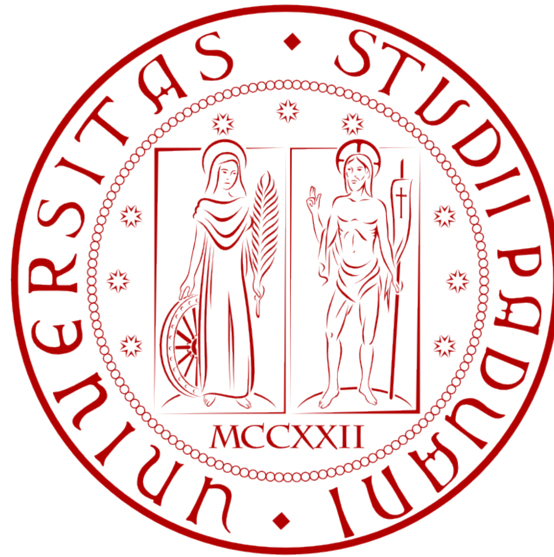

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria dell'Informazione

TESI DI LAUREA

PARITORRENT: ROUTING DHT

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Michele Bonazza

LAUREANDO: Alessandro Dal Corso

A.A. 2010-2011

Ai miei cari

Indice

1 PariPari **5**
1.1 Introduzione . . . 5
1.2 Struttura di PariPari . . . 5
1.3 eXtreme Programming . . . 7

2 Il protocollo BitTorrent **9**
2.1 Descrizione generale . . . 9
2.2 Il file .torrent . . . 9
2.3 Tracker . . . 10
2.4 Peer Wire Protocol . . . 11

3 DHT **13**
3.1 Caratteristiche generali . . . 13
3.2 Struttura . . . 14
3.2.1 Partizionamento del keyspace . . . 14
3.2.2 Overlay network . . . 15

4 La Mainline DHT **17**
4.1 Note storiche e BEP . . . 17
4.2 Descrizione generale . . . 17
4.3 Funzionamento . . . 20
4.4 Routing Table . . . 21
4.4.1 Aggiunta di un nodo . . . 22
4.4.2 Ricerca dei nodi più vicini ad un dato nodo (*Algoritmo FB*) 24
4.5 Protocollo KRPC . . . 30

5 Implementazione della MainLine DHT e della routing table **33**
5.1 Il plug-in Torrent . . . 33
5.2 L'estensione alla MainLine DHT . . . 34
5.3 La routing table . . . 37
5.3.1 Ricerca di un bucket (*getBucketIndexOf*) . . . 39
5.3.2 Aggiunta di un nodo (*addNode*) . . . 41
5.3.3 Algoritmo FB (*getClosestNodes*) . . . 44
5.4 Gestione della routing table . . . 48
5.4.1 Bootstrap . . . 48
5.4.2 Ricezione di un messaggio . . . 49

6 Conclusioni e stato dell'arte **50**

7 Bibliografia **51**

Sommario

Questa tesi riguarda BitTorrent, un protocollo sviluppato negli ultimi anni per permettere il file-sharing peer-to-peer, e attualmente uno dei più diffusi al mondo. Nel progetto PariPari, uno specifico plug-in, Torrent, è dedicato all'implementazione in Java di questo protocollo e di alcune delle sue numerose estensioni. Il plug-in Torrent di PariPari vuole proporsi come un'alternativa valida e open-source ai vari client BitTorrent (proprietari e non) in circolazione.

La trattazione riguarderà l'implementazione dell'estensione DHT (Mainline DHT) al protocollo, e in particolare tratterà della creazione, struttura e gestione di una routing table, ovvero una struttura dati che mantiene i collegamenti all'interno del sistema distribuito. Si fornirà una descrizione generale delle DHT (Distributed Hash Table, o tabella hash distribuita), illustrandone principi di funzionamento e algoritmi di base. Lo scopo dell'estensione è migliorare le prestazioni del client e la robustezza della rete di utenti.

1 PariPari

Questa sezione si propone di descrivere gli aspetti principali e la filosofia di PariPari, il software *peer-to-peer*[1] attualmente in sviluppo presso la facoltà di Ingegneria dell'Università degli Studi di Padova.

1.1 Introduzione

Il progetto *PariPari* si pone come obbiettivo lo sviluppo di una applicazione *peer-to-peer serverless*. Il progetto è stato presentato per la prima volta dal professor Paolo Bertasi nel 2006, introducendo un sistema DHT basato su *Kademlia*, un sistema *peer-to-peer* decentralizzato sviluppato originariamente da Petar Maymounkov e David Mazières nel 2002. La più nota implementazione di Kademlia è quella del client di *file-sharing* eMule, ma ne esistono molte altre.[2] Una caratteristica molto importante di PariPari è la sua struttura modulare, che lo rende particolarmente scalabile alle esigenze del singolo utente.

1.2 Struttura di PariPari

La struttura di PariPari, come si è detto, è modulare: ogni funzionalità del programma è fornita come un modulo (o *plug-in*) aggiuntivo per il programma, il che consente agli utenti di poter aggiungere o rimuovere *plug-in* in funzione delle proprie esigenze.

Esiste un modulo principale, detto `Core`, che si occupa di avviare o chiudere il programma, di gestire le comunicazioni tra i vari moduli (tramite un altro modulo, `PluginSender`) e di fornire i permessi necessari a compiere determinate azioni da parte dei *plug-in* (come l'accesso alla gestione dei file su disco o alla rete).

Tutti gli altri *plug-in*, escluso il `Core`, si suddividono in due grandi macro-aree: una, detta *cerchia interna*, fornisce le risorse essenziali al funzionamento del programma. Tra i *plug-in* che contiene si ricordano:

`DHT:` questo modulo si occupa di memorizzare le informazioni relative agli altri utenti nella rete dei client PariPari, utilizzando una versione modificata di Kademlia.

`Connectivity:` questo modulo permette l'interfaccia con la rete IP esterna alla DHT di PariPari, permettendo l'apertura di connessioni con altri utenti della rete.

`Local Storage:` questo modulo permette di interfacciarsi con il disco rigido del sistema fornendo funzioni quali il salvataggio o il caricamento di file su disco.

`Credits:` Questo modulo implementa un sistema per poter assegnare dinamicamente le risorse della macchina ai vari *plug-in*, utilizzando dei crediti. Essi sono delle monete che vengono pagate al Core dagli altri *plug-in* quando essi vogliono ottenere l'accesso a determinate risorse della macchina, come la rete o l'hard disk.

[1] Si veda par. 3.1 a pagina 13 per maggiori informazioni.

[2] Per un elenco parziale, si veda http://en.wikipedia.org/wiki/Kademlia#Implementations.

La seconda macro-area in cui si suddividono i plug-in è detta *cerchia esterna*. In quest'area sono inclusi tutti gli altri moduli non necessari al funzionamento di base di PariPari, che quindi possono essere aggiunti o rimossi a seconda delle esigenze dell'utente. Qui si concentra la grande maggioranza dei *plug-in*, tra cui quello interessato da questa tesi, `Torrent`. Tra i principali *plug-in* della cerchia esterna vi sono, ad esempio:

`Torrent`, che si occupa di fornire un'implementazione del protocollo BitTorrent e delle sue estensioni;

`Mulo`, un'implementazione per fornire supporto alle reti *ed2k* e *Kad*. Il nome prende spunto dai più celebri cugini' eMule ed eDonkey2000;

`Web Server`, un web server distribuito;

`IM`, un servizio di messaggistica istantanea che si avvale del protocollo MSN (MicroSoft Network);

`IRC`, un servizio simile al precedente, ma basato sul protocollo IRC (Internet Relay Chat);

`Distributed Storage`, un servizio per condividere file tra computer diversi;

`VoIP`, un sistema per poter eseguire delle chat vocali tramite IP.

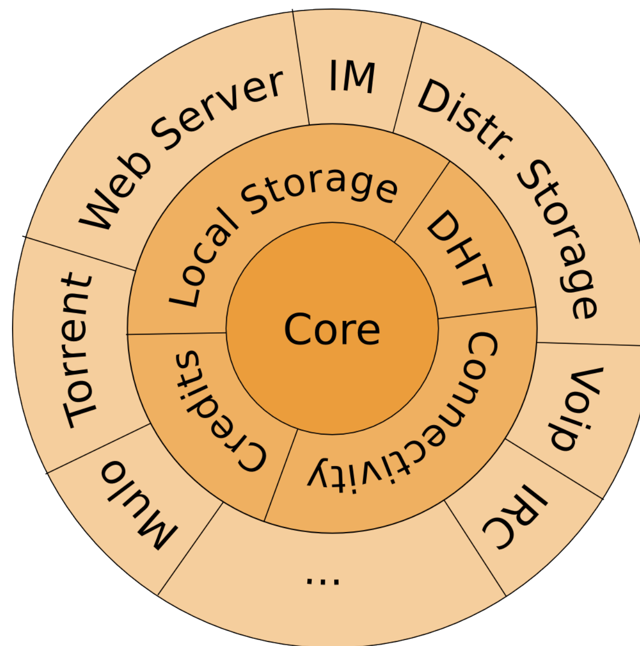

Figura 1: La struttura di PariPari, con evidenziate la cerchia interna ed esterna.

1.3 eXtreme Programming

L'*eXtreme Programming* (XP) è una metodologia di sviluppo software ideata da Kent Beck tra il 1996 e il 1999 mentre lavorava ad un sistema di gestione dei pagamenti della Chrysler, il *Chrysler Comprehensive Compensation System*. In realtà la grande novità dell'XP non fu l'ideazione di nuove metodologie di sviluppo, ma l'unione di diverse metodologie già utilizzate nella comunità software internazionale. In questo modo, prendendo il meglio da ogni tecnica, Beck voleva sommarne gli effetti benefici estremizzandone l'efficacia complessiva (da cui *extreme*).

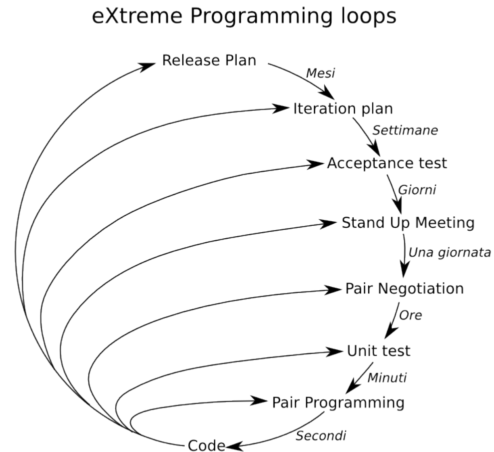

Figura 2: Schema dell'eXtreme Programming. In corsivo è indicata la suddivisione temporale delle varie fasi.

L'XP prevede di suddividere lo sviluppo software in cicli, nei quali vengono via via implementate le caratteristiche del software richieste dal cliente (*customer*). L'XP inoltre prevede un'estesa comunicazione tra cliente e team di sviluppo (*developing team*), in modo che anche delle eventuali modifiche alle caratteristiche base del prodotto possano essere incluse nel ciclo di sviluppo e implementate velocemente.

L'XP prevede cinque regole fondamentali sui vari aspetti del codice[3]:

Planning (pianificazione): La progettazione del software è divisa in cicli di sviluppo. Esiste un piano di rilascio del software *(release plan)*, nel quale sono definite le funzioni (*features*) che verranno inserite via via nel programma. Bisogna cercare di fare delle release piccole e ravvicinate, con dei piani specifici per ogni ciclo di sviluppo (*iteration plan*);

Managing (gestione): Questa parte prevede di gestire il team dietro lo sviluppo del software: bisogna fornire al team uno spazio sostenibile in cui poter lavorare, fare dei meeting di sviluppo giornalieri, per risolvere i problemi che si possano essere verificati durante la giornata.

Designing (design). In questa parte viene posto l'accento sul mantenere il design più semplice possibile, utilizzando delle metafore per descrivere il sistema che si sta utilizzando; rendendo la progettazione simile alla soluzione di un problema reale, diviene più facile fare delle scelte di design appropriate. La regola prevede tra le altre cose di evitare di scrivere del codice che si prevede servirà in futuro: per la maggior parte delle volte questo codice non servirà, inquinando il progetto con codice inutile.

Coding (scrittura del codice). In questa parte si definiscono le specifiche per progettare del codice ottimale. Bisogna prima di tutto definire uno standard comune per tutti. La

[3]http://www.extremeprogramming.org/rules.html

programmazione è organizzata a coppie (*pair programming*): i programmatori scrivono prima del codice dei test su di esso (*unit tests*), in modo da avere già idea di come il codice si dovrà comportare una volta scritto. Per poter mantenere il codice aggiornato è prevista per esso una struttura a repository, in cui le modifiche vanno continuamente inserite (*committed*) e integrate con il lavoro eseguito in parallelo da altre coppie di programmatori sulle stesse parti di codice. In questo modo i problemi vengono alla luce presto, evitando lunghe sessioni di debug per risolvere i problemi.

Testing (test). Il testing prevede di scrivere dei test su ogni parte del codice. Prima di essere diffuso, il codice deve passare tutti gli unit test, e se viene trovato un bug altri test vengono creati per approfondirlo e risolverlo. Si creano inoltre dei test più generici (*acceptance tests*) per testare le funzionalità principali del programma.

All'interno del modulo Torrent l'XP è stato applicato per quel che riguarda e il testing a coppie e gli unit test (essendo il codice scritto in Java, si è scelto di usare *JUnit*, scritto dallo stesso Kent Beck).

2 Il protocollo BitTorrent

Si descrive brevemente in questo paragrafo il funzionamento di base del protocollo *BitTorrent*, sulla base del quale è stato creato il modulo Torrent di PariPari.

2.1 Descrizione generale

BitTorrent è un protocollo per la condivisione dei dati *(file-sharing)* utilizzato per distribuire grandi moli di dati attraverso la rete Internet. Ad oggi, è uno dei protocolli più diffusi per il trasferimento di file di grande dimensione: alcune stime riportano una percentuale del traffico Internet occupato dai client BitTorrent tra il 2755%[4].

La prima versione del protocollo BitTorrent fu proposta per la prima volta da Bram Cohen nel 2001, ed è mantenuto ad oggi dalla compagnia fondata dallo stesso Cohen, la *BitTorrent, Inc.* Oltre al client principale sviluppato dalla *BitTorrent, Inc.* esistono moltissimi altri client che sfruttano il protocollo, disponibili su quasi tutte le piattaforme disponibili.

Figura 3: Il logo della BitTorrent, Inc.

Uno dei punti cardine del protocollo BitTorrent è che un utente che utilizza il protocollo (*client*) non solo scarica il file, ma contemporaneamente ne diventa anche distributore. Ogni file che viene scaricato viene suddiviso in piccole parti, che vengono scambiate tra gli utenti della rete: essi tentano di scaricare le parti che mancano loro del file e contemporaneamente condividono le parti che hanno già scaricato. Per poter distribuire un file attraverso BitTorrent sono necessari:

- Un file *metainfo* (con estensione .torrent);
- Un *tracker* BitTorrent (un server web che utilizza il protocollo HTTP, *Hyper-Text Transfer Protocol*);
- Un client dotato di browser web e client BitTorrent installato e funzionante.

2.2 Il file .torrent

Per poter scaricare un file è necessario ottenere un file *metainfo* (ovvero di informazioni associate), solitamente con estensione .torrent, che contiene tutte le informazioni necessarie per il download del file. Il file *metainfo* è necessario per questioni di prestazioni, in quanto la ricerca dei file all'interno della rete BitTorrent non è possibile (i file non vengono indicizzati, cosa che invece ad esempio succede nella rete eDonkey2000). Il file .torrent solitamente viene reperito navigando in Internet, spesso tramite motori di ricerca specializzati.

Il file contiene diverse informazioni, suddivise come coppie chiave-valore di un dizionario:

[4]http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf.

info: un sotto-dizionario che descrive il file presente nel torrent, fornendo informazioni come il numero o la lunghezza dei pezzi, e il nome del file. Un torrent può contenere anche delle sotto-cartelle e vari file invece che uno solo.

announce: L'URL di announce del torrent (si veda la sezione Tracker più avanti).

announce-list: Lista di URL di announce alternativi (opzionale).

creation-date: Data di creazione del torrent, nel formato UNIX standard (numero di secondi passati dal 1° gennaio 1970 00:00:00 UTC) (opzionale).

comment: commento dell'autore del torrent (opzionale).

created-by: nome e versione del programma che ha generato il file .torrent (opzionale).

Le informazioni nel file metainfo sono codificate secondo la codifica *bencode*.[5] Una volta che si è ottenuto il file .torrent, è possibile cominciare il download del file.

Ogni file .torrent possiede un identificativo, detto *infohash*, ovvero una stringa di 20 byte che consiste nell'hash SHA-1[6] del dizionario info (che quando è codificato si presenta come una stringa).

2.3 Tracker

Il *tracker* è il mezzo principale per poter ottenere informazioni sugli altri utenti (*peers*, - compagni) che stanno scaricando lo stesso file .torrent. Il tracker non è altro che un server che risponde a richieste HTTP di tipo GET (preleva). L'URL a cui poter eseguire queste richieste è indicato nella chiave *announce* del file .torrent. Il client deve inviare un insieme di informazioni al tracker (infohash, identificativo del client, eccetera) ed esso risponde con un certo numero di peer utili per quell'infohash (di solito 50) nelle sue liste.

Il tracker inoltre mantiene statistiche sulla salute del torrent e sulle statistiche di download. La salute di un torrent si misura solitamente sulla base di quanti utenti in media sono presenti nella rete che possiedono quel file. Questi utenti vengono suddivisi in due categorie: i cosiddetti *seed*, ovvero quelli che possiedono il file completo, cioè che non stanno scaricando ma solo condividendo quel particolare file, e i *leech*, ovvero quelli che non hanno ancora terminato di scaricare il file. Un basso numero di *seed* riduce la probabilità di trovare il file completo disponibile, ovvero potrebbe succedere in alcuni momenti di non riuscire a scaricare tutte le parti del file.

[5]Essa è una particolare codifica utilizzata per codificare informazioni nel protocollo torrent: essa può codificare stringhe, interi, liste e dizionari.

[6]L'hash SHA-1 è una funzione particolare che trasforma una qualunque stringa in un'altra (hash) di una lunghezza prefissata. Esso è utilizzato per questioni di sicurezza (dato che dal risultato non è possibile risalire alla stringa che ha generato il file) o, come in questo caso, di identificazione, dato che la funzione è costruita in modo che la probabilità che due hash siano uguali (collidano) è molto bassa.

2.4 Peer Wire Protocol

Una volta che il client ha ottenuto le informazioni per contattare un altro peer, si deve comunicare con esso. Per fare questo si utilizza il *Peer Wire Protocol*, un sistema di messaggi definito dalle specifiche del protocollo BitTorrent.

Quando inizia la comunicazione tra due nodi A e B, A invia a B un particolare messaggio, detto *handshake* (stretta di mano). Se il peer B sta scaricando lo stesso torrent di A (il che può essere verificato dall'infohash, che viene inviato assieme all'handshake) allora esso risponde con un altro messaggio handshake. Dopo che i messaggi handshake sono stati ricevuti correttamente, si prosegue nello scambio di informazioni.

Ogni peer (ad esempio A) deve mantenere per ogni connessione verso un peer remoto (ad esempio B) due variabili sullo stato del peer:

- ***choked***[7]: (strozzato) la connessione è *choked*, vuol dire che B non risponderà alla richieste di A fino a che esso non toglierà da esso lo stato di choked (*unchoking*). Questo è necessario per non congestionare troppo la rete con connessioni multiple in contemporanea.

- ***interested***: (interessato) se lo stato è *interested*, vuol dire che B è interessato a scaricare delle parti di file che il client A può offrire. Questo significa che il peer B comincerà a richiedere ad A appena A sarà *unchoked*.

Ora si osservino gli altri messaggi del *Peer Protocol* oltre al messaggio di tipo *handshake* (si suppone sempre che il messaggio sia scambiato da A verso B):

keep-alive: Questo messaggio è usato per mantenere viva la connessione tra due peer se per un determinato lasso di tempo tra di loro non vengono scambiati messaggi. Questo tempo in genere è di due minuti.

choke: Viene inviato quando A vuole interrompere temporaneamente le comunicazioni con B, come descritto sopra.

interested: Viene inviato quando il nodo A è interessato ai pezzi del nodo B.

not interested: Viene inviato se il nodo B non ha più pezzi che interessano al nodo A.

bitfield: Questo messaggio opzionale può essere inviato soltanto subito dopo la fine delle comunicazioni dell'handshake. Esso viene utilizzato per comunicare quali pezzi si possiedono e quali no, affinché l'altro peer possa regolarsi su quali pezzi richiedere. Non è necessario inviare il messaggio se non si possiedono pezzi per il torrent.

request: A invia questo messaggio quando vuole richiedere a B un pezzo, di cui viene inviato l'identificativo.

piece: Questo messaggio viene inviato da B in risposta al messaggio request precedente. Esso è il download del pezzo vero e proprio, e può avvenire solo se il peer B è nello stato unchoked.

[7]Dalla specifica di BitTorrent: Choking is a notification that no data will be sent until unchoking happens.

have: Questo messaggio viene inviato da un client che ha appena finito di scaricare un pezzo per segnalare che lo possiede e lo rende disponibile per il download.

cancel: Si invia questo messaggio se si vuole interrompere il download di un pezzo in corso da un peer.

port: Per la spiegazione di questo messaggio, utilizzato nella *MainLine DHT*, si veda la Sezione 4 più avanti.

Il download di un file si articola quindi come una serie di messaggi *handshake-request-piece*, finché non si è scaricato tutto il file. Solitamente il programma tenta di scaricare prima i pezzi più rari, e verso la fine del download si utilizzano degli algoritmi particolari per non far ridurre la velocità di download.

3 DHT

In questa sezione si definiscono e si descrivono le tabelle hash distribuite (DHT, *distributed hash tables*), dei sistemi distribuiti che hanno avuto un grande sviluppo negli ultimi anni.

3.1 Caratteristiche generali

Una DHT svolge essenzialmente le funzioni di una *hash table*, ovvero contenere un insieme di coppie chiave-valore. Le chiavi (come il nome di una persona) vengono mappate nei valori (come il numero di telefono della persona) attraverso una particolare funzione, detta *funzione hash*. La funzione hash trasforma la chiave in un indice utilizzabile per memorizzare i valori all'interno di una struttura dati, come un array.

La funzione hash deve garantire che due chiavi per quanto possibile non vengano associate allo stesso valore, e che gli hash siano il più possibile distribuiti uniformemente (ovvero devono evitare che i valori si ammassino in determinati sottoinsiemi di hash). Le DHT funzionano come un'hash table, ma i dati non si trovano tutti nello stesso luogo fisico; essi vengono distribuiti tra vari utenti della rete. Ogni utente del sistema (detto *nodo*) può in maniera efficiente ricavare il valore associato ad una singola chiave attraverso la rete.

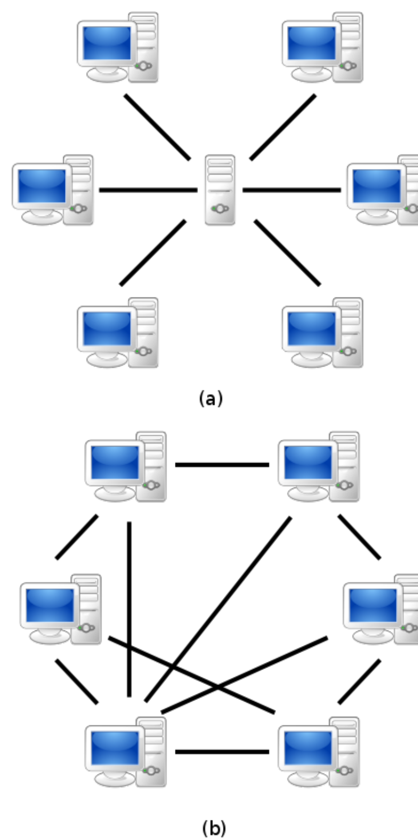

Figura 4: Sistema client-server (a) e sistema p2p (b) a confronto.

In una DHT, la responsabilità di mantenere la mappatura tra chiavi e valori è distribuita tra i vari nodi, in modo tale che un cambiamento nell'insieme dei nodi non causi modifiche a cascata all'intera struttura della DHT stessa. In questo modo il sistema diviene estremamente scalabile alle esigenze della rete, essendo capace di gestire le continue connessioni, disconnessioni o fallimenti dei singoli nodi. Nonostante questa instabilità delle connessioni, esistono dimostrazioni matematiche che forniscono dei limiti superiori all'efficienza degli algoritmi utilizzati.

Le tabelle hash distribuite ebbero un grande sviluppo nei primi anni duemila, quando avvenne il perfezionamento di alcuni modelli di DHT. Negli anni precedenti, le prime tipologie di reti distribuite erano utilizzate soprattutto nelle architetture *peer-to-peer* (P2P), che però erano create ad hoc, caotiche e inefficienti. Nei sistemi *peer-to-peer* non esiste una gerarchia tra i vari partecipanti, e gli utenti della rete possono comunicare tra loro senza dover attendere la mediazione di un server centrale.

Tra i difetti nelle prime architetture, c'erano l'utilizzo di un server centrale per indicizzare i nodi (Napster), il che rendeva la rete vulnerabile agli attacchi esterni; oppure un sistema di *flooding* (inondazione) dei messaggi per comunicare tra gli utenti (Gnutella), ovvero ogni

utente trasmetteva i messaggi (query) a tutti i nodi conosciuti, causando però l'inefficienza complessiva del sistema; oppure un key-based routing (FreeNet), ovvero un sistema in cui ogni file è associato ad una singola chiave, dove però non si garantisce che un certo file venga trovato pur essendo presente nella rete.

Grazie al lavoro di alcuni istituti universitari molti modelli di DHT vennero sviluppati ed implementati dalle comunità *open source*, superando di gran lunga in performance le vecchie architetture. Tra le più famose architetture *open source* sviluppate si ricordano *Chord, Kademlia, Tapestry* e *Pastry*. In questo elaborato, in particolare, sarà descritta una particolare architettura basata su *Kademlia*, la *Mainline DHT* del protocollo BitTorrent.

3.2 Struttura

La struttura generica di una DHT può essere suddivisa in due grandi macro-aree: lo spazio delle chiavi (*keyspace*) e il network di supporto (*overlay network*). Il primo è uno spazio astratto di stringhe (dette *identificativi* o *ID*, ad esempio le stringhe binarie lunghe 128 bit) usate per definire le chiavi dell'hash table in una DHT; ogni utente ha la responsabilità su una parte dello spazio delle chiavi. Gli utenti che desiderano un valore dell'hash table contenuto in quella parte dovranno quindi in un qualche modo rivolgersi all'utente responsabile per quella parte dello spazio per ottenere il dato. La DHT quindi prevede uno schema di partizionamento (*keyspace partitioning*) che decide quali porzioni di questo spazio siano destinate ai singoli nodi. L'*overlay network* invece permette le connessioni tra i nodi, in modo tale che essi possano comunicare per poter ottenere le informazioni sui proprietari delle parti del *keyspace*.

3.2.1 Partizionamento del keyspace

Il partizionamento dello spazio delle chiavi è una caratteristica fondamentale all'interno delle DHT. In ogni DHT, ad ogni nodo è assegnato un identificativo unico (ID), solitamente una chiave di 160 o 128 bit generata pseudo-casualmente. Anche i dati presenti nella DHT hanno un identificativo simile; ad ogni identificativo è quindi associata una chiave della DHT. Data la natura distribuita del sistema, ogni chiave di un dato deve avere un luogo fisico in cui deve essere possibile reperire il dato stesso.

Dato che il numero di ID possibili è sempre di diversi ordini di grandezza superiore alla somma del numero di nodi e del numero di dati presenti all'interno del sistema, si necessita di suddividere lo spazio delle chiavi in modo da poter definire per ogni nodo di quale gruppo di chiavi esso è responsabile. Anche nel caso in cui ogni computer della terra (secondo stime recenti, circa un miliardo[8]) condividesse un milione di file unici, si avrebbero circa un milione di miliardi (10^{15}) identificativi occupati, che è comunque un milionesimo di miliardesimo (per una DHT a 128 bit) del numero complessivo di ID possibili ($\approx 3,4 \cdot 10^{38}$).

Le DHT generalmente utilizzano uno schema detto di *consistent hashing*: nel caso si modifichi una mappatura chiave-valore in un'hash table, questa non causerà un effetto a valanga in tutta la struttura dati, forzando la rimappatura di quasi tutte le chiavi con i loro valori (come avviene nelle hash table classiche). Solo un piccolo numero di chiavi infatti verrà rimappato.

[8]http://www.science-portal.org/in/71

Il consistent hashing consiste nell'immaginare le chiavi come dei punti su di una circonferenza su cui nodi e dati vengono mappati in maniera pseudo-casuale. In seguito si definisce una particolare funzione $\delta(k_1, k_2)$, detta *distanza*, dove k_1 e k_2 sono due identificativi. Un nodo con ID i_n possiede tutte le chiavi per le quali lui è l'identificativo più vicino secondo δ. Ovvero:

$$K_n = \{k \mid \min_{i_k}(\delta(k, i_k)) = i_n,\ i_k \in I\}$$

Dove I è l'insieme degli ID dei nodi connessi alla DHT.

Nelle implementazioni effettive delle DHT, spesso si preferisce comunque affidare la gestione di una coppia chiave-valore a più nodi in contemporanea: aggiungendo questa ridondanza, si aumenta il traffico della rete, ma si guadagna molto in termini di affidabilità e robustezza della rete stessa.

3.2.2 Overlay network

Ogni nodo al suo interno deve mantenere un insieme di collegamenti verso altri nodi, detti comunemente *vicini* (*neighbours*) o *routing table.* Ogni nodo sceglie i suoi vicini secondo la topologia della rete fissata in ogni schema DHT. Ad esempio, nelle DHT di tipo Chord si usa una tipologia di rete ad anello, nella quale ogni utente è collegato solamente ai suoi due compagni più vicini. Per comunicare i nodi si passano le informazioni da uno all'altro attraverso l'anello.

Le DHT, per comunicare con un nodo che possiede la chiave k, si basano sull'assunto che ogni nodo $n_1$possiede o un collegamento diretto al nodo che possiede k o un collegamento ad un altro nodo n_2 tale che $\delta(n_2, k) < \delta(n_1, k)$, ovvero più vicini alla chiave k in termini della distanza definita precedentemente. Quando questa ricerca ritorna sempre lo stesso risultato, allora si è giunti al nodo più vicino a k nella DHT, che per definizione è il nodo che possiede k.

Questo permette di definire un algoritmo greedy[9] per la ricerca di una chiave k:

Algoritmo 1 getClosestNode(Key k)

```
1 def Node n il nodo più vicino a k nella mia routingTable;
2   while(loop) {
3     def Node m = nodo più vicino a k nella routingTable di n;
4     if( ID(m) = ID(n) )
5       return n;
6     n=m;
7 }
```

Si osserva che da questo algoritmo si ha un importante *trade-off*: infatti, l'efficienza di questo algoritmo è tanto più elevata quanto le dimensioni della routing table sono elevate.

[9]Un algoritmo greedy è un algoritmo che sceglie sequenzialmente la soluzione ottimale alle parti di un problema sperando che complessivamente portino alla soluzione ottimale del problema complessivo. Si noti che non sempre un algoritmo greedy darà una soluzione ottimale.

E' importante mantenere sia alta l'efficienza dell'algoritmo (ovvero la lunghezza media del percorso tra nodi) che piccole le dimensioni della routing table (in modo da ridurre il traffico di overhead[10]).

Utilizzando la notazione O-grande, e indicando con n il numero di identificativi possibili nella DHT, nelle DHT di tipo Chord le dimensioni della routing table sono $O(1)$, in quanto ogni nodo è collegato a soli altri due nodi. La lunghezza media del percorso tra due nodi però diventa $O(n)$,in quanto una richiesta può dover attraversare anche metà dei nodi connessi alla rete prima di giungere a destinazione. In altre tipologie, come *Kademlia*, la routing table ha dimensioni $O(log(n))$, il che permette una lunghezza media sempre $O(log(n))$. Inoltre, nella scelta dei nodi da includere nella routing table in alcune DHT più avanzate si tende a inserire i nodi con latenza più bassa (ovvero nodi più veloci), migliorando l'efficienza complessiva.

[10] L'overhead, nelle reti di trasmissione e nei programmi di file-sharing, è tutto quel traffico non strettamente legato al trasferimento effettivo dei dati. Nel caso della Mainline DHT, tutto il traffico generato dalla DHT è solo di overhead, in quanto è il modulo principale di Torrent ad occuparsi del trasferimento dei dati.

4 La Mainline DHT

4.1 Note storiche e BEP

Si passa ora a descrivere l'oggetto di questo elaborato, la *Mainline DHT*. Storicamente, la prima applicazione di una DHT al protocollo BitTorrent si ebbe quando il client *Vuze* (prima conosciuto come *Azureus*), giunto alla versione 2.3.0.0 introdusse una propria implementazione di una DHT, come supporto per torrent cosiddetti *trackerless*, ovvero che non avessero bisogno del supporto di un tracker per funzionare. Poco dopo, la BitTorrent, Inc. la compagnia che inizialmente studiò e creò il protocollo BitTorrent, rilasciò una nuova versione del proprio client, introducendo un supporto per una DHT propria, incompatibile con quella di Vuze. Questa implementazione è detta comunemente *Mainline DHT*.[11] Entrambe le DHT, la MainLine e quella di Azureus, sono basate su *Kademlia*. Oltre al client BitTorrent ufficiale sviluppato dalla BitTorrent, Inc., altri client alternativi come *µTorrent*, *BitComet*, *Transmission* e *BitSpirit* supportano la Mainline DHT.

La Mainline DHT è una BEP (*BitTorrent Enhancement Proposal*, proposta di miglioramento di BitTorrent), ovvero una proposta (identificata da un numero) di estensione al protocollo BitTorrent. Una BEP contiene una descrizione tecnica e concisa dell'estensione proposta al protocollo; il sistema delle BEP prevede inoltre un processo di revisione continuo, finché le proposte non vengono approvate dalla comunità degli sviluppatori di BitTorrent. Attualmente solo alcune BEP sono state approvate (0, 1, 2, 3, 4, 20 e 1000), mentre due (9 e 23) sono state approvate ma attendono l'approvazione definitiva da parte della comunità prima di essere implementate. Le prime BEP (1 e 2) definiscono il processo stesso delle BEP, mentre la numero 3 è il protocollo di BitTorrent stesso. Le altre BEP sono solo allo stato di proposta (*Draft*), ma alcune di esse sono ormai diventate degli standard de facto della comunità, come le *fast extension*, il *superseeding* e la *Mainline DHT* stessa.[12]

4.2 Descrizione generale

La Mainline DHT utilizza una particolare tabella hash distribuita per memorizzare al suo interno le cosiddette *peer contact information*, ovvero le informazioni utili a contattare altri peer per il torrent che si sta scaricando. Invece di affidarsi ad un server fisico a cui gli utenti si registrano, ogni nodo nella DHT svolge parte delle funzioni di un tracker.

Questa DHT è stata pensata per permettere l'utilizzo di torrent privi di tracker (*trackerless*), ma nella realtà si usa affiancare la DHT ad un elenco di tracker esistenti come mezzo alternativo per procurarsi dei peer. Mentre il protocollo standard di BitTorrent utilizza esclusivamente connessioni TCP, la Mainline DHT opera esclusivamente su UDP.[13]

[11] In generale, quello che viene sviluppato dalla BitTorrent, Inc. non ha un nome vero è proprio. Anche il client prodotto dalla compagnia ha semplicemente nome BitTorrent, senza qualifiche aggiuntive. Nella community degli sviluppatori si è quindi affermato il termine *MainLine* (linea principale) per indicare i prodotti della BitTorrent,Inc.

[12] Un elenco completo delle BEP è disponibile presso http://www.bittorrent.org/beps/bep_0000.html.

[13] Il TCP e l'UDP sono due protocolli di trasmissione dei pacchetti; la differenza principale tra i due è che il primo è di tipo a connessione, mentre il secondo no. Il primo garantisce tra le altre cose garanzia di trasmissione e mantenimento all'arrivo dell'ordine di partenza dei pacchetti, mentre il secondo è più veloce ma non garantisce l'avvenuta trasmissione nè la correttezza dei dati trasmessi; tuttavia esso garantisce velocità di

Ogni client BitTorrent è il nodo di una DHT, al quale viene assegnato un nuovo ID a 160 bit generato casualmente. Come dimostrato precedentemente, la probabilità che due ID nella rete collidano è molto bassa, quindi ogni nodo nella rete avrà un identificativo unico.

Come in *Kademlia*, la metrica di distanza utilizzata è lo *XOR* (*eXclusive OR*) tra gli identificativi dei nodi. L'operazione logica XOR è definita come segue:

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{array}{l} 100111000010\oplus \\ 011011110111= \\ \hline 111100110101 \end{array} \qquad \begin{array}{l} \text{9C2}\oplus \\ \text{6F7}= \\ \hline \text{F35} \end{array}$$

Tabella 1: Tabella di verità della funzione XOR ed esempio di utilizzo con due identificativi a 12 bit (in binario e in esadecimale).

Lo XOR tra gli identificativi viene interpretato come un intero di tipo *unsigned* (privo di segno): interi più grandi corrispondono a distanze più grandi. Si noti che la distanza tra due nodi non dipende assolutamente dalla distanza fisica tra di essi, né dal ritardo esistente tra i nodi.

Ogni nodo deve mantenere una *routing table*, ovvero un insieme di collegamenti verso altri nodi. Ad essa è possibile aggiungere nodi considerati buoni. Inoltre, dato un identificativo, è possibile trovare al suo interno in maniera efficiente un certo numero di nodi più vicini all'ID dato in metrica XOR. Per poter avviare la DHT, sono necessari un certo numero di nodi, detti nodi di bootstrap. Essi vengono aggiunti all'interno del file .torrent, sotto una speciale chiave chiamata nodes.

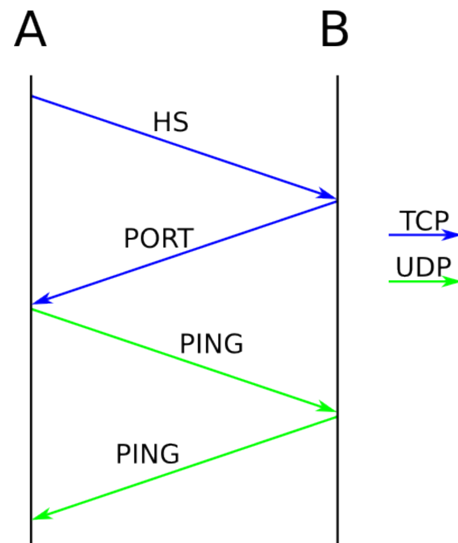

Figura 5: Comunicazione tra i nodi A e B descritta nel testo.

All'interno del protocollo BitTorrent è stata prevista la possibilità per i peer TCP di potersi scambiare la propria porta UDP, se entrambi supportano la Mainline DHT. Questo viene segnalato tramite un bit settato nei messaggi *handshake* tra due peer, ad esempio A e B. Se A manda un messaggio *handshake* segnalando che supporta la DHT, allora il peer B, ricevente l'*handshake*, manda il messaggio PORT del *peer protocol*, segnalando la propria porta UDP. A, ricevendo il messaggio PORT, tenta di pingare B tramite UDP sulla porta segnalata, e in caso di risposta B deve essere inserito all'interno della routing table.

La Mainline DHT prevede un sistema di messaggi, detto *protocollo KRPC* (*Key Remote Procedure Call*), attraverso il quale i nodi comunicano all'interno delle DHT. I messaggi possono essere di quattro tipologie:

ping per poter controllare se un nodo è ancora connesso o no;

trasmissione più elevate.

find_node per avviare una ricerca delle informazioni per poter contattare un nodo specifico;

get_peers per trovare dei peer per il torrent che si sta scaricando;

announce_peer per annunciare agli altri nodi che il client è un peer valido per un dato torrent.

Si veda più avanti (paragrafo 4.5) per una descrizione più approfondita di questo protocollo.

4.3 Funzionamento

Come detto, lo scopo della DHT è trovare nuovi peer per un torrent che si sta scaricando. Del torrent, l'unica cosa che ci serve è il suo infohash, ovvero un identificativo univoco di quel torrent. Si suppone di aver già un insieme di collegamenti verso altri nodi (una routing table piena), di cui si possiedono gli ID e di cui si possano misurare la distanza da noi stessi e dall'infohash del torrent.

Quando si vogliono trovare dei peer per un torrent, si cercano i nodi più vicini ad esso nella routing table, mandando poi a tutti loro una richiesta *get_peers* (Fig. 6-a). Se i nodi non possiedono peer per quel torrent, essi inviano i loro nodi più vicini al torrent fornito nella loro routing table (Fig. 6-b).

Se vengono forniti dei nodi, allora si ripete continuamente la richiesta di *get_peers* finché non si riescono a trovare altri nodi più vicini al torrent dato (Fig. 6-c). Questi nodi più vicini, rispondono o con altri nodi che già si conoscono o con dei peer per il torrent cercato (Fig. 6-d). Ad ognuno di questi nodi viene poi inviato un messaggio di tipo *announce_peer* annunciando che si è un peer valido per quel torrent (Fig. 6-e).

In questo modo alla lunga i nodi più vicini ad un dato infohash si ricevono peer utili per quel torrent, dove poi i nodi che cercano quel particolare torrent saranno reindirizzati dagli altri nodi.

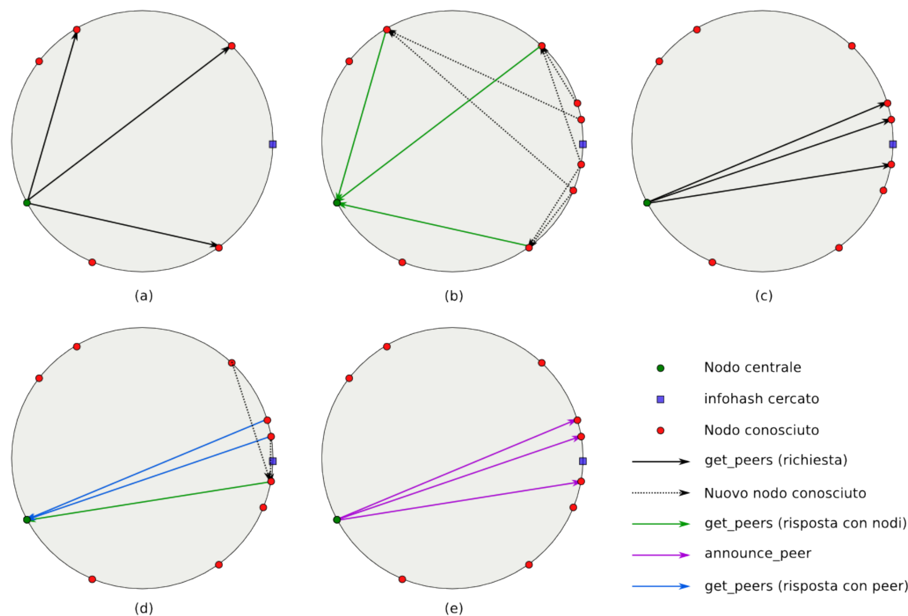

Figura 6: Schema di funzionamento della MainLine DHT.

4.4 Routing Table

La routing table è la struttura dati in cui vengono memorizzate le informazioni per poter contattare i nodi (come indirizzo IP, identificativo e porta UDP). Essa permette di aggiungere, rimuovere o cercare un nodo o un insieme di nodi dell'*overlay network* della DHT.

Nella Mainline DHT, la routing table ha dimensioni basse rispetto all'insieme complessivo dei nodi ($O(log(n))$), e viene organizzata nel seguente modo: ogni nodo nella sua routing table possiede molti nodi che secondo la metrica XOR hanno un identificativo molto vicino al suo, mentre ne possiede molto pochi di quelli molto distanti da lui.

La routing table utilizza come spazio delle chiavi quello a 160 bit. Interpretando gli identificativi come degli interi senza segno la DHT diventa in grado di coprire gli interi che vanno da 0 a $2^{160} - 1$.[14] La routing table è suddivisa in *bucket* (secchi), ognuno dei quali occupa una suddivisione degli interi del keyspace; ogni bucket può contenere solo un certo numero fisso di nodi, attualmente $K = 8$, prima di diventare pieno. Nella metafora di immaginare la routing table come un cerchio sulla cui circonferenza sono posti gli identificativi, i bucket possono essere pensati come fette del cerchio.

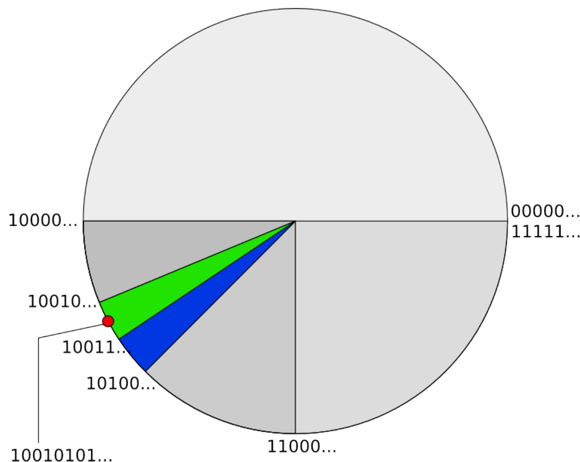

Figura 7: Esempio di routing table costruita intorno all'identificativo 10010101... Nello schema sono indicati i confini dei singoli bucket (il limite inferiore di un bucket coincide con il superiore del precedente). In verde è indicato il bucket principale, in blu il fratello del bucket principale.

Per costruzione, ogni bucket contiene tutti i nodi che si trovano tra il suo estremo inferiore e il suo estremo superiore, estremo inferiore incluso ed estremo superiore escluso, interpretando gli ID come interi senza segno. In formule,

$$n \in B \Leftrightarrow \min(B) \leq ID(n) < \max(B)$$

Dove n è un nodo e B è un bucket.

La suddivisione della routing table è eseguita in modo esponenziale, ovvero le dimensioni dei bucket, che sono sempre del tipo 2^n, con $3 \leq n \leq 159$, si dimezzano via via che ci si avvicina all'identificativo del client. Quindi esisteranno un bucket che copre un intervallo grande 2^{159}, uno 2^{158}, e così via, fino ad arrivare al bucket più piccolo, che contiene *myID*. Per poter fare in modo che i nodi coprano tutto lo spazio disponibile, ci dovranno essere due bucket di dimensione minima, e questi due saranno gli unici ad avere le stesse dimensioni.

L'ID attorno alla quale viene costruita la routing table, ovvero il nodo del client che sta eseguendo la MainLine DHT, è detto *myID*. Dei due bucket più piccoli, quello che contiene *myID* è detto bucket centrale (o anche *myBucket*), mentre l'altro bucket di dimensione minima viene detto fratello del bucket centrale.

[14] La specifica in realtà prevede di coprire gli interi che vanno da 0 a 2^{160}. Nella realtà con 160bit non è possibile rappresentare questo intervallo, in quanto esso prevede $2^{160} + 1$ interi.

Per comodità, negli algoritmi successivi verrà usata una specifica quantità, detta *ordine* del bucket. Essa consiste nel logaritmo in base 2 delle dimensioni del bucket. Ad esempio, il bucket che può contenere 2^{159} nodi avrà ordine $\log_2 2^{159} = 159$.

Inizialmente, la routing table viene suddivisa in due bucket che suddividono a metà lo spazio, comprendendo i due intervalli $0...2^{159}$e $2^{159}...2^{160} - 1$, in ID $[000...000, 100...000[$ e $[100...000, 111...111]$.

4.4.1 Aggiunta di un nodo

Non tutti i nodi di cui si è a conoscenza sono uguali: alcuni nodi sono migliori di altri, nel senso che rispondono spesso alle richieste del client e con tempi relativamente bassi. Ogni nodo possiede quindi uno stato, che può essere di tre tipi in base all'attività rilevata del nodo:

- *GOOD*, se si è avuto un contatto con il nodo negli ultimi 15 minuti;
- *QUESTIONABLE*, se si sono avuti contatti con il nodo negli ultimi 15 minuti;
- *BAD*, se il nodo fallisce alle richieste per un certo numero di volte di fila.

Questa distinzione nello stato dei nodi è utilizzata quando un nodo viene aggiunto alla routing table, nel modo descritto di seguito.

La particolare suddivisione della routing table, che diventa più concentrata attorno all'ID principale del nodo, deriva proprio dall'algoritmo utilizzato per aggiungere un nodo alla routing table. Esso è presentato di seguito:

Algoritmo 2 addNode(Node n_{new})

```
def Bucket b_old = il bucket in cui dovrebbe cadere il nodo n_new.
if ( b_old non è pieno ) then
    aggiungi il nodo n_new a b_old ;
else
    if (b contiene myID ) then
        dividi a metà b_old, creando due nuovi bucket b_inf e b_sup;
        ripartisci i nodi contenuti in b_old e il nuovo nodo n_new tra b_inf e b_sup ;
    else
        if ( tutti i nodi in b_old sono GOOD) then
            scarta il nodo n_new;
        if ( b_old contiene un nodo BAD n*) then
            rimpiazza n* con n_new;
        if( b_old contiene dei nodi QUESTIONABLE) then
            manageQuestionableNodes(n_new,b_old);
```

L'algoritmo in sè è piuttosto semplice: se il bucket relativo al nodo non è pieno, si aggiunge il nodo n al bucket b_{old} (che per gli scopi dell'algoritmo può essere pensato come una lista o un array di nodi). Se il bucket è pieno e ci si trova nel *myBucket* si tiene da parte n e si divide myBucket in due, ripartendo i nodi. Quindi se un nodo cade vicino a *myBucket* è più alta la

probabilità di inserirlo nella routing table. I nodi più distanti da *myID*, invece, vengono inseriti nella routing table se e solo si deve rimpiazzare un nodo *BAD* al suo interno, migliorando la stabilità della DHT, in quanto si rimpiazzano nodi inaffidabili con nodi considerati buoni. In questo modo, dividendo il bucket solo se ci si trova in *myBucket*, la routing table si concentra progressivamente intorno a *myID*, come si vede dalla figura seguente:

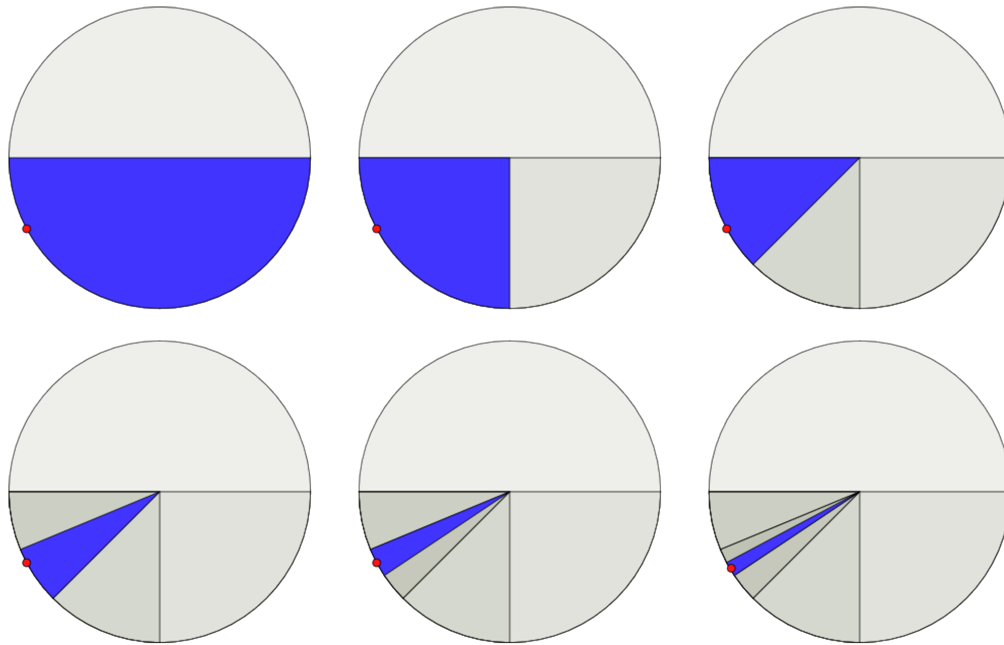

Figura 8: Esempio di progressiva suddivisione della routing table attorno all'ID 10010101... In blu è indicato il bucket principale.

Nel caso all'interno del bucket pieno non abbia nodi *BAD* ma solo nodi *QUESTIONABLE* allora si dovrà utilizzare un altro algoritmo, *manageQuestionableNodes*, per gestirli. L'algoritmo manda un messaggio `ping` a tutti i nodi di tipo *QUESTIONABLE* nel bucket; se tutti rispondono, diventano tutti *GOOD* e il nuovo nodo viene scartato. Mentre se uno fallisce due `ping` consecutivi viene rimpiazzato con il nuovo nodo.

Algoritmo 3 manageQuestionableNodes(Node n_{good}, Bucket b)

def qList = i nodi marcati come *QUESTIONABLE* in b.
for(Node $n_{questionable}$ in qList)
  PING($n_{questionable}$);
  if ($n_{questionable}$ risponde) then
    imposta lo stato di $n_{questionable}$ a *GOOD*;
    rimuovi $n_{questionable}$ da qList e passa al prossimo nodo;
  else
    PING($n_{questionable}$);
    if ($n_{questionable}$ risponde) then
      imposta lo stato di $n_{questionable}$ a *GOOD*;
      rimuovi $n_{questionable}$ da qList e passa al prossimo nodo;
    else
      rimuovi $n_{questionable}$ da b e aggiungi n_{good};

Questo processo garantisce che la routing table alla lunga sia piena solo di nodi di tipo *GOOD*, che rispondono velocemente alle richieste che gli vengono inviate.

4.4.2 Ricerca dei nodi più vicini ad un dato nodo (*Algoritmo FB*)

Oltre all'aggiunta dei nodi, come si vedrà più avanti per la DHT è essenziale conoscere nella routing table un certo numero di nodi più vicini a quello dato, di solito gli 8 nodi più vicini.

Un algoritmo molto semplice per realizzare ciò potrebbe essere un algoritmo che ordina secondo la metrica XOR tutti i nodi della routing table e poi prendendo il primi 8 membri del risultato dell'ordinamento. L'efficienza di questo algoritmo dipende dal tipo di ordinamento usato, ma non potrà comunque essere inferiore ad un $\Omega(n \log n)$, dove n è il numero di nodi all'interno della routing table.

Sfruttando però la particolare struttura della routing table, è possibile aumentare l'efficienza dell'algoritmo. Di seguito verrà descritto un particolare algoritmo sviluppato sulla Mainline DHT dal gruppo Torrent di PariPari, chiamato *Algorimo FB (Forward-Back)*.

Prima di passare a descrivere l'algoritmo, si introducono alcuni concetti di base per garantire una comprensione migliore. L'algoritmo prende in ingresso un identificativo generico che sarà chiamato in questo paragrafo *qID*; il suo bucket sarà chiamato *qBucket*. Il bucket principale sarà chiamato *myBucket*, mentre l'ID attorno a cui è costruita la routing table verrà chiamato *myID*. Il bucket di pari dimensioni a *myBucket* sarà chiamato fratello oppure *brotherBucket*. In generale per ogni bucket sarà scelto un particolare elemento che lo rappresenta, ovvero l'elemento minimo di quel bucket: ad esempio, per il bucket evidenziato in verde nell'immagine sottostante, il rappresentante sarà l'ID 101000000000. In questo paragrafo si è scelto di rappresentare la DHT non nella consueta rappresentazione a torta, ma in maniera lineare, in modo da visualizzare meglio un numero maggiore di bucket. Si noti che gli ID utilizzati sono a 12 bit invece che i consueti 160: per la descrizione dell'algoritmo, dato che per esso servono solo i bit più significativi degli ID, è irrilevante sapere quali sono i 148 bit rimanenti. Li si può immaginare sempre a 0 per comodità.

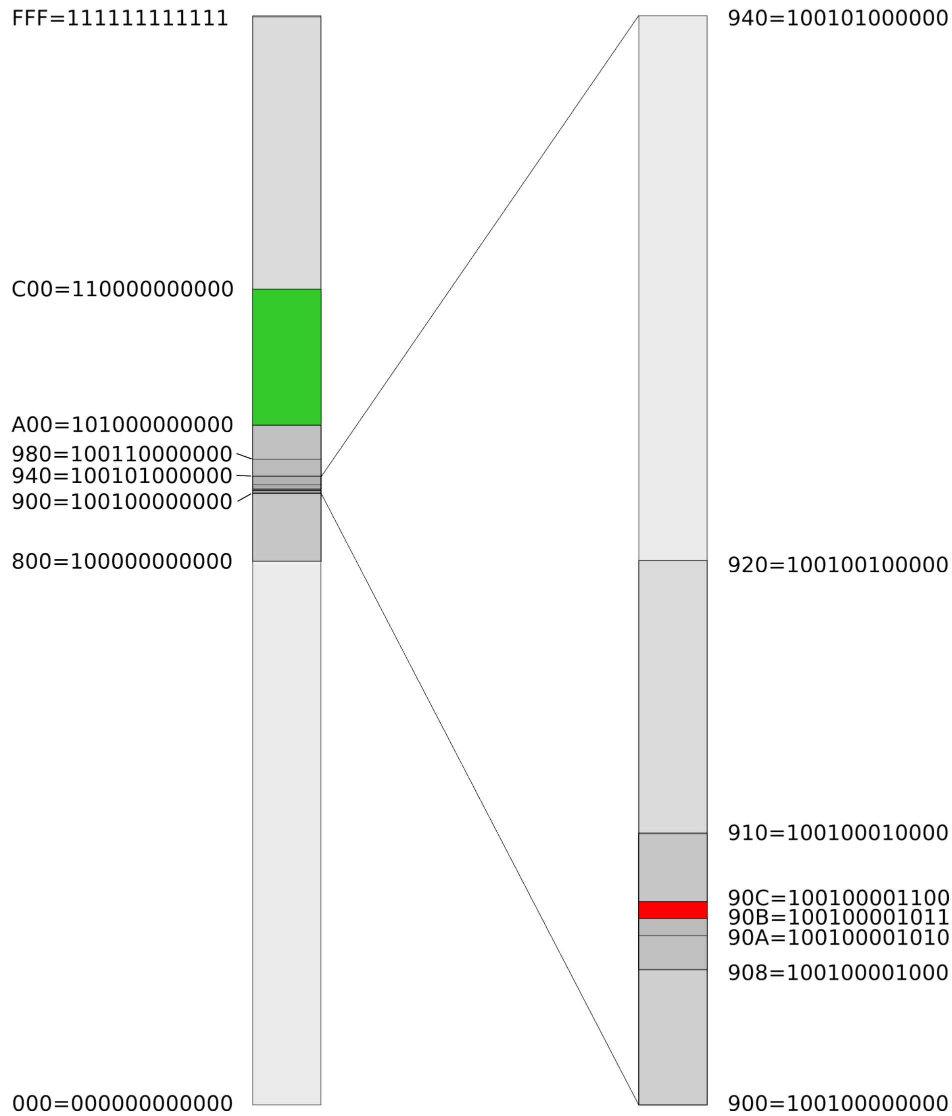

Figura 9: Rappresentazione di una DHT in modo lineare. In rosso è rappresentato il bucket principale. In verde è rappresentato *qBuck*, utilizzato più avanti. Gli ID sono rappresentati sia in binario che in esadecimale.

Si definisce inoltre un'altra quantità significativa dei bucket; oltre all'ordine $ord = log_2(\max(b) - \min(b))$ di definisce il numero di bit significativi di un bucket: esso è il numero di cifre del più lungo prefisso comune a tutti i nodi. Ad esempio, per il bucket in verde evidenziato in figura gli ID occupati vanno da 101000000000 a $110000000000 - 1 = 101111111111$ (l'estremo superiore non appartiene al bucket): il prefisso comune a tutti è 101, e quindi i bit significativi *bits* di quel bucket sono 3. Si può verificare che tra ordine e bit significativi esiste la relazione $bits = 160 - ord$ per ogni bucket nella routing table.

Algoritmo Per poter sfruttare la suddivisione della figura del paragrafo precedente si prendano:

$myID = 100100001011$ $qID = 101001011111$

Quindi *qBucket* sarà il bucket evidenziato in verde e *myBucket* quello evidenziato in rosso. Ora, preso un qualsiasi altro nodo *xNode* con identificativo x della DHT, posso avere tre casi (d denota la distanza di *xNode* da *qID*):

1. Il nodo *xNode* cade nello stesso bucket di q. Allora esso è a distanza $d = x \oplus qID = 101... \oplus 101... = 000...$, da cui $d < 2^{157}$. Questo succede perché se il nodo *xNode* cade nel bucket di *qID*, *qBucket*, allora avrà sicuramente i primi tre bit significativi posti a 101, per come sono stati definiti.

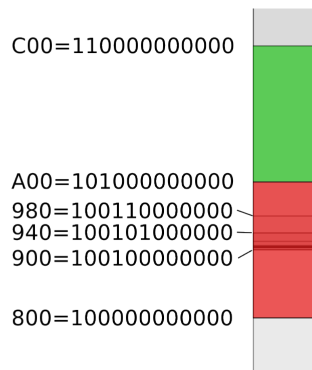

Figura 10: Particolare della DHT presentata precedentemente. Regione simmetrica (in rosso) di *qBucket* (in verde).

2. Il nodo *xNode* cade nella regione simmetrica al bucket di *qID*. Questa regione può essere definita come la regione di dimensione pari al bucket di qID, affiancata ad esso in cui cadono tutti i bucket di ordine inferiore a quello di qID. Nel caso specifico, si può notare che tutti i bucket della regione per costruzione hanno come prime cifre significative comuni 100, il che porta a $d = x \oplus qID = 001...$, da cui $2^{157} \leq d < 2^{158}$.

3. Il nodo *xNode* cade in un qualsiasi altro punto non considerato nei casi precedenti, ovvero nei bucket più grandi. Questi bucket per costruzione non possono avere le prime cifre significative uguali a quelle di *qBucket*, e non possono differire solo dell'ultima (altrimenti x cadrebbe nel simmetrico di *qID*). Quindi almeno una delle prime cifre diverse dall'utlima significativa è diversa, il che porta a dire che questi sono i nodi più distanti da *qID*. Infatti, nel caso specifico, $x \oplus qID$ può essere del tipo 1... o 01..., da cui $d \geq 2^{158}$.

Queste considerazioni possono essere generalmente estese ad un ID qualsiasi in un bucket qualsiasi. Detto *ord* l'ordine del bucket, in un qualsiasi ID appartenente a quel bucket partizionerà lo spazio delle chiavi in tre regioni, una con $d < 2^{ord}$, una con $2^{ord} \leq d < 2^{ord+1}$e un'altra con $d \geq 2^{ord+1}$. Sarà naturale quindi visitare le regioni in quest'ordine.

Queste considerazioni portano a definire l'algoritmo in maniera molto generale: come prima cosa si devono prendere i nodi di *qBucket*, poi andare a cercare verso i bucket di ordine inferiore e poi tornare indietro verso i bucket di ordine superiore (da qui il nome dell'algoritmo). Segue l'algoritmo in pseudocodice, supponendo di dover cercare K nodi (dove K è anche il numero massimo di nodi che può contenere un bucket):

Algoritmo 4 getClosestNodes(qID)

1. def *closestNodeList* = lista dei nodi in cui si aggiungeranno man mano i nodi più vicini trovati.
2. Cerca *qBucket*, aggiungi tutti i nodi di *qBucket* a *closestNodeList*;
3. Se ho già K nodi goto 13;
4. Percorri la routing table verso *myID* fino al prossimo bucket *nextBucket* nella regione simmetrica che contiene nodi più vicini a quello dato;
5. Aggiungi tutti i nodi di *nextBucket* a *closestNodeList*;
6. Se ho K nodi goto 13;
7. if $nextBucket \neq myBucket$ o $nextBucket \neq brotherBucket$ then goto 4;
8. def $backBucket = nextBucket$, ovvero il bucket da cui cominceremo ad andare all'indietro;
9. Percorri la tabella all'indietro, seguendo l'ordine crescente dei bucket, fino a trovare il prossimo bucket *backBucket* non visitato;
10. Aggiungi tutti i nodi di *backBucket* a *closestNodeList*;
11. Se ho K nodi goto 13;
12. if ordine di *backBucket* NON è quello di ordine massimo (159) then goto 9;
13. return $closestNodesList$.

I punti 1,2,3 consistono nel visitare *qBucket*. Poi dai punti 4 al 7 si ha il cosiddetto *Forward* dell'algoritmo, ovvero esso si muove verso bucket sempre più piccoli fino a raggiungere il bucket centrale della routing table (*myBucket*) o suo fratello (*brotherBucket*). Quando l'algoritmo raggiunge uno di questi due bucket, comincia la cosiddetta fase di *Back* dell'algoritmo (punti 8-12), nella quale esso ripercorre tutti i bucket da quello di ordine più basso a quello di ordine più alto, non visitando però i bucket già incontrati durante il *Forward* (*qBucket* incluso). La prima regione definita sopra si visita nei punti 1,2,3; la seconda regione si visita sia nel *Forward* che nel *Back* mentre la terza regione si visita nel *Back* dopo che si è incontrato *qBucket*.

Si passa ora a descrivere in dettaglio come l'algoritmo effettivamente percorre la routing table.

Forward Questa prima parte si compone di un ciclo che prende il bucket corrente ed il suo rappresentante e confrontandolo con *myID*: in questo modo si riescono a ricostruire gli estremi della regione simmetrica rispetto a *qBucket*.

Dati gli estremi di questa regione, per costruzione una metà di questa regione è un bucket, e l'altra metà è un'altra regione simmetrica (si veda la figura). Dato questo fatto, si cercherà di determinare in quale delle due metà dovranno cadere i nodi più vicini a *qID*. Nell'esempio in figura prendo la regione delimitata da 100000000000 e 101000000000. La regione viene suddivisa in due, $R_1 = [100000000000, 100100000000[$, che è un bucket di ordine 156, e da $R_2 = [100100000000, 101000000000[$, un'altra regione simmetrica. I due limiti inferiori sono i rappresentanti dei due bucket. Ora faccio la distanza tra *qID* e i due rappresentativi:

$101001011111 \oplus 100000000000 = 001001011111$
$101001011111 \oplus 100100000000 = 001101011111$

La minore delle due distanze è la prima delle due. Si noti che questo è vero per ogni elemento delle due metà: si prendano infatti i bit significativi dei bucket:

$d_1 = 101001011111 \oplus 1000... = 0010...$ $d_2 = 101001011111 \oplus 1001... = 0011...$

Da cui $d_1 = \delta(qID, i_1) < \delta(qID, i_2) = d_2, \forall i_1 \in R_1 \wedge \forall i_2 \in R_2$. Si è quindi definito in quale delle due metà si devono cercare i nodi più vicini a myBucket. Avremo due situazioni:

- la metà prescelta è il bucket: allora estrarrò i nodi da quel bucket:
- la metà prescelta è la regione simmetrica: chiamerò l'algoritmo ricorsivamente sulla regione simmetrica.

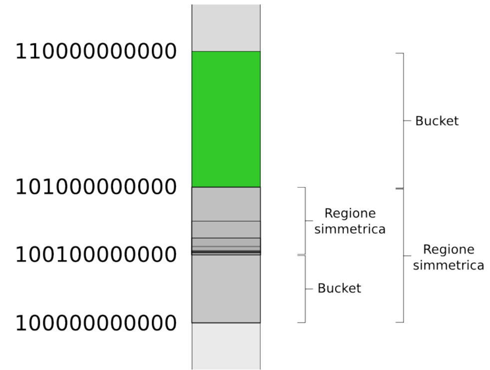

Figura 11: Struttura regione simmetrica/bucket.

Questo passo viene ripetuto finchè non raggiungo *myBucket* o *brotherBucket*: in questo caso, infatti da una parte avrò un bucket e la regione simmetrica dall'altro sarà (caso degenere) a sua volta un bucket. Questa è la condizione di terminazione dell'algoritmo in quanto essa ritorna sempre un bucket in cui cercare i nodi.

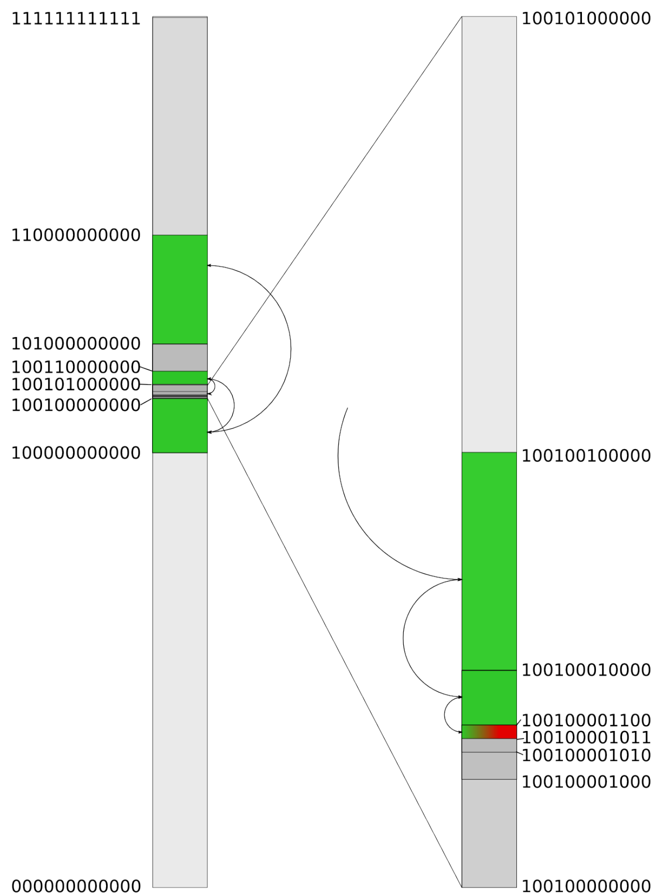

Figura 12: Bucket visitati durante la parte Forward. In verde sono indicati i bucket visitati.

Back La parte *Forward*, terminando con il bucket principale o con suo fratello, trova solo i bucket che hanno gli id con una distanza da *qID* una distanza compresa tra 0 e $\sim qID \oplus myID$. Questo a volte può non essere sufficiente, in particolare nel caso in cui *qID* è molto vicino a *myID* secondo la metrica XOR, oppure nel caso che *qID* coincida con *myID*, ricerca spesso richiesta in molte fasi del funzionamento della DHT. Bisogna quindi visitare i bucket non visitati durante il Forward. Questo lo si fa in maniera semplice: a partire da *myBucket* o *brotherBucket*, si visita intanto il corrispettivo fratello. Poi si prosegue attraversando i nodi della DHT da quello di ordine più basso a quello di ordine più alto. Per ogni bucket visitato *currentBucket*, si esegue questo semplice confronto:

$$qID \oplus min(currentBucket) < qID \oplus myID$$

Ovvero se la distanza dei nodi di *currentBucket* da qID è minore di $qID \oplus myID$, per quanto detto prima ci si trova sicuramente in uno dei bucket visitati nel *Forward*.

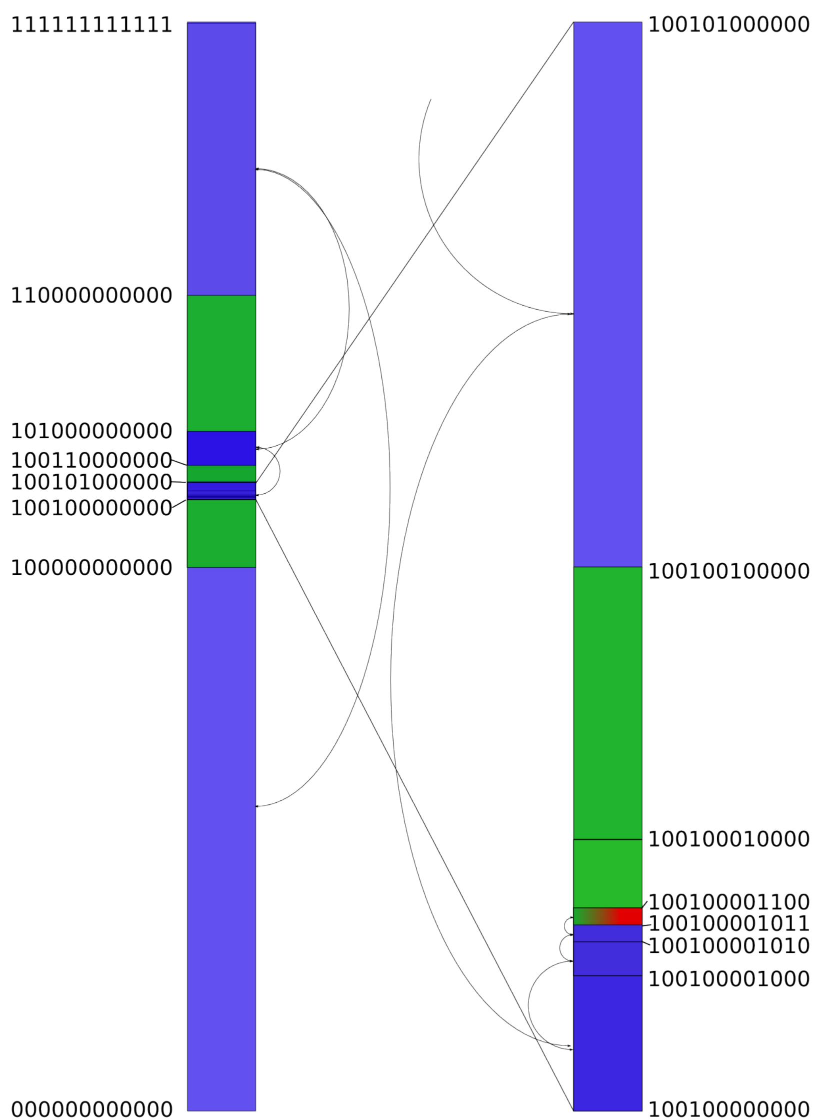

Figura 13: Bucket visitati durante la parte *Back*. In verde sono indicati i nodi visitati nel *Forward*, in blu quelli visitati durante il *Back*.

4.5 Protocollo KRPC

La Mainline DHT si avvale di un sistema di messaggi, detto protocollo *KRPC*. Il sistema, come dice il nome stesso, è di tipo RPC *(Remote Procedure Call*), ovvero i messaggi inviati tramite questo protocollo causano l'attivazione di una parte di codice del programma (procedura) su un altro computer della rete. I messaggi sono dei dizionari tradotti in *bencode*[15] e inviati tramite UDP. I messaggi hanno sempre due chiavi: $''t''$ e $''y''$. La prima contiene il cosiddetto

[15] Il bencode è un sistema di codifica delle informazioni utilizzato dal protocollo BitTorrent.

transaction ID, ovvero l'identificativo della transizione, che può essere utilizzato per correlare tra loro vari messaggi in entrata e in uscita.

La seconda, $''y''$, identifica il tipo del messaggio: può assumere tre valori, $''q''$, $''r''$, o $''e''$. In base al valore di y, altre chiavi vengono aggiunte al messaggio:

- $''q''$ (Fig. 14-b) indica un messaggio di tipo ***query***, ovvero un messaggio che viene inviato per chiamare una procedura sul computer remoto. Le query contengono due chiavi in più, $''q''$ che indica la tipologia della query e $''a''$ (*arguments*) che contiene i parametri della query, a loro volta organizzati come chiavi-valore.

- $''r''$ (Fig. 14-c) indica un messaggio di tipo ***response***, ovvero una risposta ad un messaggio di tipo *query*. Le risposte contengono una sola chiave in più, $''r''$, che contiene i parametri della risposta.

- $''e''$ (Fig. 14-d) indica un messaggio di tipo ***error***, ovvero un errore ad una query inviata o ad una risposta ricevuta. Anche gli errori hanno una sola chiave aggiuntiva, $''e''$, che contiene una lista di due elementi: il codice dell'errore generatosi e una stringa con la descrizione dell'errore.

Si veda l'immagine seguente per un riepilogo della struttura dei messaggi:

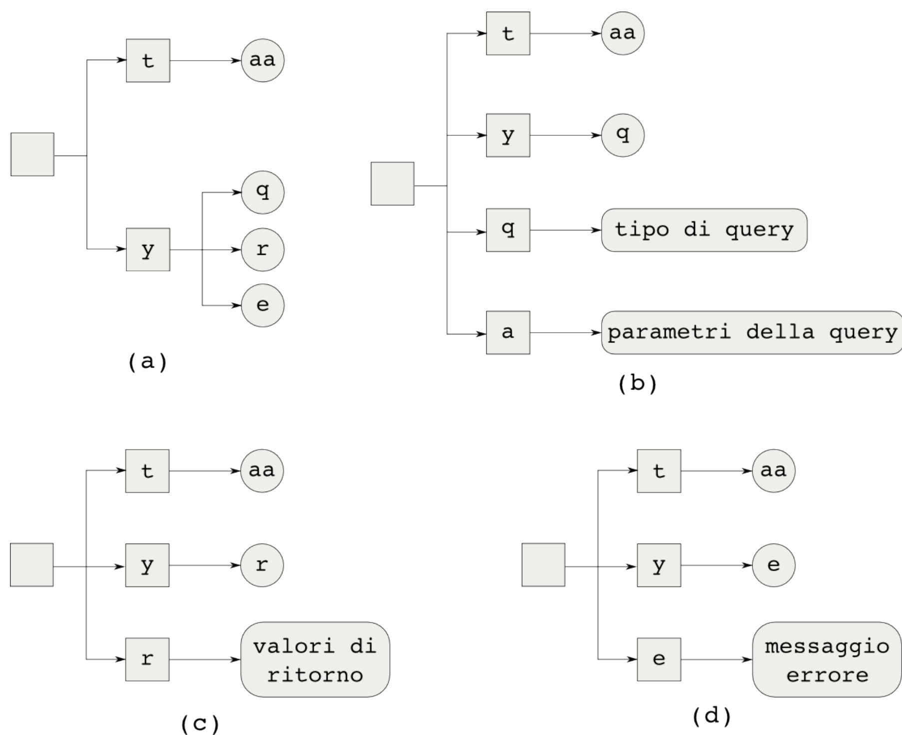

Figura 14: I vari tipi di messaggi del protocollo KRPC. Le icone rettangolari corrispondono a chiavi, quelle rotonde a valori. (a) Messaggio generico; (b) Messaggio di tipo ***query***; (c) messaggio di tipo ***response***; (d) messaggio di tipo ***error***.

Si esaminano ora caso per caso le varie tipologie di messaggio (supponendo A mandante la query, e B la risposta):

ping: è la query di base. Un nodo può essere pingato per poter aggiornare il suo stato da *BAD* o *QUESTIONABLE* di nuovo a *GOOD*. Il ping di A contiene al suo interno come parametro il proprio ID, mentre la risposta di B contiene all'interno l'identificativo di B stesso.

find_node: con questa query, si richiede ad un nodo di trovare le informazioni di contatto per un dato nodo usando il suo ID. In query, si mandano l'identificativo di A più l'ID cercato. In risposta si mandano gli 8 nodi più vicini all'ID dato nella routing table di B.

get_peers: con questa query si cercano dei peer per un determinato infohash di un torrent che si sta scaricando. Si possono verificare due casi: nel primo caso, il nodo contattato possiede dei peer per l'infohash cercato, e quindi risponde con quelli; nel secondo caso, se il nodo non possiede peer per quell'infohash esso risponde con i suoi 8 nodi più vicini ad esso.

announce_peer: questa query è utilizzata dal client per annunciare che si sta scaricando un particolare torrent su una certa porta. L'annuncio viene fatto ai nodi più vicini ad un determinato infohash nella DHT, in modo che essi abbiano delle informazioni da dare ai nodi che ne fanno richiesta.

5 Implementazione della MainLine DHT e della routing table

5.1 Il plug-in Torrent

Nel progetto PariPari, il protocollo BitTorrent è implementato nel plug-in *Torrent*. Come tutto il progetto PariPari, per programmare il plug-in è stato utilizzato il linguaggio di programmazione Java, il quale garantisce l'esecuzione di PariPari su tutte le piattaforme per le quali è disponibile una *Java Virtual Machine*.

Per poter funzionare, il modulo Torrent ha bisogno di altri tre plugin aggiuntivi:

- **Connectivity**, per poter avviare le connessioni con il tracker e gli altri peer della rete;
- **Local Storage**, per poter salvare i file su disco;
- **Logger**, per poter creare i file di log necessari agli sviluppatori.

L'interfaccia tra questi tre plug-in e Torrent avviene tramite il modulo `PluginSender`, il quale fornisce l'interfaccia verso il core di PariPari attraverso il quale è possibile richiedere le risorse della cerchia interna.

Prima di passare ad una descrizione del funzionamento nel dettaglio del modulo, si veda lo schema seguente, che illustra la struttura generica del plug-in:

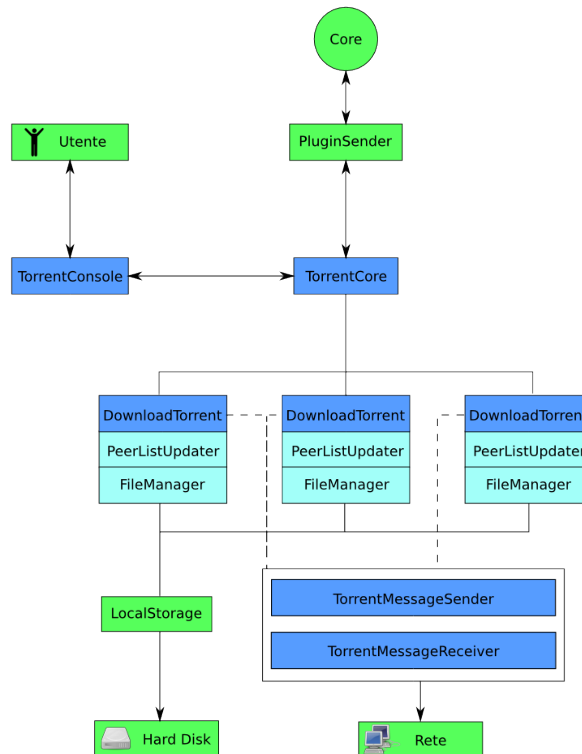

Figura 15: Schema essenziale del plug-in Torrent con tre torrent avviati in contemporanea.

Nello schema, in azzurro/blu sono indicate le classi proprie del plug-in torrent, mentre in verde sono indicate le risorse che vengono reperite tramite il modulo `PluginSender`.

Il centro del programma è la classe `TorrentCore`, che coordina il funzionamento dell'intero plug-in. I comandi immessi dall'utente nell'interfaccia (attualmente testuale) passano attraverso la classe `TorrentConsole` e da lì vengono mandate al `TorrentCore`, che si premura di eseguire l'operazione richiesta. Tra le operazioni possibili, vi sono avviare, aggiungere, fermare e cancellare un .torrent, oppure ottenere statistiche sul comportamento del plug-in (numero dei peer, eccetera).

La classe delegata alla gestione di un singolo download è `DownloadTorrent`: esiste una istanza della classe per ogni download avviato nel `TorrentCore`. Il `TorrentCore` si occupa di inizializzare i due oggetti `TorrentMessageSender` e `TorrentMessageReceiver` e a passarne un riferimento ad ogni oggetto `DownloadTorrent`; i due oggetti servono come dice il nome a interfacciarsi con la rete e a recapitare o ricevere i messaggi del Peer Protocol dagli altri peer.

Ogni istanza di `DownloadTorrent` possiede due oggetti: un `FileManager`, che garantisce che i pezzi una volta scaricati vengano immediatamente salvati su disco; e un `PeerListUpdater`, che attraverso la rete svolge la funzione di effettuare le richieste HTTP al tracker.

Oltre alle classi illustrate, sono presenti altre classi ausiliarie, come ad esempio la classe `Peer`, i cui oggetti presentano all'interno tutte le caratteristiche proprie che servono ad un peer, come stato, indirizzo ip, porta, eccetera.

Per costruire un oggetto `DownloadTorrent`, è inoltre necessaria la classe ausiliaria `TorrentFile`, che fornisce un accesso a tutte le caratteristiche di un file .torrent. Essa viene costruita a partire dal dizionario bencode ricavato dal file .torrent.

Esistono inoltre anche classi ausiliarie per gestire i pezzi (classe `Piece`) e il bencode (`BDecoder`, `BEncoder` e `BEValue`), oltre ad una classe `Utils` che raccoglie vari metodi ausiliari utilizzabili all'interno delle varie classi.

5.2 L'estensione alla MainLine DHT

Per l'estendere il plug-in Torrent al supporto alla Mainline DHT, sono state introdotte alcune modifiche al codice di alcune classi di base, oltre alla creazione di quattro nuovi pacchetti:

paripari.torrent.dht che contiene le classi principali di gestione della DHT;

paripari.torrent.dht.interfaces che contiene le interfacce relative al pacchetto precedente;

paripari.torrent.dht.messages che contiene le classi utilizzate per descrivere i vari tipi di messaggi del protocollo KRPC;

paripari.torrent.dht.messages.interfaces che contiene l'interfaccia generica utilizzata per descrivere i messaggi.

I primi due pacchetti contengono le classi fondamentali per la gestione della DHT:

- `DHTCore`, la classe che controlla le funzioni base della DHT, come la gestione della routing table;

- `RoutingTable` (interfaccia relativa `IRoutingTable`), la classe che rappresenta una routing table, fornendo un accesso alla struttura;
- `QuestionableNodesManagement`, utilizzato per gestire i nodi QUESTIONABLE secondo l'algoritmo 3 (si veda pagina 24);
- `DHTMessageSender` (interfaccia relativa `IDHTMessageSender`), che ricalca `TorrentMessageSender` (su UDP invece di TCP) per inviare i messaggi;
- `DHTMessageReceiver` (interfaccia relativa `IDHTMessageReceiver`), come sopra, per ricevere i messaggi.

Oltre a queste, nei pacchetti vi sono delle classi ausiliarie utilizzate per rappresentare gli elementi base della DHT e della routing table:

- `Node` (interfaccia relativa `INode`);
- `Bucket` (interfaccia relativa `IBucket`);
- `UnsignedInteger`, classe creata ad hoc per la gestione degli interi a 160 bit senza segno: esso è rappresentato come un array di `boolean`, sul quale sono state implementate varie funzioni come addizione, sottrazione, shift, concatenazione, eccetera;
- `DHTUtils`, che estende la classe `Utils` aggiungendo funzioni utili proprie della DHT.

Gli altri due pacchetti contengono invece i messaggi del protocollo KPRC:

- `DHTMessagePingQuery` e `DHTMessageAnnouncePingResponse`;
- `DHTMessageFindNodeQuery` e `DHTMessageFindNodeResponse`;
- `DHTMessageGetPeersQuery` e `DHTMessageGetPeersResponse`;
- `DHTMessageAnnouncePeerQuery` (la risposta, essendo identica e prevedendo gli stessi parametri di quella del ping, è stata accorpata in `DHTMessageAnnouncePingResponse`);
- `DHTMessageError`;
- `DHTMessage`, che fa da classe astratta generica da cui ereditano tutte le altre. Essa contiene metodi generici come la rivelazione del tipo e la gestione del transactionID.

Prima di descrivere il funzionamento generale, come in precedenza si veda lo schema generale delle modifiche apportate al plug-in Torrent:

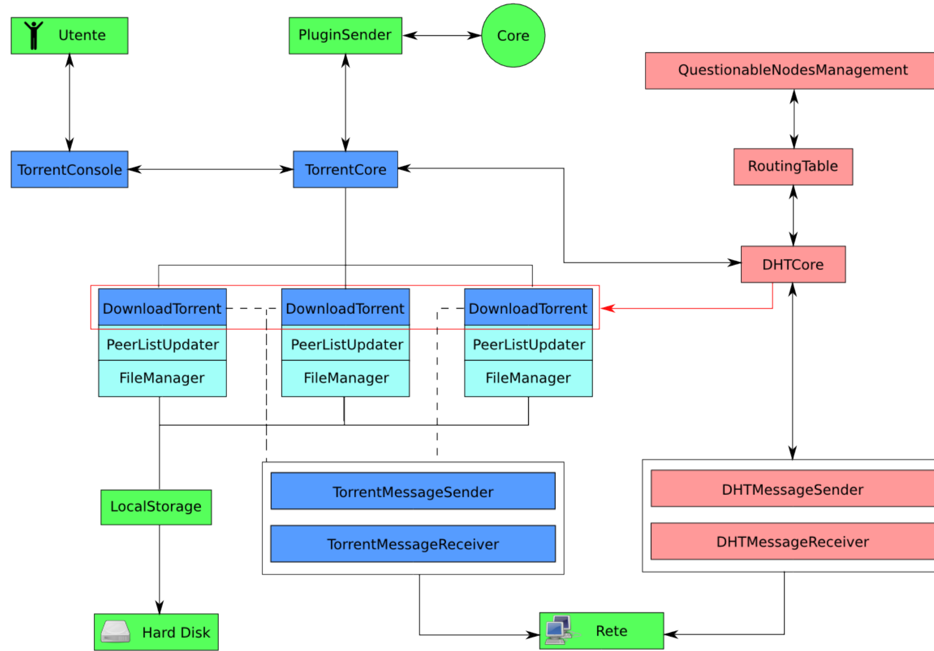

Figura 16: Schema del plug-in Torrent dopo l'introduzione della Mainline DHT.

L'oggetto `DHTCore` viene inizializzato da `TorrentCore` all'avvio, e svolge delle funzioni simili al suo omologo `TorrentCore`: anch'esso possiede un elenco delle istanze di `Download-Torrent` che supportano la DHT. Se i `DownloadTorrent` trovano un nuovo nodo per la DHT tramite lo scambio di messaggi PORT, inviano i dati a `DHTCore`: a sua volta, DHTCore appena riceve dei peer utili per il torrent relativo li inserisce nel relativo `DownloadTorrent`, tramite il relativo metodo `addPeer(String ip, int port)`.

Le classi `DHTMessageSender` e `DHTMessageReceiver` servono a gestire i messaggi. Esse lo fanno nel modo seguente: per prima cosa, si registra il nodo `Node` ad entrambe le classi, tramite il metodo `registerNode(INode n)`. La registrazione di solito viene fatta quando il nodo viene aggiunto alla `RoutingTable`. Un messaggio, per essere inviato al nodo Node, viene aggiunto al `DHTMessageSender` tramite il metodo `addMessageToNodeQueue(Node n, DHTMessage msg)`. Il messaggio viene aggiunto alla coda dei messaggi in uscita del nodo; `DHTMessageSender` scansiona tutti i nodi registrati e invia i messaggi presenti nelle loro code ad intervalli regolari. Il `DHTMessageReceiver` funziona in modo duale: una volta che il messaggio è stato ricevuto dai nodi registrati, esso viene tradotto in un oggetto di tipo `DHTMessage` e passato al `DHTCore`, che si premurerà di eseguire le istruzioni in esso contenuto.

5.3 La routing table

La routing table è essenzialmente un contenitore di nodi: si esamina quindi prima l'interfaccia `INode`, evidenziandone i metodi principali.

```
1  public interface INode {
2
3     public enum State {
4        GOOD, QUESTIONABLE, BAD
5     }
6
7     UnsignedInteger getID();
8     byte[] getIP();
9     int getPort();
10    boolean isComplete();
11
12    void setID(UnsignedInteger id);
13    byte[] getCompactInfo();
14    String getHostName();
15
16    UnsignedInteger distanceFrom(UnsignedInteger other);
17
18    [...]
19
20    long getLastUpdate();
21    void update();
22
23    INode.State getNodeState();
24    void setNodeState(INode.State state);
25
26    [...]
27 }
```

Listing 1: Interfaccia selezionata di INode.

Come si evince dall'interfaccia, il nodo possiede tre parametri principali: il suo identificativo, il suo indirizzo IP e la porta UDP su cui esso può ricevere i messaggi. Questi tre parametri possono essere ispezionati tramite i metodi `getID()`, `getIP()` e `getPort()`. Solo l'ID del nodo è modificabile tramite il metodo `setID(...)`, in quanto è permesso che il nodo per un certo periodo di tempo sia privo di ID: in questo stato il nodo è incompleto, mentre quando arriva l'ID il nodo diviene completo. Questa caratteristica viene ispezionata dal metodo `isComplete()`.

Il nodo prevede inoltre un metodo per calcolare la distanza da un altro nodo, il metodo `distanceFrom(...)`.

Altri parametri del nodo sono la variabile che mantiene lo stato, le cui possibilità sono definite dall'enumeratore `State`, e una variabile per ottenere quando è stato l'ultimo aggiornamento del nodo, definito dai metodi `getLastUpdate()` e `update()`. Oltre a questa classe, è stata creata l'interfaccia `IBucket`, che fornisce un accesso alle funzionalità richieste da un bucket. L'interfaccia implementa essenzialmente le caratteristiche di una lista, con i metodi per aggiungere, rimuovere e cercare i nodi.

```java
public interface IBucket {

  static final short K = IRoutingTable.K;
  static final short MAX_ORDER = 160;

  short getOrder();
  UnsignedInteger span();
  void removeNode(INode toRemove);
  void addNode(INode newNode);
  boolean isEmpty();
  void clear();
  int nodeListSize();

  UnsignedInteger getMinID();
  UnsignedInteger getMaxID();
  List<INode> getNodeList();

  long getLastChange();
  void setLastChange(long update);

  boolean contains(INode n);
  boolean containsID(UnsignedInteger b);

  INode getFirstGoodNode();
  INode getFirstQuestionableNode();
  INode getFirstBadNode();
  List<INode> getQuestionableNodes();

}
```

Listing 2: L'interfaccia IBucket.

Oltre ai metodi standard di accesso, ci sono i metodi di ispezione degli estremi del bucket (i quali ritornano un oggetto della classe `UnsignedInteger` rappresentante l'estremo), oltre ad un metodo `span()` per ricavare l'`UnsignedInteger` che rappresenta gli estremi del bucket, oltre al metodo `getOrder()` per ottenere l'ordine del bucket.

Oltre al metodo che verifica se un bucket contiene al suo interno uno specifico nodo, esiste un metodo containsID per verificare se l'ID fornito è compreso nello span dell'`IBucket`.

Ci sono poi metodi per ottenere quando il bucket è stato aggiornato per l'ultima volta (`getLastChange()`) e per ottenere i nodi di un particolare stato dei nodi in esso contenuto (ad esempio `getFirstQuestionableNode()`).

Si osservi ora l'interfaccia della routing table, `IRoutingTable`:

```java
public interface IRoutingTable {

  static final int K = 8;

  void addNode(INode node);
  boolean contains(INode n);
  boolean removeNode(INode n);
  int size();
  List<INode> getAllNodes();
```

```
10
11     List<INode> getClosestNodes(byte[] n);
12     List<INode> getClosestNodes(INode n);
13     List<INode> getClosestNodes(UnsignedInteger qID);
14
15     List<Bucket> getOutOfDateBucketsSince(long timeOfUpdate);
16     void updateBucketOf(INode n);
17
18  }
```

Listing 3: L'interfaccia IRoutingTable

Oltre ai metodi standard di inserzione e rimozione (righe 5-9) vi sono dei metodi getClosestNodes(...) per reperire i K nodi più vicini ad un ID dato (righe 11-13) e due metodi di update il cui funzionamento sarà esplicato più tardi quando si parlerà della gestione della routing table.

A livello di implementazione, la routing table è organizzata come una lista (`java.util.ArrayList`) di oggetti di tipo `IBucket`. Ogni bucket, al suo interno possiede due oggetti `UnsignedInteger` che indicano il minimo e il massimo del range di copertura del bucket. Oltre a questo, all'interno del bucket vi è una lista di nodi che viene mantenuta ordinata secondo l'ordine definito da `UnsignedInteger.compareTo(UnsignedInteger other)`, ovvero l'ID interpretato come un numero binario senza segno.

Oltre alla lista, all'interno della `RoutingTable` sono presenti tre oggetti aggiuntivi:*myID, myBucket e myIndex* che mantengono rispettivamente l'ID, un riferimento e l'indice al bucket contenente l'ID centrale della routing table. Questo viene fatto perchè il nodo centrale in molti algoritmi viene spesso richiamato, e quindi averne un riferimento migliora di molto l'efficienza degli algoritmi. Inoltre, dato che la routing table viene costruita soprattutto all'avvio del modulo (fase di bootstrap) dopo un po' essa non viene più modificata. Infatti, la routing table non vieta che ci siano dei bucket vuoti: essa può riempirsi e successivamente svuotarsi, ma la struttura dei bucket non viene mutata. In questo modo le tre variabili sopracitate dopo un po' non vengono più modificate.

5.3.1 Ricerca di un bucket (*getBucketIndexOf*)

La ricerca di un bucket dato un particolare identificativo è un'operazione chiave della routing table. Inizialmente, si era scelto un approccio del tipo ricerca binaria all'interno della routing table, ma alla fine si propone un altro algoritmo che sfrutta gli identificativi dei nodi. Questo processo si articola in due fasi:

- Recuperare l'ordine del bucket in cui cade l'identificativo
- Ottenere l'indice del bucket relativo a partire dall'ordine (come detto, questo è vero perché ad ogni bucket è associato un ordine, eccetto per l'ordine di myBucket).

Per recuperare l'ordine del bucket relativo, si osservi che il bucket in cui cade un nodo dipende da quanti bit ha in comune l'ID del nodo con myID. Più bit ha in comune il nodo con *myID*, più esso sarà vicino al nodo stesso e quindi più il suo ordine sarà basso. Il numero di bit comuni è il numero di zeri prima del primo uno nello XOR tra i due ID. Ad esempio, con

$myID = 100100001011$ $qID = 101001011111$

Dato che $myID \oplus qID = 001101010100$, prima del primo uno ci sono due zeri, e quindi avrò $ord = 159 - 2 = 157$.

Per ottenere l'indice del bucket relativo, invece, si osserva un altro fatto: osservando la routing table in maniera lineare, si vede che se il 159-ord bit di myID è posto a 1, il bucket cadrà nella metà verso lo 0 della routingTable, immaginando myBucket come linea ideale di demarcazione.

Dato che questo è vero fin dal primo bucket, si avrà che il numero di 1 a sinistra dell'1 trovato è proprio il numero di bucket che ci sono prima del bucket dell'ordine cercato. Sommando questa quantità a zero è possibile rislaire quindi all'indice del bucket. Si procede dualmente nell'altra metà. In codice:

```
protected int getBucketIndexOf(UnsignedInteger b) {
   if (myBucket.containsID(b))
      return myIndex;
   if (this.getMyBrother().containsID(b)) {
      if (routingTable.get((myIndex + 1) % this.getBucketSize()).getOrder() == myBucket.
          getOrder()) {
         return myIndex + 1;
      } else {
         return myIndex - 1;
      }
   }
   UnsignedInteger xor = myID.xor(b);
   return getIndexOfBucketOfOrder(Bucket.MAX_ORDER - 1 - xor.indexOfFirstOccurrenceOf(
       UnsignedInteger.ONE));
}

protected int getIndexOfBucketOfOrder(int order) {
   if (order < myBucket.getOrder())
      return -1;
   if(order == myBucket.getOrder())
      return myIndex;
   int significantBit = Bucket.MAX_ORDER - 1 - order;
   int resultIndex = 0;
   if (myID.get(significantBit) == UnsignedInteger.ONE) {
      resultIndex = 0;
      for (int i = 0; i < significantBit; i++) {
         if (myID.get(i) == UnsignedInteger.ONE)
            resultIndex++;
      }
   } else if (myID.get(significantBit) == UnsignedInteger.ZERO) {
      resultIndex = this.getBucketSize() - 1;
      for (int i = 0; i < significantBit; i++) {
         if (myID.get(i) == UnsignedInteger.ZERO)
            resultIndex--;
      }
   }
   return resultIndex;
}
```

Listing 4: I metodi getBucketIndexOf() e getIndexOfBucketOfOrder()

Si noti l'oggetto routingTable che è un'istanza di *ArrayList<Bucket>*, ovvero la routing table vera e propria.

Prestazioni Le prestazioni dell'algoritmo sono un $O(b)$, dove b è il numero di bit significativi rispetto a *myID* dell'ID cercato (ovvero il più lungo prefisso comune con *myID*). Di seguito si confrontano l'algoritmo proposto (a), la prima parte dell'algoritmo proposto più una ricerca lineare sui bucket (c) e una ricerca binaria (b).

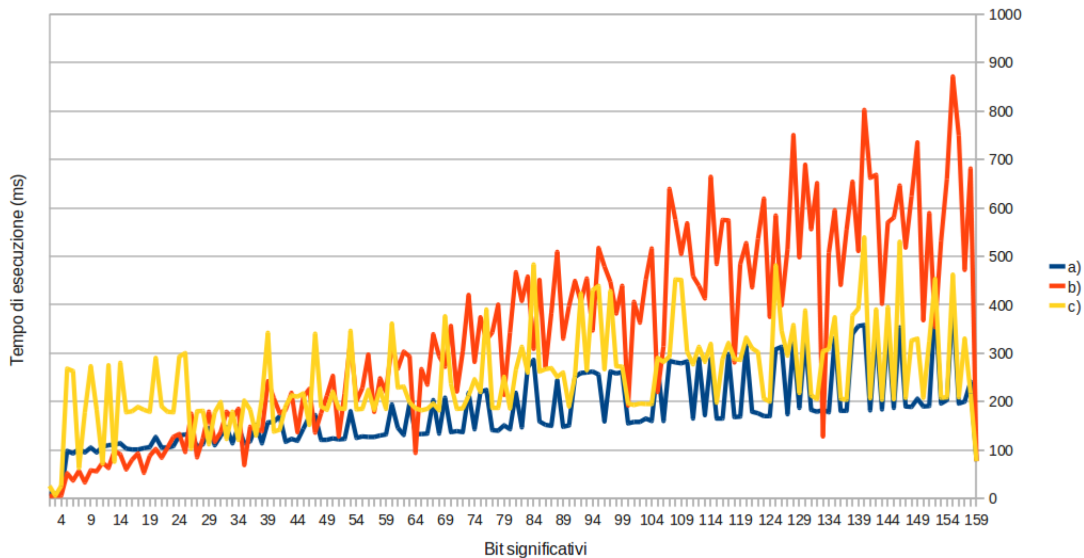

Figura 17: Algoritmi per la ricerca di un bucket a confronto (50000 esecuzioni).

Si nota che per i bucket di bit significativi bassi (<20) l'algoritmo di ricerca binaria è leggermente migliore, mentre gli algoritmi lineari (a) e (c) sono decisamente più efficienti per bucket con molti bit significativi (di ordine basso).

5.3.2 Aggiunta di un nodo (*addNode*)

Per aggiungere un nodo alla routing table, si utilizza l'algoritmo descritto al paragrafo 4.4.1 a pagina 22, che aggiunge il nodo *nodeToAdd* alla routing table. Il codice del metodo è presentato nel listato 5. Le prime righe (2-5) estraggono l'indice del bucket che dovrebbe contenere l'ID del nodo cercato (l'indice e il bucket sono nominati *bucketIndex* e *tempBucket* rispettivamente). Dopo si controlla che il nodo cercato non sia già contenuto nella routing table, o che non si stia tentando di inserire il nodo corrispondente a *myID*.

Dopo di questo, se il *tempBucket* non è pieno, si inserisce il nodo nel bucket e il metodo termina la sua esecuzione, mentre se è pieno esse esegue il blocco di codice alle righe 8-50.

```java
public void addNode(INode nodeToAdd) {
   int bucketIndex = getBucketIndexOf(nodeToAdd.getID());
   Bucket tempBucket = routingTable.get(bucketIndex);
   if (myID.compareTo(nodeToAdd.getID()) == 0 || tempBucket.contains(nodeToAdd))
      return;
   if (tempBucket.nodeListSize() < K) {
      tempBucket.addNode(nodeToAdd);
   } else if (tempBucket.nodeListSize() == K) {
   Bucket bucketToCheck = tempBucket;
      if (tempBucket == myBucket && tempBucket.getOrder() > 3) {
         Bucket newBucket = divide(tempBucket, bucketIndex + 1);
         if (newBucket.containsID(myID)) {
            myBucket = newBucket;
            myIndex++;
         } else {
            myBucket = tempBucket;
         }
         if (tempBucket.containsID(nodeToAdd.getID())) {
            if (tempBucket.getNodeList().size() == K) {
               bucketToCheck = tempBucket;
            } else {
               tempBucket.addNode(nodeToAdd);
               bucketToCheck = null;
            }
         } else {
            if (newBucket.getNodeList().size() == K) {
               bucketToCheck = newBucket;
            } else {
               newBucket.addNode(nodeToAdd);
               bucketToCheck = null;
            }
         }
      }
      if (bucketToCheck != null) {
         boolean allGoodNodes = true;
         for (INode n : bucketToCheck.getNodeList()) {
            allGoodNodes &= n.getNodeState() == INode.State.GOOD;
         }
         if (!allGoodNodes) {
            INode toRemove = bucketToCheck.getFirstBadNode();
            if (toRemove != null) {
               bucketToCheck.removeNode(toRemove);
               bucketToCheck.addNode(nodeToAdd);
            } else {
               List<INode> questionables = bucketToCheck.getQuestionableNodes();
               qnm.addTask(questionables, nodeToAdd, bucketToCheck);
            }
         }
      }
   }
}
```

Listing 5: Il metodo addNode().

In questo blocco il codice è suddiviso in due aree: il codice che viene eseguito nel caso il bucket in cui ci si trova sia proprio *myBucket*, e il codice che viene eseguito per la gestione degli stati dei nodi interni al bucket.

Nella prima parte (righe 10-33), per prima cosa si possa ancora dividere la routing table: se l'ordine di *myBucket* è minore o uguale a 3, significa che il bucket non può contenere gli 8 nodi previsti dalla specifica e quindi esso non può essere un bucket valido. Se è possibile dividere, il *tempBucket* viene diviso in due parti dal metodo `divide(...)`, *tempBucket* e *newBucket*, quelli che nell'algoritmo in pseudo codice erano stati chiamati b_{inf} e b_{sup}, rispettivamente.

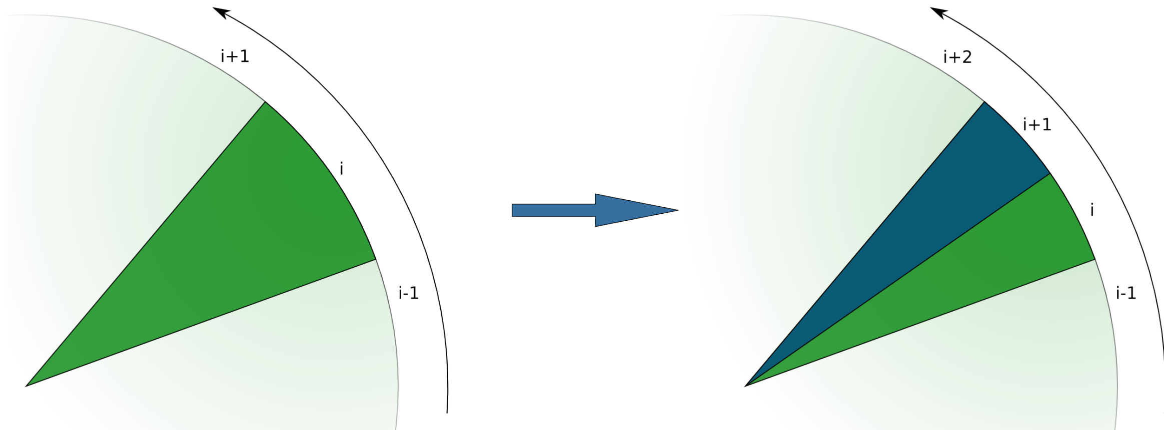

Figura 18: Effetto del metodo `divide(...)`. La freccia indica il verso degli indici dei bucket crescente. In verde è indicato *tempBucket*, in blu *newBucket*.

Poi, alle righe 12-17, viene controllato quale tra i due nuovi bucket creati è il nuovo *myBucket* e il riferimento viene aggiornato; inoltre, si aggiorna *myIndex*: esso viene incrementato se il nuovo *myBucket* è *newBucket*. Nella prima parte, infine, si controlla che tutti i nodi contenuti in *tempBucket* non siano finiti tutti in uno solo dei due nuovi bucket (righe 18-32), ovvero uno è pieno e l'altro è vuoto: se questo avviene, e il nuovo nodo cade nel bucket pieno esso sarebbe il K+1-esimo nodo, e questo non è possibile; in questo caso si passa il testimone alla seconda parte dell'algoritmo, che gestisce il caso in cui si debba inserire un K+1-esimo nodo in un bucket e questo non sia divisibile ulteriormente.

La seconda parte (righe 34-48) per prima cosa scansiona tutti i nodi interni al bucket e controlla se tutti siano posti a *GOOD*; se questo avviene, il blocco seguente (righe 39-48) non viene eseguito, il metodo termina e il nodo non viene aggiunto, come da specifica. Se il blocco viene eseguito, allora *bucketToCheck* contiene o dei nodi *QUESTIONABLE* o dei nodi *BAD*. Per prima cosa, si controlla se contiene un nodo *BAD* con il metodo `getFirstBadNode()` di `IBucket`, e se questo viene trovato (`toRemove` non viene posto a `null`) esso viene rimosso e il nuovo nodo viene inserito.

Infine, se vengono trovati dei nodi *QUESTIONABLE*, il metodo passa il testimone a qnm, ovvero l'istanza della classe `QuestionableNodesManagement` presente nella classe `RoutingTable`. Essa si occupa di gestire i nodi *QUESTIONABLE*, dato che questo richiede un tempo rispetto all'esecuzione di `addNode` molto elevato, in quanto deve attendere i tempi di

risposta ai ping dei nodi *QUESTIONABLE*. Si è scelto quindi di delegare il tutto ad un thread aggiuntivo.

5.3.3 Algoritmo FB (*getClosestNodes*)

L'ultimo algoritmo principale trattato riguarda la ricerca dei nodi più vicini ad un dato nodo. Per comodità, si divide la trattazione in due parti, una sulla parte *Back* e una sulla parte *Forward*. Per lo pseudocodice dell'algoritmo, si veda l'algoritmo 4 a pagina 27.

Segue il codice della parte Forward, impostato per una ricerca degli 8 nodi più vicini ad un identificativo qID della classe `UnsignedInteger`:

```
1  public List<INode> getClosestNodes(UnsignedInteger qID) {
2     List<INode> result = new ArrayList<INode>(K);
3     XORComparator comp = new XORComparator(qID);
4
5     int qIndex = this.getBucketIndexOf(qID);
6     IBucket qBucket = routingTable.get(qIndex);
7     List<INode> list = new ArrayList<INode>(qBucket.getNodeList());
8     Collections.sort(list, comp);
9     for (int i = 0; i < list.size(); i++) {
10       if (list.get(i).getNodeState() == INode.State.GOOD)
11          result.add(list.get(i));
12    }
13
14    IBucket nextBucket = qBucket;
15    while (result.size() < K && nextBucket != myBucket && nextBucket != this.getMyBrother()) {
16       UnsignedInteger bottom = nextBucket.getMinID();
17       UnsignedInteger top = nextBucket.getMaxID();
18       boolean firstDifferentBit = bottom.get(myID.xor(bottom).indexOfFirstOccurrenceOf(
             UnsignedInteger.ONE));
19       UnsignedInteger bucketSpan = nextBucket.span();
20       if (firstDifferentBit == UnsignedInteger.ZERO) {
21          UnsignedInteger newTop = bucketSpan.add(top);
22          nextBucket = getNextBucket(top, newTop, qID);
23       } else {
24          UnsignedInteger newBottom = bottom.subtract(bucketSpan);
25          nextBucket = getNextBucket(newBottom, bottom, qID);
26       }
27
28       [...]
29    }
30    [...]
```

Listing 6: la parte Forward del metodo getClosestNodes().

La prima parte del codice (righe 2-8) controlla il bucket in cui dovrebbe cadere qID, e ne estrae i nodi in ordine di distanza descrescente da *qID*. La lista *result* è costruita in modo che i nodi siano posti in ordine crescente, dal più vicino al più lontano da *qID*. Per fare ciò, viene creato un oggetto di tipo `XORComparator` (riga 3) che viene passato come parametro al metodo `Collections.sort(...)` assieme alla lista da ordinare.

Viene poi definita una variabile *nextBucket* (riga 14) che verrà aggiornata di volta in volta con il prossimo bucket da cui estrarre i nodi. Il blocco *while* seguente reitera l'aggiornamento

di *nextBucket* fino a che il *Forward* non termina (si raggiunge *myBucket* o suo fratello, estratto tramite il metodo `getMyBrother()`) o fino a che *result* non si riempie.

Di seguito (righe 15-28) l'algoritmo ricava gli estremi della regione simmetrica di *nextBucket*, e chiama su questa regione il metodo `getNextBucket(...)` che ritorna il nuovo bucket da cui estrarre dei nodi.

Per ricavare la regione simmetrica (righe 16-26), si guarda (riga 18) il primo bit discorde tra *myID* e *bottom* (bottom è l'estremo inferiore del bucket di qID), si prende l'indice e si guarda quel bit su *bottom*, ovvero il limite inferiore di *nextBucket*. Se questo bit è posto a zero, allora vuol dire che il corrispondente bit di *myID* è a 1, dato che sono discordi. Quindi, $myID > bottom$ e la regione simmetrica si estenderà da `top` fino a `top+nextBucket.span()`, dato che la regione simmetrica ha le stesse dimensioni di *nextBucket*. In caso contrario, essa si estenderà da `bottom-nextBucket.span()` a `bottom`.

La parte omessa in figura (riga 28) è molto simile alle righe 7-12: la lista del bucket viene clonata, ordinata e i suoi membri vengono inseriti in *result*.

Si osservi ora il metodo `getNextBucket(...)`:

```
1  private IBucket getNextBucket(UnsignedInteger bottomLimit, UnsignedInteger topLimit,
       UnsignedInteger searchedID) {
2     short order = DHTUtils.calculateOrder(bottomLimit, topLimit);
3     if (order == myBucket.getOrder()) {
4        if (bottomLimit.compareTo(myBucket.getMinID()) == 0 && topLimit.compareTo(myBucket.
             getMaxID()) == 0)
5           return myBucket;
6        if (topLimit.compareTo(myBucket.getMinID()) == 0)
7           return routingTable.get(myIndex - 1);
8        if (bottomLimit.compareTo(myBucket.getMaxID()) == 0)
9           return routingTable.get(myIndex + 1);
10    }
11    UnsignedInteger average = DHTUtils.average(bottomLimit, topLimit);
12
13    boolean zeroSideContainsMyNode = (myID.compareTo(average) < 0 && myID.compareTo(bottomLimit
          ) >= 0);
14    boolean shouldGoToZeroSide = searchedID.get(IBucket.MAX_ORDER - order) == UnsignedInteger.
          ZERO;
15
16       if (zeroSideContainsMyNode) {
17       if (shouldGoToZeroSide) {
18          return getNextBucket(bottomLimit, average, searchedID);
19       } else {
20          return routingTable.get(this.getBucketIndexOf(average));
21       }
22    } else {
23       if (shouldGoToZeroSide) {
24          return routingTable.get(this.getBucketIndexOf(bottomLimit));
25       } else {
26          return getNextBucket(average, topLimit, searchedID);
27       }
28    }
29 }
```

Listing 7: Il metodo getNextBucket(...)

Il metodo è ricorsivo, e come parametri ha i limiti inferiori e superiori della regione simmetrica, *bottomLimit* e *topLimit*, oltre all'ID cercato, *searchedID*. Viene calcolato l'ordine di grandezza della regione compresa tra *bottomLimit* e *topLimit* e questo ordine viene confrontato con quello di myBucket: se il confronto ha successo, allora si è arrivati alla fine del *Forward*: *bottomLimit* e *topLimit* devono corrispondere ai confini o di *myBucket* o di suo fratello. Il primo blocco (righe 3-10), caso base della ricorsione, serve a riconoscere questa eventualità.

Viene calcolata il valore medio tra *bottomLimit* e *topLimit*, *average*, e vengono poi definite due variabili: *zeroSideContainsMyNode* e *shouldGoToZeroSide*. La prima viene posta a true se il lato verso lo zero (ovvero quello compreso tra bottomLimit e average) contiene *myID*: questo sarà il lato in cui ci sarà la regione simmetrica successiva, se non ci si trova nel caso base. Per la seconda variabile, si prende il bit significativo corrispondente all'ordine calcolato, e si guarda a cosa corrisponde su *searchedID*: se esso è a zero, allora gli ID più vicini a *searchedID* si troveranno dalla parte in cui lo XOR con *searchedID* dà il risultato minore, ovvero quella in cui il bit significativo è concorde con quello di *searchedID*. Quindi se questo bit è zero, allora dovrò andare verso lo zero, e viceversa.

L'ultima parte confronta le due variabili. Se esse sono concordi (righe 18,26) allora ci si dovrà muovere verso una regione simmetrica, mentre se sono discordi (righe 20, 24) la ricerca termina e il rispettivo bucket viene restituito.

Ora si veda la parte *Back*:

```
1  [...]
2  IBucket greatestBucket = (myID.get(0) == UnsignedInteger.ONE) ? routingTable.get(0) :
       routingTable.get(getBucketSize() - 1);
3  IBucket retBucket = nextBucket;
4  boolean haveToVisitMyBrother = true;
5  UnsignedInteger Xor = myID.xor(qID);
6  int currentOrder = myBucket.getOrder();
7  while (result.size() < K && retBucket != greatestBucket) {
8     boolean bucketToVisit = true;
9     if (haveToVisitMyBrother && retBucket == myBucket) {
10       retBucket = this.getMyBrother();
11       haveToVisitMyBrother = false;
12    } else if (haveToVisitMyBrother && retBucket == this.getMyBrother()) {
13       retBucket = myBucket;
14       haveToVisitMyBrother = false;
15    } else {
16       retBucket = routingTable.get(this.getIndexOfBucketOfOrder(currentOrder));
17       if ((retBucket.getMinID().xor(qID)).compareTo(Xor) < 0) {
18          bucketToVisit = false;
19       }
20    }
21    if (bucketToVisit) {
22       [...]
23    }
24    currentOrder++;
25    }
26 return result;
27 }
```

Listing 8: La parte finale del metodo getClosestNodes(...).

Nella prima parte (righe 2-6) vengono impostate alcune variabili utili, come *retBucket*, che ha lo stesso utilizzo di *nextBucket* nella parte *Forward*. Si ricava anche il bucket più grande, *greatestBucket*. L'algoritmo parte da *currentOrder* e *myBucket* (o suo fratello) e risale attraverso i bucket secondo l'ordine crescente (*currentOrder++*, alla riga 24). Nel ciclo (righe) si analizza prima se *retBucket* è uguale a *myBucket* o suo fratello, e per prima cosa si visita l'opposto a questo: se$retBucket = myBucket$ si visita il fratello e viceversa. Questa operazione viene effettuata una sola volta, grazie alla flag *haveToVisitMyBrother*. Infine, per ogni ordine, si decide se visitare il rispettivo bucket secondo la formula già presentata:

$$qID \oplus min(retBucket) \geq qID \oplus myID$$

Se questa condizione è vera, il bucket viene visitato lasciando a true la variabile *bucketToVisit*, altrimenti questa viene posta a false (riga 18) e il bucket non viene visitato. Alla riga 22 è presente il solito blocco di codice che clona ordina ed estrae i nodi dal bucket cercato.

Efficienza L'efficienza dell'algoritmo è un parametro strettamente legato al fattore di carico della routing table. Esso è definito come:

$$f = \frac{N}{8 \cdot B}$$

Dove N è il numero di nodi presenti nella routing table, mentre B è il numero di bucket presenti. Questo fattore indica quindi in media quanto è pieno ogni bucket.

Naturalmente, per fattori di carico alti, l'algoritmo ha alte probabilità di fermarsi dopo una o due chiamate ricorsive della parte Forward, o addirittura nella prima iterazione se il primo bucket trovato è pieno. Nel caso migliore l'algoritmo è quindi $O(1)$. Il caso peggiore dell'algoritmo si avrà quindi per fattori di carico molto bassi, in generale con $f < 0.125$, quando la probabilità che ci siano dei bucket vuoti diventa considerevole.

Il caso peggiore si avrà quando tutti i bucket saranno vuoti ($f = 0$) e si cercherà un ID qualsiasi. In ogni caso, tutti i bucket saranno visitati. Quindi l'algoritmo avrà efficienza $O(b)$, con b uguale al numero di bucket presenti nella routing table.

Se invece la routing table non è vuota ($f > 0$) il caso peggiore si avrà quando i bucket di ordine più alto saranno pieni mentre quelli di ordine basso saranno vuoti, e si chiamerà l'algoritmo su *myID*. Si veda il grafico sottostante, dove l'algoritmo viene confrontato con un ordinamento della lista di tutti i nodi della routing table tramite il metodo `Collections.sort(...)`, e dal risultato dell'ordinamento si prendono i primi 8 nodi.

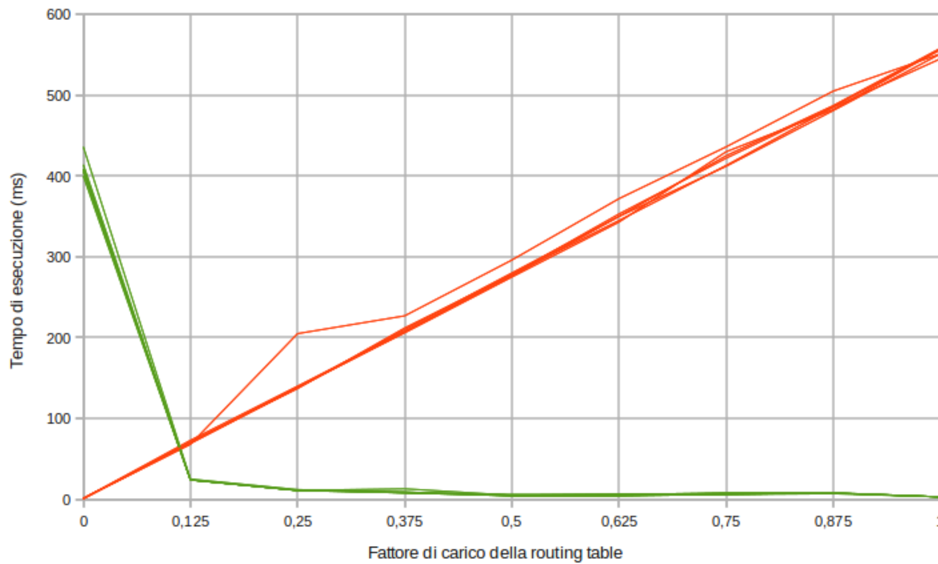

Figura 19: Confronto tra gli algoritmi FB (in verde) e ordinamento standard (in rosso) nel caso peggiore.

5.4 Gestione della routing table

La routing table, viene gestita dalla classe `DHTCore`, che coordina tutta l'estensione DHT. Di seguito si fanno alcuni esempi di come la routing table, rappresentata dall'oggetto chiamato *rt* della classe `RoutingTable`, viene gestita all'interno della DHT.

5.4.1 Bootstrap

La routing table inizialmente viene costruita e lasciata vuota dal `DHTCore`, fino a che non arrivano dei nodi validi a cui poter inviare un ping. Questo succede quando viene avviato un torrent contenente la chiave nodes, ovvero la chiave dei nodi cosiddetti di *bootstrap*.

Quando DHTCore riceve i nodi di *bootstrap*, ad essi viene inviato un messaggio di tipo `ping`, e quelli che rispondono vengono inseriti nella routing table. Se questa non è ancora stata riempita, da specifica, bisogna operare una ricerca di tipo `find_node` su noi stessi fino a che non si ottengono sempre gli stessi nodi.

Il metodo posto a questa operazione è `fillRT()`; esso manda un messaggio di tipo `DHTMessageFindNodeQuery` viene inviato ad ognuno degli 8 nodi più vicini a *myID*. Dopo l'invio dei messaggi, `DHTCore` viene messo in attesa per un certo periodo di tempo per dare il tempo ai nodi pingati di rispondere; si noti che il risultato della risposta viene inserito nella routing table in un'altra parte del codice, nel metodo `DHTCore.parseMessage(...)`, che riceve i messaggi dal `DHTMessageReceiver` ed esegue le loro direttive. Oltre a questo, nel metodo è presente un *securitycounter* che dopo un certo numero di esecuzioni smette di eseguire la ricerca, per evitare un tempo di riempimento della routing table troppo elevato.

```java
private void fillRT() {
   List<INode> tempList = null;
   List<INode> closestNodes = rt.getClosestNodes(myID);
   int securityCounter = 0;
   while (tempList.equals(closestNodes) && securityCounter++ < FIND_NODE_TRIES) {
      tempList = closestNodes;
      for (INode n : tempList) {
         dhtMessageSender.addMessageToNodeQueue(n, new DHTMessageFindNodeQuery(myID, myID));
      }
      try {
         Thread.sleep(FIND_NODE_TIMEOUT);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      closestNodes = rt.getClosestNodes(myID);
   }
}
```

Listing 9: fillRT().

5.4.2 Ricezione di un messaggio

In ricezione di un messaggio, `DHTMessageReceiver` attiva il metodo `parseMessage` della classe `DHTCore`, che si occupa di gestire il messaggio e di chiamare i metodi della routing table. Ad esempio, si veda questa parte del codice di `parseMessage(...)`, in cui viene gestita la ricezione di tipo `ping`:

```java
public void parseMessage(INode node, DHTMessage msg) {
   if (msg instanceof DHTMessagePingQuery) {
      dhtMessageSender.addMessageToNodeQueue(node, new DHTMessageAnnouncePingResponse(msg.
           getTransactionID(), myID));
   } else if (msg instanceof DHTMessageAnnouncePingResponse) {
      DHTMessageAnnouncePingResponse msgParsed = (DHTMessageAnnouncePingResponse) msg;
      if (!node.isComplete()) {
         node.setID(new UnsignedInteger(msgParsed.getAnsweringNodeID()));
         rt.addNode(node);
      } else {
         /* When a node in a bucket is pinged and it responds [...] the
            bucket's last changed property should be updated.
            if the node is added to the RT the update is automatic
         */
         rt.updateBucketOf(node);
      }
[...]
```

Listing 10: parte del metodo parseMessage(...) della classe DHTCore.

6 Conclusioni e stato dell'arte

L'aggiunta dell'estensione DHT ad modulo Torrent di PariPari è ad oggi in grande parte completata. La specifica della BitTorrent Foundation è stata seguita accuratamente, lasciando fuori solo alcune funzionalità minori; ad esempio, la routing table non viene salvata tra un esecuzione del client e l'altra.

Attualmente l'estensione è in fase di testing offline, in quanto l'interfaccia con la rete non è ancora completa: per questo, si attende che sia completa la nuova versione del plug-in Connectivity di PariPari, che consentirà un'interfaccia completa e semplice con lo scambio di dati via UDP.

Si prevede, in futuro, della creazione di una classe che simuli il comportamento della rete (*dummy*) in modo da porte testare in modo completo le funzionalità di questa estensione.

Il completamento di questa estensione consentirà anche la possibilità di completare il supporto ad alcune funzionalità di libtorrent, che non sono state implementate proprio perchè mancava questa estensione alla MainLine DHT.

Le possibili aggiunte e migliorie sono ancora molte: tuttavia, si spera che questa estensione, assieme alle altre del plug-in Torrent, possa contribuire ad incrementare la velocità in download e in upload dei file, per garantire una migliore esperienza di file sharing agli utenti.

7 Bibliografia

1. Brandon Wiley. *Distributed Hash Tables, Part I.* Linux Journal, 1° ottobre 2003. http://www.linuxjournal.com/article/6797

2. Peter Maymounkov, David Mazières. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* IPTPS 2002. http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf

3. Bram Cohen. *BEP 3: The BitTorrent Protocol Specification.* 10 gennaio 2008. http://www.bittorrent.org/beps/bep_0003.html

4. Andrew Loewenstern. *BEP 5: DHT Protocol.* 31 gennaio 2008. http://www.bittorrent.org/beps/bep_0005.html

5. Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. *Looking up data in P2P systems.* Communications of the ACM. Febbraio 2003. http://www.cs.berkeley.edu/~istoica/papers/2003/cacm03.pdf

Elenco delle figure

1 La struttura di PariPari, con evidenziate la cerchia interna ed esterna. 6
2 Schema dell'eXtreme Programming. In corsivo è indicata la suddivisione temporale delle varie fasi. 7
3 Il logo della BitTorrent, Inc. 9
4 Sistema client-server (a) e sistema p2p (b) a confronto. 13
5 Comunicazione tra i nodi A e B descritta nel testo. 18
6 Schema di funzionamento della MainLine DHT. 20
7 Esempio di routing table costruita intorno all'identificativo 10010101... Nello schema sono indicati i confini dei singoli bucket (il limite inferiore di un bucket coincide con il superiore del precedente). In verde è indicato il bucket principale, in blu il fratello del bucket principale. 21
8 Esempio di progressiva suddivisione della routing table attorno all'ID 10010101... In blu è indicato il bucket principale. 23
9 Rappresentazione di una DHT in modo lineare. In rosso è rappresentato il bucket principale. In verde è rappresentato *qBuck*, utilizzato più avanti. Gli ID sono rappresentati sia in binario che in esadecimale. 25
10 Particolare della DHT presentata precedentemente. Regione simmetrica (in rosso) di *qBucket* (in verde). 26
11 Struttura regione simmetrica/bucket. 28
12 Bucket visitati durante la parte Forward. In verde sono indicati i bucket visitati. 29
13 Bucket visitati durante la parte *Back*. In verde sono indicati i nodi visitati nel *Forward*, in blu quelli visitati durante il *Back*. 30
14 I vari tipi di messaggi del protocollo KRPC. Le icone rettangolari corrispondono a chiavi, quelle rotonde a valori. (a) Messaggio generico; (b) Messaggio di tipo *query*; (c) messaggio di tipo *response*; (d) messaggio di tipo *error*. 31
15 Schema essenziale del plug-in Torrent con tre torrent avviati in contemporanea. 33
16 Schema del plug-in Torrent dopo l'introduzione della Mainline DHT. 36
17 Algoritmi per la ricerca di un bucket a confronto (50000 esecuzioni). 41
18 Effetto del metodo `divide(...)`. La freccia indica il verso degli indici dei bucket crescente. In verde è indicato *tempBucket*, in blu *newBucket*. 43
19 Confronto tra gli algoritmi FB (in verde) e ordinamento standard (in rosso) nel caso peggiore. 48

Elenco degli algoritmi

1 getClosestNode(Key k) 15
2 addNode(Node n_{new}) 22
3 manageQuestionableNodes(Node n_{good}, Bucket b) 24
4 getClosestNodes(qID) 27