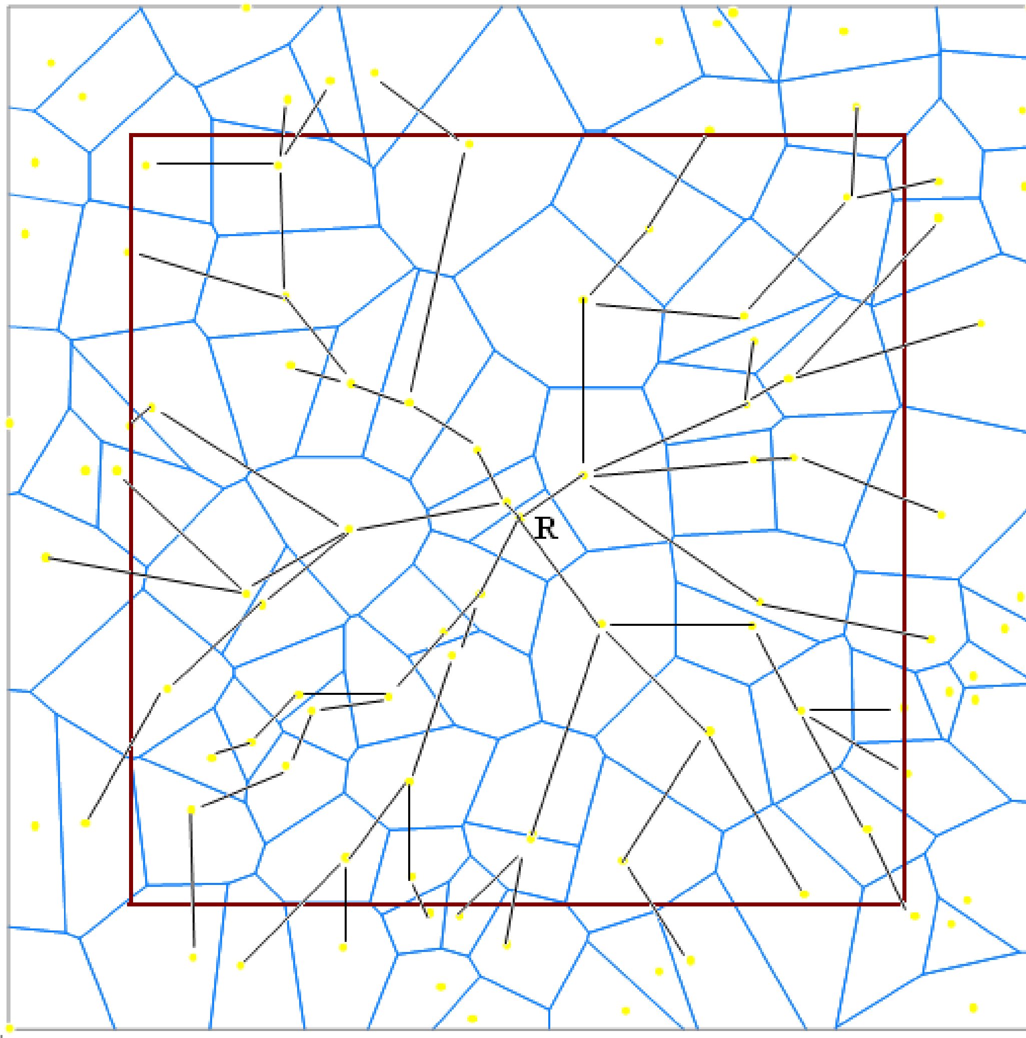


Figura 6.11: Spanning tree per la query $100 \leq x, y \leq 700$ con radice (400,400).

In figura 6.11[6] possiamo vedere che vengono raggiunti tutti match per la query $100 \leq x, y \leq 700$ generata dal nodo (400, 400). Ma osserviamo la figura 6.12.

Le regioni di Voronoi evidenziate in giallo sono i match che non vengono recuperati. Ciò accade perché il padre del nodo a è b, ma b non è un match per la range query e non riceverà mai il messaggio con la richiesta e non la inoltrerà mai ad a. Come conseguenza vengono persi 7 potenziali match, poichè essi vengono raggiunti tramite a.

Lo spanning tree mostrato in figura 6.13, dimostra che i nodi che non vengono recuperati nella query precedente adesso vengono raggiunti in quanto b è un match e riceve il messaggio di range query. Anche in questo caso però si ha lo stesso problema, perché un potenziale risultato viene perso (regione di Voronoi in giallo).

Da quanto detto risulta che in generale è meglio scegliere il punto centrale della range query prima di iniziare la fase di compass routing, infatti in questo modo vengono recuperati tutti i risultati.

[6] Il diagramma di Voronoi utilizzato in questa figura e nelle successive, rappresenta la topologia completa della rete di overlay. Ogni nodo mantiene un proprio diagramma del quale fanno parte i suoi VoronoiNeighbours e CloseNeighbours, ma la sua conoscenza della topologia generale della rete è solo parziale, quindi in realtà i confini dei suoi vicini possono essere diversi rispetto alla visione globale del sistema.

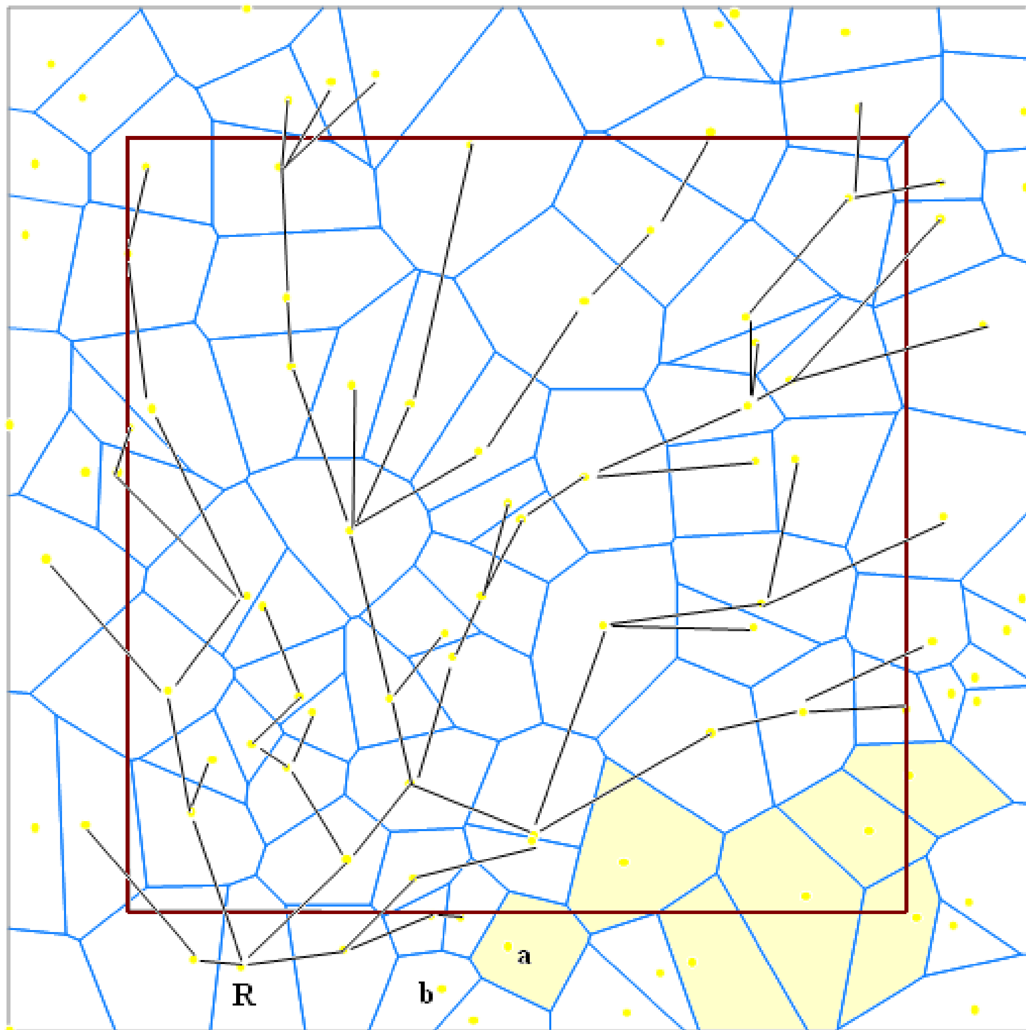


Figura 6.12: Spanning tree per la query $100 \leq x, y \leq 700$ **con radice (184,52).**

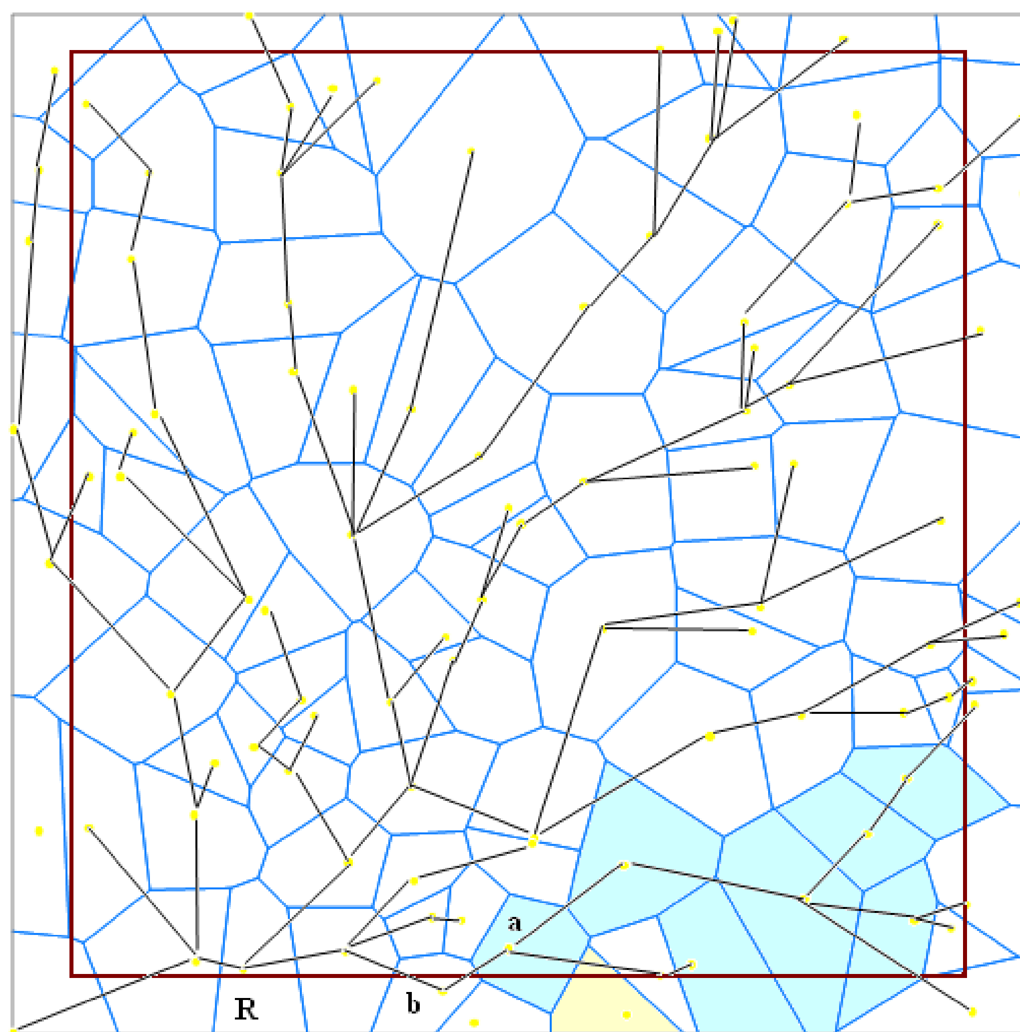


Figura 6.13: Spanning tree per la query $50 \leq x, y \leq 750$ **con radice (184,52).**

6.3.1 Modifiche proposte per il compass routing

Il problema descritto nel paragrafo precedente può essere risolto utilizzando due tecniche diverse. La prima soluzione è una modifica del compass routing, mentre la seconda soluzione utilizza il compass routing insieme ad un routing flooding lungo i lati dell'area di interesse della query.

Modifica algoritmica del compass routing

Questa prima soluzione prevede che, quando un nodo P riceve una richiesta di vicini di Voronoi tramite il messaggio di VoronoiNeighMSG, restituisca al richiedente la lista dei peer confinanti formata soltanto da quei nodi la cui regione di Voronoi interseca per almeno un punto l'area di interesse definita dalla query. Osserviamo il particolare di figura 6.14.

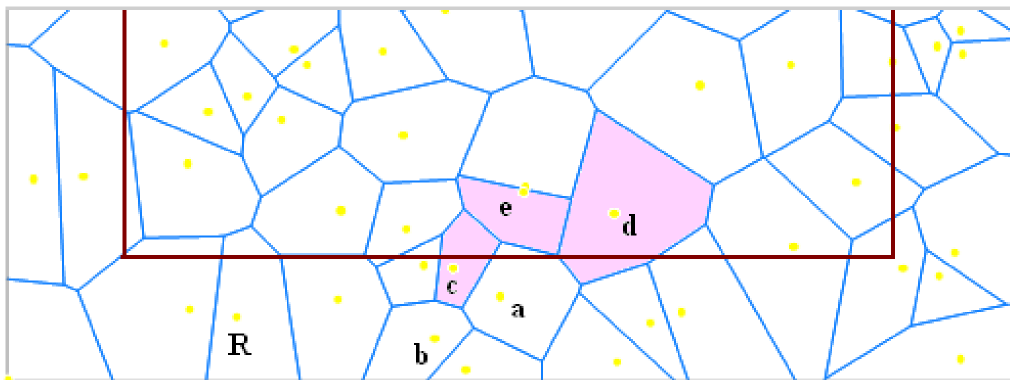


Figura 6.14: Particolare della query $100 \leq x, y \leq 700$ **con radice R di coordinate (184,52).**

Quando il nodo c riceve la richiesta di RangeMSG, invia il pacchetto di VoronoiNeighMSG ad a per ricevere la sua lista di vicini di Voronoi la cui regione interseca per almeno un punto l'area di interesse della query (regioni colorate in rosa). Ricevuta tale lista, c applica le regole definite dal compass routing per individuare i suoi figli nello spanning tree. Grazie a questa modifica, c diventa il padre di a. Il compass routing non

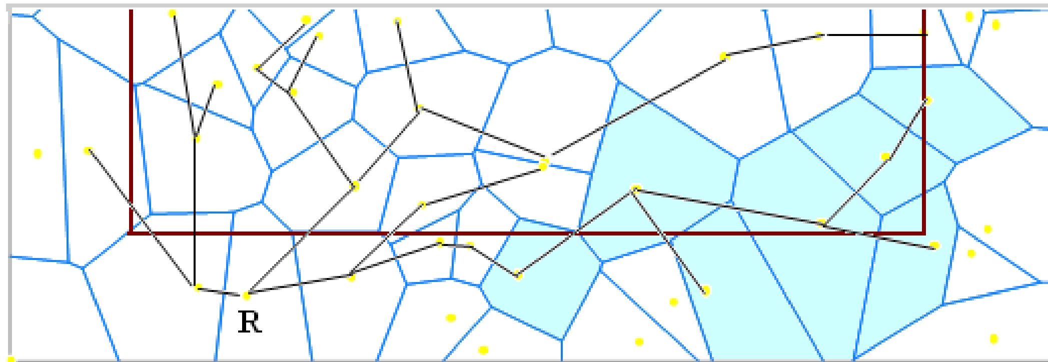


Figura 6.15: Particolare relativo alla risoluzione della query $100 \leq x, y \leq 700$ **generata dal nodo di coordinate (184,52).**

tenterà di inoltrare la query verso un nodo esterno all'area di interesse (b) e saranno raggiunti tutti i peer che nella prima implementazione venivano persi. Il risultato è visibile in figura 6.15.

Flooding sui bordi dell'area di interesse

La seconda soluzione prevede che, se un nodo P viene attraversato da un lato dell'area descritta dalla query e riceve il messaggio di RangeMSG, recuperi i vicini confinanti che condividono il lato della regione che viene tagliato dal lato dell'area di interesse. Ottenuti tali vicini, P inoltra ad ognuno di loro la richiesta di range query senza calcolare se sono loro figli nello spanning tree. Mentre per i VoronoiNeighbours, la cui regione è completamente contenuta in AOI, viene eseguito ancora il compass routing.

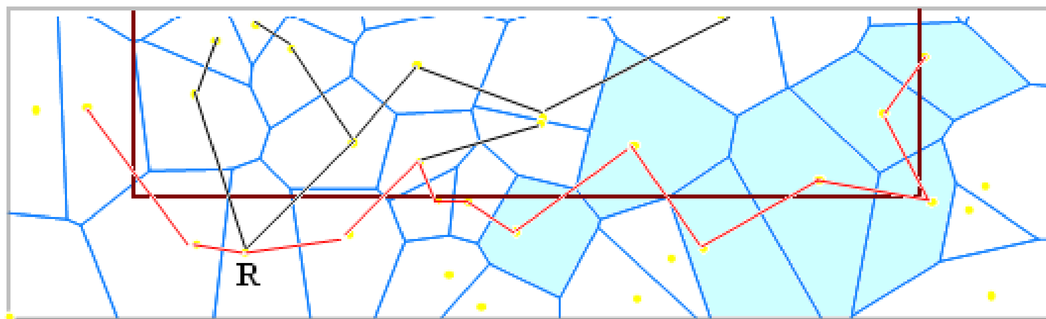


Figura 6.16: Particolare relativo alla risoluzione della query $100 \leq x, y \leq 700$ **generata dal nodo di coordinate (184,52).**

Come possiamo vedere in figura 6.16, utilizzando questa modifica al protocollo riusciamo a raggiungere quei nodi che altrimenti verrebbero persi (figura 6.12). Le linee rosse disegnano un cammino tra i peer che giacciono sul bordo dell'area di interesse della query, mentre le linee nere rappresentano lo spanning tree che viene costruito adottando la tecnica del compass routing.

Conclusioni

L'obiettivo di questa tesi è stato lo studio e l'implementazione in Java di un protocollo per la risoluzione efficiente di range query sulle reti di Voronoi.

Il supporto realizzato prende il nome di VoRaQue e si ispira alle ricerche che hanno portato alla definizione di VoroNet [12]. L'aspetto innovativo che introduce VoRaQue è il protocollo per la risoluzione delle range query che viene realizzato sfruttando l'algoritmo di compass routing che costruisce uno spanning tree per eseguire un multicast efficiente sulla triangolazione di Delaunay.

Caso di Studio: GIS

Come abbiamo visto, VoRaQue è un sistema P2P nato con l'obbiettivo di risolvere in maniera efficiente le range query. Da notare che fino ad ora abbiamo parlato di spazio bi-dimensionale, ma tale sistema può essere esteso anche in più dimensioni. Per questo motivo un suo utilizzo potrebbe essere come supporto ad un **Grid Information Service**. In questo caso ogni coordinata dello spazio di VoRaQue rappresenterebbe un attributo della risorsa computazionale che viene messa a disposizione da un host.

Un modo per estendere VoRaQue ad uno spazio in più dimensioni può avvenire con l'interazione con un tool di nome **Qhull** [61], che calcola i Diagrammi di Voronoi fino a $8D$. In altre parole, il package **lowerlevel** invece di interagire con Nomade [50] andrebbe ad interagire con Qhull, pur rimanendo valido il protocollo definito perché è stato pensato per essere facilmente estendibile in più dimensioni.

Il costo computazionale di Qhull è di $O(n \log v)$ fino a $3D$, dove n è il numero di punti per i quali viene costruito il diagramma, e v è il numero di vertici in output. A partire dalla quarta dimensione, la sua performance diventa $O(nf_v/v)$, dove f_v è una funzione che cresce rapidamente perché dipende dal numero di facce che vengono calcolate [62].

Nel caso di risorse computazionali, è ragionevole pensare uno spazio degli attributi di al massimo 6 dimensioni, a cui corrisponderebbe comunque un costo computazionale non trascurabile. In merito a ciò, è importante ricordare che, anche se tale costo è elevato, ogni nodo che fa parte della rete di VoRaQue calcola un diagramma di Voronoi con un numero piccolo e limitato di nodi rispetto a quelli realmente connessi. Ne possiamo dedurre che tale costo non è drammatico ai fini dell'applicazione. Comunque sia, un'alternativa valida può essere l'uso di due reti di VoRaQue separate, entrambe in $3D$, ottenendo come vantaggio il costo logaritmico di Qhull. D'altra parte, avremmo un maggiore overhead di comunicazione per la risoluzione delle range query. Un esempio di sistema P2P che utilizza più reti per condividere le risorse è la soluzione multi-DHT proposta in [6].

Da notare che le problematiche relative al costo computazionale di Qhull si ripercuotono nella procedura di inserimento e cancellazione di un nodo dalla rete, infatti i vicini che percepiscono il cambiamento devono ricalcolare il diagramma di Voronoi. Ciò non va ad influire sulle prestazione del protocollo per la risoluzione delle range query definito in VoRaQue nel quale vengono descritte delle tecniche per recuperare in maniera efficiente tutti i match di una richiesta indipendentemente dal numero di dimensioni che vengono utilizzate.

Descriviamo ora gli sviluppi futuri previsti per VoRaQue.

Come abbiamo visto nel capitolo 4, la versione attuale di VoRaQue non utilizza i **Long Range Neighbours**, che sono i vicini remoti definiti in VoroNet per rispettare le caratteristiche del piccolo mondo di Kleinberg [13, 14]. In merito a ciò è importante ricordare che i Voronoi Neighbours, i Close Neighbours e i Long Range Neighbours

garantiscono un **routing poli-logaritmico** nel numero dei nodi presenti nel sistema.

Affiché VoRaQue possa utilizzare i **Long Range Neighbours** è necessario introdurre un altro insieme di vicini che prende il nome di **Back Long Range Neighbours**. Sia i Voronoi Neighbours che i Close Neighbours formano connessioni di natura simmetrica, ma ciò non vale per i Long range neighbours. Consideriamo il peer p e il nodo t appartenente a **Long Range Neighbours(p)** e supponiamo che t lasci la rete. In questo caso, t non è in grado di contattare p per notificare la modifica della topologia e l'identità del nuovo **Long Range Neighbours**. Per superare questo problema, p diventa un vicino di t, e prende il nome di **Back Long Range Neighbours**. Quest'ultimo non verrà utilizzato per il routing, ma solo per mantenere consistente la topologia della rete.

Inoltre, la gestione dei **Long Range Neighbours** comporta delle modifiche nell'operazione di inserimento di un nodo nell'overlay network perchè durante questa procedura avviene la scoperta di tutti i vicini.

Un ulteriore sviluppo futuro è lo studio più approfondito delle soluzioni proposte nel capitolo 6 per individuare la tecnica che massimizza il numero di risultati restituiti anche in presenza di perdita di match in prossimità dei bordi dell'area di interesse, pur mantenendo le performance dimostrate dal compass routing.

Infine, uno dei principali obiettivi è di introdurre VoRaQue nel contesto di Grid Information Service dove ogni nodo partecipante alla rete mette a disposizione le proprie risorse computazionali. In questo caso, la risoluzione di una range query permette di individuare un certo numero di host con caratteristiche fisiche richieste in modo da poterli sfruttare nell'ambito delle griglie computazionali.

Bibliografia

[1] **Christina Schmidt e Manish Parashar**: *Flexible Information Discovery in Decentralized Distributed Systems*. Proc. 12th Int. Symp. on High-Performance Distributed Computing (HPDC-12 2003), pp 226-235, 2003.

[2] **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, December 2004.

[3] **Paolo Trunfio, Domenico Talia, Paraskevi Fragopoulou, Charis Papadakis, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov e Seif Haridi**: *Peer-to-Peer Models for Resource Discovery on Grids*. In Proc. of the 2nd CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, 2006.

[4] **Ian Foster e Adriana Iamnitchi**: *On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing*. Proc Second Int. Workshop on Peer-to-Peer Systems (IPTPS 2003), pp 118-128, 2003.

[5] **Diego Puppin, Stefano Moncelli, Ranieri Baraglia, Nicola Tonellotto e Fabrizio Silvestri**: *A Grid Information Service based on Peer-to-Peer*. Proc. 11th Euro-Par Conf. (Euro-Par 2005), LNCS, vol. 3648, pp 454-464, Sprinter 2005.

[6] **Domenico Talia, Paolo Trunfio, Jingdi Zeng e Mikael Hogqvist**: *A DHT-based Peer-to-Peer Framework for Resource Discovery in Grids*. Tech. Rep. TR-

0048, Institute on Knowledge and Data Management and Institute on System Architecture, CoreGRID Network of Excellence, June 2006.

[7] **Moreno Marzolla, Matteo Mordacchini e Salvatore Orlando**: *Peer-to-Peer Systems for Discovering Resources in a Dynamic Grid*. In Proc. of the 2nd CoreGRID Workshop on Grid and Peer to Peer Systems Architecture, Paris, France, January 2006.

[8] **Sameh El-Ansa-ry, Luc Onana Alima, Per Brand e Seif Haridi**: *Efficient Broadcast in Structured P2P Networks*. Proc. Of IEEE/ACM Int. Symp. on Cluster Computing and the Grid (CCGRID'05), Cardiff, UK, 2005.

[9] **Arturo Crespo e Hector Garcia-Molina**: *Routing indices for peer-to-peer systems*. In Proceedings of 22nd Int. Conf. on Distributed Computing Systems (ICDCS'02), pp 23-30, 2002.

[10] **Artur Andrzejak e Zhichen Xu**: *Scalable, Efficient Range Queries for Grid Information Services*. Proc. Of 2nd IEEE Int. Conf. Peer-to-Peer Computing (P2P'02), Sweden, September 2002.

[11] **Adeep Cheema, Moosa Muhammad ed Indranil Gupta**: *Peer-to-Peer Discovery for Computantional Resources for Grid Applications*. Proc. IEEE/ACM Workshop on Grid Computing (GRID), 2005.

[12] **Oliver Beaumont, Anne-Marie Kermarrec, Loris Marchal e Etienne Rivière**: *Voronet: a scalable object network based on Voronoi tessellations*. INRIA Research Report, INRIA, France, February 2006.

[13] **John Kleinberg**: *The Small-World Phenomenon: An algorithmic Perspective*. In Proc. 32nd ACM Symposium on Theory of Computing, 2000.

[14] **John Kleinberg**: *Navigation in a Small World*. Nature, 406, 2000.

[15] **Napster Home Page**, 1999.
http://napster.com.

[16] **Clip2**: *The Gnutella Protocol Specification v0.4 (Document Revision 1.2)*.
http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, 2001.

[17] **Tor Klingberg e Raphael Manfredi**: *Gnutella 0.6*.
http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, 2002.

[18] **Gnutella Home Page**.
http://www.gnutella.org.

[19] **Overlay Network**. http://en.wikipedia.org/wiki/Overlay_network.

[20] **Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek ed Hari Balakrishnan**: *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp 149-160. ACM Press, 2001.

[21] **Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp e Scott Shenker**: *A Scalable Content-Addressable Network*. In Proceedings of ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp 161-172, 2001.

[22] **Antony Rowstron e Peter Druschel**: *Pastry: Scalable, Decentralized Object Location and Routing for Large Scale Peer-to-Peer Systems*. Proc. IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware 2001), pp 329-350, 2001, LNCS, vol 2218, Springer 2001.

[23] **Ian Foster e Carl Kesselman**: *The Grid 2: Blueprint for a New Computing Infrastructure*. Second Edition, Morgan Kaufmann, 2004.

[24] **The Globus Alliance**.
http://www.globus.org.

[25] **LCG Home Page**.
http://lcg.web.cern.ch/LCG/index.html.

[26] **CERN Home Page**.
http://public.web.cern.ch/Public/Welcome.html.

[27] **CERN**.
http://h40047.www4.hp.com/references_global/detail.php?id=6816§ion_id=1460&cc=it&ll=it&scat=0.

[28] **Elisabetta Durante**: *Italia ed Europa locomotive della grande Grid*. Il Sole 24 Ore, Giugno 2005.
http://www.infn.it/wyp2005/ilsole24ore/locomotive.htm.

[29] **SETI Home Page**.
http://www.seti.org/.

[30] **MMOG**.
http://it.wikipedia.org/wiki/Massively_multiplayer_online_game.

[31] **Shun-Yun Hu, Jui-Fa Chen e Tsu-Han Chen**: *VON: A Scalable Peer-to-Peer Network for Virtual Environments*. IEEE Network, vol. 20, no. 4, Jul./Aug. 2006.

[32] **Lorenzo Zaninetti**: *Dynamical Voronoi Tessellation. ii - The three-dimensional case*. In the journal of Astronomy and Astrophysics, vol 233, pp 293-300, 1990.

[33] **Lorenzo Zaninetti**: *The galaxies distribution as given from the Voronoi Diagrams*. In the journal of Astronomy and Astrophysics, Italia.

[34] **Massimo Ramella, Walter Boschin, Dario Fadda e Mario Nonino**: *Finding Galaxy Clusters Using Voronoi Tessellations*. In the journal of Astronomy and Astrophysics, vol. 368, pp. 776-786, 2001.

[35] **Marina Gavrilova**: *Weighted Voronoi Diagrams in Biology*. http://pages.cpsc.ucalgary.ca/~marina/vpplants/.

[36] **B. J. Gellatly e J. L. Finney**: *Calculation of protein volumes: An alternative to the Voronoi Procedure*. In the Journal of Molecular Biology, vol 161, pp 305-322, 1982.

[37] **F. M. Richards**: *The interpretation of protein structures: total volume, group volume distributions, and packing density*. In the Journal of Molecular Biology, vol 82, pp 1-14, 1974.

[38] **V. N. Serezhkin, V. A. Blatov ed A. P. Shevchenko**: *Voronoi-Dirichlet polyhedra for uranium (vi) atoms in oxygen containing compounds*. In the Russian Journal of Coordination Chemistry, vol 21, pp 155-161, 1995.

[39] **Mar Serrano, Gianni De Fabritiis, Pep Español, Eirik Flekky e Peter Coveney**: *Mesoscopic dynamics of Voronoi fluid particles*. In the Journal of Physics A: Mathematics and General, col 35, pp 1605-1625, 2002.

[40] **Dan Stora, Pierre-Olivier Agliati, Marie-Paule Cani, Fabrice Neyret e Jean-Dominique Gascuel**: *Animating lava flows*. In Graphics Interface, 1999.

[41] **Sei gradi di separazione (sociologia)**. http://it.wikipedia.org/wiki/Sei_gradi_di_separazione_(sociologia).

[42] **Voronoi Diagram**.
http://en.wikipedia.org/wiki/Voronoi.

[43] **Sujoy Basu, Sujata Banerje, Puneet Sharma e Sung-Ju Lee**: *NodeWiz: Peer-To-Peer Resource Discovery for Grids*. In Proceedings of 5th International Workshop on Global and Peer-to-Peer Computing (May 2005), IEEE Computer Society Press, pp. 213–220.

[44] **Roger Zimmermann, Wei-Shinn Ku, and Haojun Wang**: *Spatial Data Query Support in Peer-to-Peer Systems*. In Proceedings of the 28th International Computer Software and Applications Conference (COMPSAC 2004), Workshop on Geographic and Biological Data Management, Hong Kong, China, 2004.

[45] **Payload**.
http://www.dizionarioinformatico.com/cgi-lib/diz.cgi?frame&key=payload.

[46] **Frank Nielsen, Jean-Daniel Boissonnat e Richard Nock**: *Bregman Voronoi Diagram: Properties, Algorithms and Applications*. In Proceeding of 18th ACM-SIAM Symposium Discrete Algorithms, 2007.

[47] **Evangelos Kranakis, Harvinder Singh ed Jorge Urrutia**: *Compass Routing on Geometric Networks*. In The Proceedings of the 11th Canadian Conference on Computational Geometry, 1999.

[48] **Jörg Liebeherr, Michael Nahas e Weisheng Si**: *Application-Layer Multicasting with Delaunay Triangulation Overlays*. IEEE Journal on Selected Areas in Communications, pp. 14721488, Ottobre 2002.

[49] **Farnoush Banaei-Kashani e Cyrus Shahabi**: *SWAM: A Family of Access Methods for Similarity-Search in Peer-to-Peer Data Networks*. Information and Knowledge Management (CIKM'04), Washington D.C., November 2004.

[50] **Andrea Salvadori**: *Nomade: overlay network dinamiche basate su diagrammi di Voronoi per ambienti virtuali distributi*. Tirocinio di Laurea, Università di Pisa, Aprile 2007.

[51] **Franco Guidi Polanco**: *GoF's Design Patterns in Java*. Giugno 2002.

[52] **FIFO**.
`http://en.wikipedia.org/wiki/FIFO`.

[53] **Multithreading**.
`http://it.wikipedia.org/wiki/Multithreading`.

[54] **Churn Rate**.
`http://en.wikipedia.org/wiki/Churn_rate`.

[55] **Mark de Berg, Marc van Kreveld, Mark Overmars e Otfried Schwarzkopf**: *Computational Geometry: Algorithms and Applications*. Seconda edizione, Springer-Verlag.

[56] **Diagramma di Voronoi e triangolazione di Delaunay**.
`http://www2.mate.polimi.it/corsi/viewpaper.php?id=92\&cf=5`.

[57] **Franz Aurenhammer**: *Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure*. ACM Computing Surveys, 23(3):345-405, 1991.

[58] **Duncan Watts e Steven Strogatz**: *Collective dynamics of "small-world" networks*, 1998, Nature pp 393-400, Londra.

[59] **Grafi geometrici**.
`http://en.wikipedia.org/wiki/Geometric_graph_theory`.

[60] **Avatar**.
`http://it.wikipedia.org/wiki/Avatar_(disambigua)`.

[61] **Qhull**.

http://www.qhull.org/.

[62] **Qhull Performance**.

http://www.qhull.org/html/qh-in.htm#performance.

Ringraziamenti

Grazie a tutti.

La gioia per aver portato a termine questa avventura è immensa.

Ringrazio la Prof. Laura Ricci che mi ha dato la possibilità di fare qualcosa di appassionante e mi ha sempre incoraggiato con un rassicurante sorriso.

Ringrazio il Prof. Ranieri Baraglia sempre disponibile e gentile.

Ringrazio il Dott. Michele Albano che mi ha aiutato e assistito continuamente.

Ringrazio il Dott. Diego Puppin per i suoi consigli tecnici.

Ringrazio papà, mamma e Ila che mi hanno sopportato e incoraggiato soprattutto negli ultimi mesi del 2006, in cui non sapevo dove sbattere la testa.

Ringrazio Tommaso che mi è sempre stato vicino, che mi ha dato tutto il sostegno di cui avevo bisogno e che è sempre riuscito a farmi ridere anche quando ero triste.

Ringrazio Silvia e Federica, mie compagnie di viaggio in questi sei anni.