UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

PARIDHT: ACCELERAZIONE

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: *Nicola Celli*

ANNO ACCADEMICO 2009/2010

Alla mia famiglia.

Indice

Sommario

L'aumento negli ultimi anni della capacità di calcolo degli elaboratori e delle connessioni a banda larga ha portato alla diffusione di sistemi peer-to-peer (**P2P**, d'ora in poi) (cfr. [18]). È in questo ambito che *PariPari* si propone come un'applicazione innovativa, potenzialmente in grado di fornire all'utente molteplici servizi tramite un'unica piattaforma facile da usare e che garantisca l'anonimato.
Inoltre PariPari vuole essere un servizio decentralizzato, che quindi non dipende da nessun server. In una simile rete risulta essere problematica la pubblicazione di risorse che un utente desidera condividere, e il conseguente reperimento di queste. In risposta a tali esigenze si sono sviluppate varie soluzioni: attualmente la più diffusa consiste nell'utilizzo di una *tabella hash distribuita* (ovvero **DHT**).
In questa tesi si illustra **l'algoritmo del figlio prediletto**, il quale permette un più rapido reperimento di risorse dalla rete, ed è principalmente rivolta agli studenti che desiderano avere maggiori informazioni sulla DHT e, in particolar modo, a quelli che dovranno adoperarsi nel modulo DHT del progetto PariPari.

Introduzione

Nel presente lavoro verrà inizialmente introdotto il progetto PariPari descrivendone le finalità, l'architettura e l'organizzazione, ma senza soffermarsi nei dettagli realizzativi. Dopo di ciò verranno presentate le reti P2P e ne verrà brevemente analizzata la loro evoluzione fino a giungere alle DHT, che verranno approfondite maggiormente e delle quali verranno esposti due esempi, Chord (cfr. [39] e [19]) e Kademlia (cfr. [38] e [20]) che serviranno per una miglior comprensione degli argomenti trattati successivamente. Chord, infatti, è una rete abbastanza semplice, una delle prime DHT di terza generazione, ma soprattutto presenta degli elementi in comune con l'*algoritmo del figlio prediletto*. Mentre Kademlia è stata presa di riferimento al momento dell'implementazione del modulo DHT di PariPari.
Dopo questa introduzione generale si passerà ad analizzare il modulo DHT. Di quest'ultimo si esporranno le specifiche seguite al momento della realizzazione e verranno esaminate le classi che implementano la comunicazione fra i nodi connessi alla rete ed il protocollo che utilizzano.
Nel capitolo successivo si entrerà nel cuore di questa tesi, verrà infatti presentato come è stata implementata la ricerca di un determinato nodo nella DHT di PariPari, ma soprattutto verrà definito l'*algoritmo del figlio prediletto*, il cui obiettivo è di "*accelerare*" le ricerche all'interno della rete. Verranno inoltre analizzate tutte quelle classi che si occupano della realizzazione di quest'ultimo.
Nell'ultimo capitolo si valuterà ciò che si è ottenuto e si presenteranno alcune limitazioni dell'attuale implementazione illustrandone i possibili miglioramenti.

Capitolo 1

PariPari

PariPari è un'applicazione il cui obiettivo è:

- creare una rete serverless (attualmente basata su una variante di Kademlia (cfr. [38] e [20]));
- garantire l'anonimato dei suoi nodi;
- implementare un sistema di crediti intelligente e transitivo;
- essere multifunzionale fornendo i più comuni servizi disponibili su Internet.

1.1 Architettura

PariPari presenta una struttura modulare dove ciascun modulo (o plugin), pur essendo una struttura a sé stante, è in grado di cooperare con gli altri per mezzo dello scambio di messaggi. Vi è una gerarchia tra moduli: in particolare quello centrale è il *Core*(cfr. [3]), il quale si occupa di gestire il caricamento dei moduli stessi e di tutte le iterazioni tra questi. Altri moduli indispensabili per il funzionamento di PariPari, che uniti al core appartengono a quella che va sotto il nome di cerchia interna, sono i seguenti quattro:

- **connectivityNIO** (cfr. [4]): gestisce i socket e le comunicazioni tra vari nodi di PariPari.
- **credits** (cfr. [5]): suddivisi in crediti interni ed esterni, costituiscono la moneta di scambio per qualsiasi transazione tra moduli (crediti interni) e con gli altri nodi della rete (crediti esterni);

- **DHT** (cfr. [6])(distribuited hash table): definisce la struttura della rete stessa e permette la localizzazione delle risorse;

- **local storage** (cfr. [7]): si occupa delle operazioni che avvengono su disco ad opera di altri moduli, ad esempio lettura e scrittura di file di configurazione.

Ci sono poi i plugin della cerchia esterna, ciascuno dei quali offre un certo servizio e tra cui possiamo trovare:

- **torrent** (cfr. [8]): il cui obiettivo è fornire un client che implementi il protocollo BitTorrent (cfr. [21]) finalizzato alla distribuzione e condivisione di file nella rete;

- **mulo** (cfr. [9]): si propone di fornire le funzionalità di un qualsiasi client per la rete eDonkey2000 (cfr. [22]) (d'ora in poi ED2K);

- **VoIP** (cfr. [23] e [10])(Voice Over Internet Protocol): rende possibile effettuare una conversazione telefonica sfruttando una connessione Internet;

- **IRC** (cfr. [24] e [11])(Internet Relay Chat): permette la comunicazione istantanea (chat) fra due o più utenti;

- **GUI** (cfr. [25])(graphical user interface): fornisce l'interfaccia grafica;

- **Distributed Storage** (cfr. [12]): permette il salvataggio, la ricerca e il cancellazione di un file nella rete;

- **DBMS** (cfr. [26] e cfr. [13])(Database Management System): ha come obiettivo quello di realizzare un DBMS distribuito sfruttando la struttura della rete di PariPari;

- **NTP** (cfr. [27] e cfr. [14]) (Network Time Protocol): crea un sistema di sincronizzazione capace di gestire un orario unico ed affidabile per tutta la rete;

- **Web server** (cfr. [28] e cfr. [15]): permette la pubblicazione di pagine web su internet.

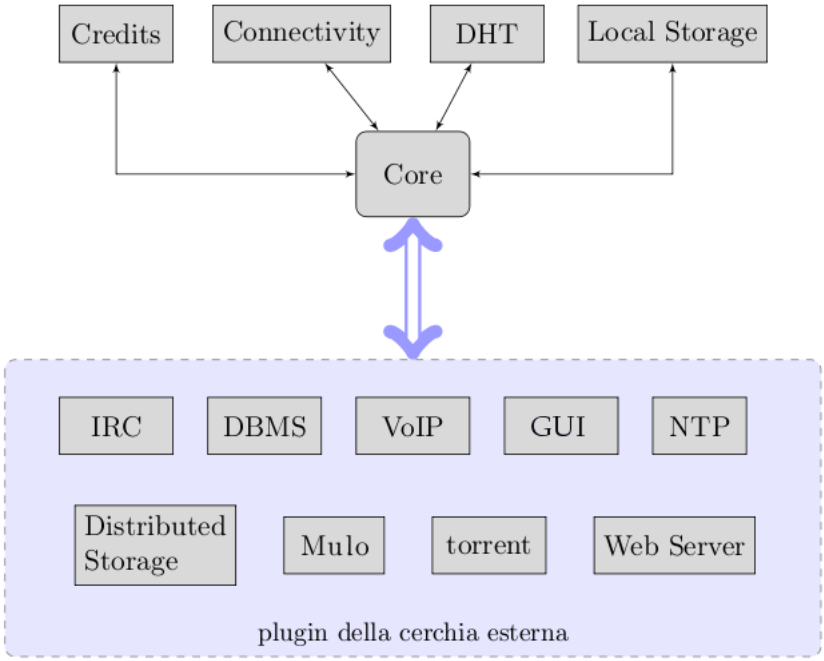


Figura 1.1: Struttura dei plugin di PariPari[37]

1.2 Gruppo

PariPari è sviluppato da circa sessanta studenti dell'Università di Padova iscritti alla laurea triennale o specialistica. A capo del progetto ci sono i professori Enoch Peserico e Paolo Bertasi. Gli studenti sono divisi in gruppi ed ognuno dei quali si occupa dello sviluppo di un plugin. All'interno di ciascun gruppo è possibile individuare un capo gruppo che si occupa della coordinazione dei lavori e della comunicazione con gli altri plugin o con i superiori. I membri di un gruppo si occupano sia di sviluppare il codice che di testarlo, purché ciascuno non testi il proprio codice, il che è molto sconsigliato perché rende difficoltoso il riconoscimento dei bug.

1.3 Implementazione

PariPari è scritto in Java per i seguenti motivi:

- la propensione di questo linguaggio ad una programmazione fortemente modulare;

- avere una maggior portabilità, potrà quindi funzionare in qualsiasi sistema operativo in cui è installato una Java Virtual Machine senza dover ricompilare il codice;
- nei primi corsi di informatica all'università di Padova viene insegnato questo linguaggio, ciò permetteva di avere a disposizione un largo numero di sviluppatori;
- permette l'integrazione dell'applicativo con il browser web grazie a Java web start (cfr. [29]).

In contrapposizione a questi vantaggi l'utilizzo di Java rispetto ad esempio a C++ comportava un leggero calo di prestazioni e non permetteva di poter scrivere codice a livello più basso, gestendo direttamente le locazioni di memoria.

Capitolo 2

Le reti P2P

In questo capitolo verranno esposti i concetti teorici necessari per la comprensione di quanto trattato nei capitoli successivi. Verranno inizialmente introdotte le reti P2P e la loro evoluzione fino alle DHT, per poi soffermarsi su due esempi di quest'ultimo tipo: Chord e Kademlia.

2.1 Introduzione alle reti P2P

Generalmente per *P2P*, cioè rete paritaria, si intende una rete di computer o qualsiasi rete informatica che non possiede nodi gerarchizzati come client o server fissi (clienti e serventi), ma un numero di nodi equivalenti (in inglese peer) che fungono sia da cliente che da. servente verso altri nodi della rete[18]. Gli esempi più famosi di queste reti sono nati per la condivisione di file, inizialmente file musicali (MP3).
Esistono tre possibili architetture:

- modello **centralizzato**;
- modello **distribuito non strutturato**;
- modello **distribuito strutturato**.

Il modello *centralizzato* utilizza dei server per mettere in contatto tra loro i vari nodi della rete, senza però prendere mai parte al trasferimento dei file. Un famoso esempio è Napster (cfr. [30]) che indicizzava i file musicali di tutti i sottoscrittori del servizio e permetteva all'utente di fare richieste (query), alle quali rispondeva con le identità dei peer che soddisfacevano i criteri di ricerca.

Questo era un sistema ibrido in cui la ricerca era basata su un sistema client-server mentre il trasferimento vero e proprio avveniva effettivamente in maniera P2P. Come si può intuire un problema fondamentale risultava essere la scalabilità: infatti il server era solo uno e più cresceva la rete, maggiore era il carico che esso doveva sopportare. Inoltre una rete di questo tipo forma una struttura ad albero in cui il server è la radice, in questo contesto l'eliminazione della radice equivale alla frantumazione dell'intero albero. Napster vista la sua relativa illegalità, non durò molto; infatti venne colpito da molte cause legali di etichette musicali, le quali sancirono la sua fine.

Nel modello *distribuito non strutturato*, a differenza che in quello centralizzato, l'accesso alla rete non avviene contattando un'entità superiore e anche la ricerca è distribuita, cioè ogni nodo risponde solo dei file che condivide. L'assenza del server implica la necessità di un'algoritmo di instradamento per la localizzare dei nodi e delle risorse. Data la natura non strutturata le ricerche avvengono in broadcast. Un esempio di questo tipo è il protocollo Gnutella(cfr. [31]). Uno dei suoi principali problemi era evitare l'inondazione della rete di messaggi dovuti alle ricerche fatte in broadcast. Per far ciò la circolazione infinita di un pacchetto era prevenuta con il campi TTL(time to live) in cui erano settati il numero massimo di nodi che poteva attraversare il messaggio, inoltre un nodo memorizzava temporaneamente l'ID dei messaggi che passavano attraverso lui e scartava eventuali messaggi già ricevuti. Non conoscendo l'intero sistema ma soltanto dei nodi vicini non era garantito il determinismo di una ricerca.

Si è dunque arrivati allo sviluppo del *modello distribuito strutturato*, in cui la topologia della rete è controllata e i messaggi del protocollo sono instradati a nodi sempre più vicini al nodo che conserva la risorsa ricercata. Questo viene ottenuto attraverso l'uso delle DHT che hanno i seguenti vantaggi:

- **decentralizzazione**: i nodi formano collettivamente il sistema senza alcun coordinamento centrale;
- **scalabilità**: il sistema è predisposto per un funzionamento efficiente anche con centinaia di milioni di nodi;
- **tolleranza ai guasti**: il sistema dovrebbe risultare affidabile anche in presenza di nodi che entrano, escono dalla rete o sono soggetti a malfunzionamenti con elevata frequenza.

Alcuni protocolli di questo tipo sono Chord (cfr. [39] e [19]) , Pastry (cfr. [40] e [32]) e Kademlia(cfr. [38] e [20]).

2.2 DHT

Le **DHT** sono una classe di sistemi distribuiti decentralizzati che partizionano l'appartenenza di un set di chiavi tra i nodi partecipanti. Per prima cosa ad ogni nodo che si connette alla rete viene assegnato un *identificatore* (ID), più precisamente è comune che un nodo che si sta connettendo si auto-assegni un ID scelto da uno spazio molto grande. Solitamente le implementazioni delle DHT utilizzano un ID di 160 bit creato tramite una funzione di hash non reversibile (ad esempio, SHA-1 (cfr. [33])). Bisogna poi definire una *metrica* nello spazio degli indirizzi, questa permetterà di calcolare la distanza fra due nodi qualsiasi della rete.
Le DHT servono a condividere risorse, a queste per essere identificate viene assegnato un *globally unique identifier* (GUID), il quale è solitamente ricavato a partire dalla risorsa stessa, o da alcuni suoi parametri, per mezzo di hash sicuro. L'utilizzo di un hash sicuro permette alla risorsa di autocertificarsi, più precisamente: i client che ricevono una risorsa possono verificare la sua autenticità, confrontando l'hash ricevuto con quello calcolato a partire dalla risorsa stessa. Per il corretto funzionamento di questa tecnica l'hash della risorsa non deve variare nel tempo, ciò comporta che la risorsa in questione deve essere immutabile. I nodi più vicini ad una risorsa sono i suoi *responsabili*, il che significa che sanno maggiori informazioni su di essa, come ad esempio da chi è stata condivisa. Per una maggiore disponibilità solitamente vengono create delle repliche degli oggetti, in modo tale che questi siano reperibili anche quando alcuni dei nodi, che ne hanno la competenza, non sono più connessi alla rete.
È necessario poi la realizzazione di un *algoritmo di instradamento*, il quale ha il compito di localizzare nodi e risorse nella rete. La procedura di instradamento è implementata utilizzando un algoritmo greedy (cfr. [34]) che iterativamente decresce la distanza fra il nodo di destinazione e l'i-esimo nodo intermedio al fine di raggiungere ciò che si sta cercando.
Le funzioni principali che deve compiere un algoritmo di instradamento di questo tipo sono le seguenti:

- un client che vuole ricercare una risorsa dovrà inviare una richiesta, comprensiva del GUID cercato, e utilizzando l'algoritmo di instradamento la farà pervenire ad uno dei nodi responsabili;
- un nodo che desidera condividere un nuovo servizio gli assegna per prima cosa un GUID e lo annuncia alla rete, la quale si assicurerà di renderlo disponibile agli altri partecipanti;
- quando un client richiede la rimozione di una risorsa, da lui messa a disposizione, l'algoritmo di instradamento deve essere in grado di soddisfare tale richiesta, non necessariamente ciò deve avvenire subito;
- i nodi possono entrare e uscire a piacimento dalla rete. Quando un nodo entra gli vengono inviate le informazioni di cui deve essere il responsabile. Quando un nodo esce (volontariamente o non) la sua responsabilità viene distribuita tra gli altri nodi.

2.3 Chord

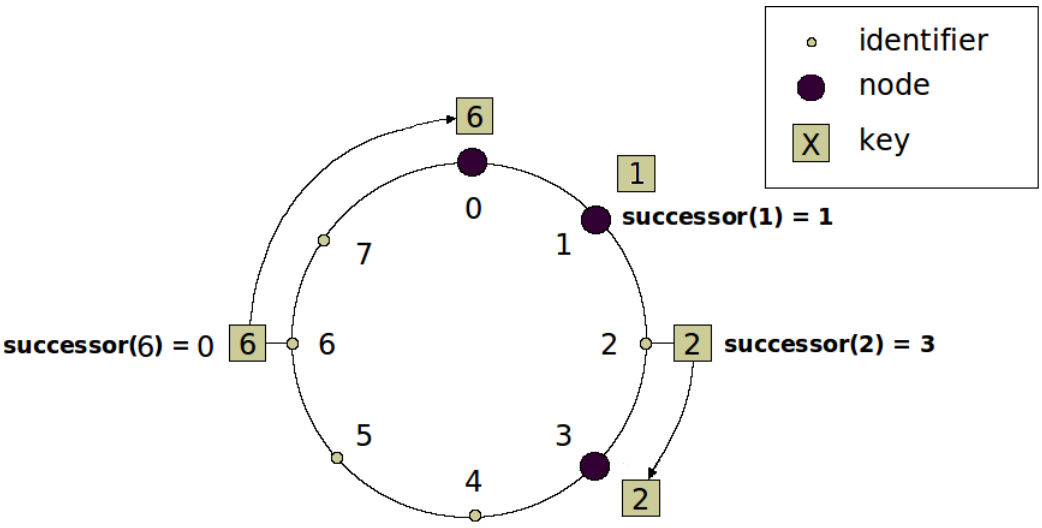


Figura 2.1: Dislocazione dei nodi in Chord con m=3

Chord è un sistema P2P puro e strutturato nato dallo studio di alcuni studiosi del MIT (Massachusetts Institute of Technology) a Cambridge nel Massachuset. Ai vari peer viene assegnato un ID lungo m bit e vengono disposti ordinati in

senso orario lungo una circonferenza modulo 2^m. Ogni nodo possiede quindi un successore e un predecessore. Il nodo successore di un nodo n è il peer che in senso orario viene subito dopo n.
Il protocollo impone che data una chiave essa venga mappata su un nodo, o meglio sul nodo con ID uguale o maggiore al valore della chiave. Per il calcolo della chiave, viene usata una funzione di hash consistente che permetta di mantenere il carico bilanciato e quindi di distribuire tra i vari nodi lo stesso numero di chiavi evitando, nel momento in cui un nudo entra o esce dal sistema, di muovere un gran numero di chiavi. Ciascun nodo mantiene una tabella di routing con m (ovvero

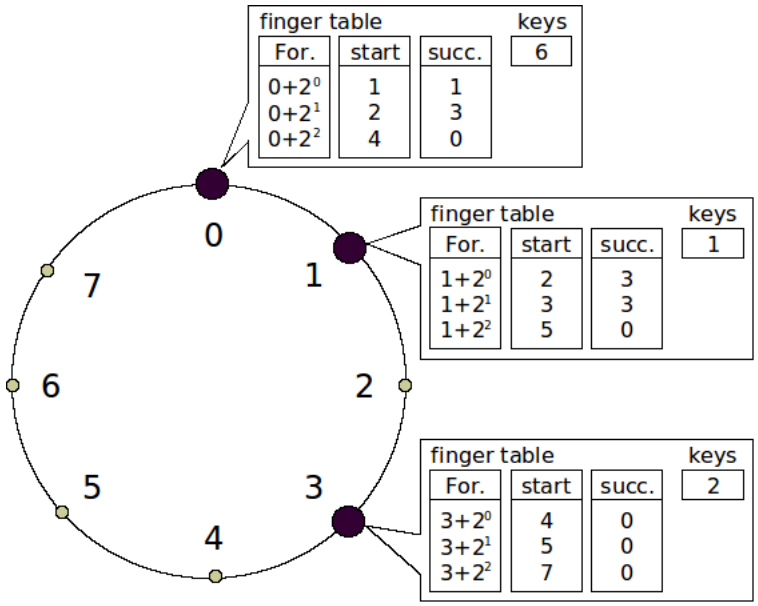


Figura 2.2: Finger table dei nodi appartenenti alla rete di tipo Chord

log N) elementi chiamata *finger table*. Dato un nodo n, l'i-esimo elemento della sua tabella contiene l'identità del primo nodo s che segue n di almeno 2^{i-1} ID sulla circonferenza.
s = successore(n + 2^{i-1}) dove $1 \leq$ i $\leq$ m (tutte le operazioni sono fatte modulo 2^m).
Il nodo s prende il nome di i-esimo finger di n. Ogni nodo in tabella è costituito da una terna del tipo <ID, IP, Port>, dove ID rappresenta l'identificativo e gli altri due parametri l'indirizzo IP e la porta a cui collegarsi per le comunicazioni. Si consideri ad esempio il caso in cui il nodo 3 voglia trovare la chiave numero 1(figura 2.3). Siccome 1 appartiene all'intervallo che parte dal 7, presente nella terza entry della finger table di 3 (figura 2.2), inoltra la richiesta al nodo contenuto in questa

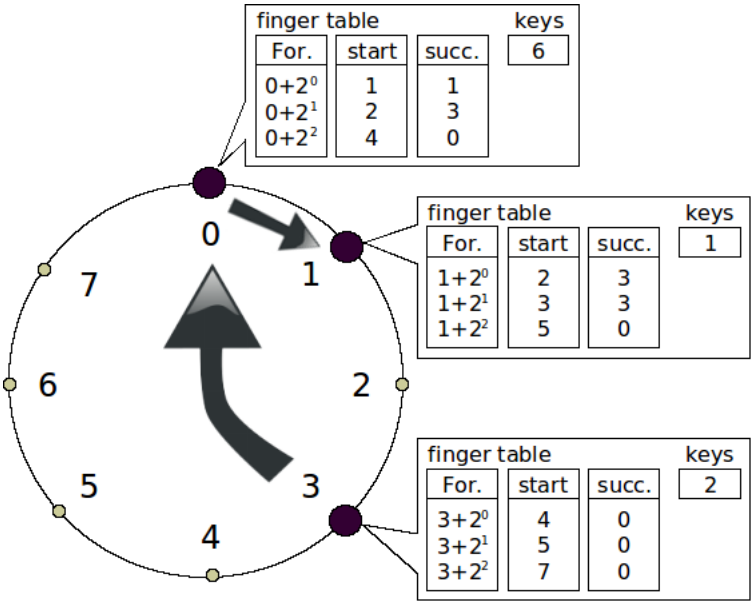


Figura 2.3: Ricerca della chiave 1 da parte del nodo con ID 3 in una rete di tipo Chord

entry: il nodo 0. Quest'ultimo, consultando la propria finger table, comprenderà che il successore della chiave 1 è il nodo 1 stesso e inoltrerà il messaggio. Questo algoritmo permette il reperimento di informazioni in al più O(log N) messaggi, a patto che la rete sia stabile. In caso contrario potrebbero essere necessari molti più messaggi, per questo sono state create delle procedure atte a mantenere la rete in uno stato consistente, preservando le seguenti due condizioni:

- il successore di ciascun nodo deve essere mantenuto correttamente;
- per ogni chiave k il nodo successore(k) è responsabile di k.

A tal fine Chord deve effettuare le seguenti operazioni ogni qual volta un nodo n entra nella rete:

- inizializzare il predecessore e la finger table di n;
- aggiornare le finger tables e i predecessori di altri nodi in modo tale da tener conto dell'ingresso di n;
- notificare il software del livello superiore, in modo tale che questo possa richiedere il trasferimento delle chiavi opportune.

Dunque in stato stazionario, cioè quando il sistema non è sottoposto a grandi cambiamenti, in una rete di N peers, ogni nodo mantiene informazioni solo su O(log N) altri nodi, e riesce a risolvere tutte le ricerche tramite O(log N) messaggi scambiati con gli altri nodi. Nel momento in cui un nodo entra o lascia la rete, con alta probabilità, saranno necessari non più di O(log^2 N) messaggi per l'aggiornamento della varie tabelle di routing[39].

2.4 Kademlia

Uno degli esempi più famosi di DHT è Kademlia, ideato da Petar Maymounkov e David Mazières, è essenzialmente un sistema per indicizzare host e risorse e permettere la ricerca di entrambi, in modo completamente decentralizzato. Questo sistema associa ad ogni risorsa e ad ogni nodo un hash univoco nello stesso spazio matematico. La metrica è basata sull'operazione di XOR.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 2.1: Tabella della verità della funzione XOR

Questo tipo di metrica definisce una topologia della rete a forma di albero binario per lo spazio degli indirizzi. I nodi sono sulle foglie e se assegniamo uno 0 agli archi sinistri e un 1 agli archi destri il percorso dalla radice fino ad una foglia definisce la rappresentazione binaria di quella foglia (che sarebbe un nodo della rete). Un nodo n ha ID = $b_{m-1}, b_{m-2}, b_{m-3}, ..., b_2, b_1, b_0$ e mantiene un set di m = O(log N) bucket nei quali memorizza i collegamenti ai nodi. Un generico bucket i contiene i nodi che:

- hanno lo stesso prefisso $b_{m-1}, b_{m-2}, b_{m-3}, ...$;
- hanno un valore differente nel bit b_i.

Tipicamente la dimensione dei bucket è limitata a 20 nodi per mantenere uno spazio occupabile dell'ordine di O(log N). Attraverso la metrica è possibile calcolare le distanze tra i nodi e le risorse ed assegnare ad ogni nodo attivo la responsabilità delle risorse a lui più vicine. Il nodo che vuole cercarne un altro nella rete, conoscendone l'ID, contatta i nodi che conosce più vicini a quel nodo richiedendo informazioni sull'oggetto della sua ricerca. I nodi interrogati rispondono fornendogli l'elenco di quelli più vicini di cui loro hanno notizia. Successivamente, il nodo cercatore, iterativamente, interrogherà dalla lista dei nodi ricevuti i più vicini a quello cercato finché questo ciclo non lo condurrà a contattare il nodo che voleva trovare. Volendo si può visualizzare questo schema di lavoro come una discesa lungo un albero binario in cui le foglie corrispondono ai nodi della rete. Ad ogni salto della ricerca si procede verso il basso escludendo mezzo sotto-albero fino ad arrivare alla foglia cercata impiegando quindi un tempo dell'ordine di O(log N).

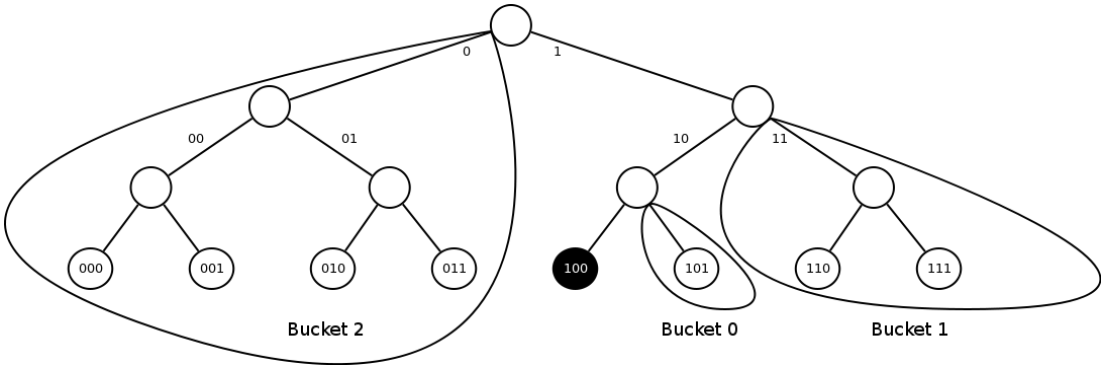


Figura 2.4: Un esempio di partizionamento della rete di kademlia in bucket per il nodo con ID = 100

Capitolo 3

La DHT in PariPari

Questo capitolo si prefigge di esporre le scelte implementative prese nella realizzazione della DHT di PariPari per poi soffermarsi sul protocollo di rete utilizzato e alla classi che si occupano della comunicazione fra i peer della rete.

3.1 Specifiche di progetto

Nella realizzazione del modulo si è cercato di rispettare le seguenti specifiche:

- **funzionalità:** obiettivo principale è l'ottenere un modulo DHT funzionante interfacciato con il resto di PariPari che gestisca i nodi all'interno della rete, dando la possibilità di mettere a disposizione delle risorse e di poterle reperire;

- **facilità di espansione:** il modulo deve essere progettato il più flessibile possibile per renderlo facilmente espandibile. Inoltre deve essere modulare poiché una buona suddivisione delle classi facilità eventuali modifiche. Il tutto cercando di utilizzare ciò che la programmazione ad oggetti mette a disposizione: sottoclassi, superclassi ed ereditarietà;

- **documentazione:** oltre della scrittura del codice è molto importante documentarlo. La documentazione è composta dalla Javadoc e dalla wiki del progetto[16] presente in Internet. Una buona documentazione è indispensabile in un progetto dove le risorse umane che vi lavorano vengono frequentemente rinnovate.

- **prestazioni:** inizialmente questo aspetto è di minor importanza rispetto all'ottenimento di un modulo funzionante, tuttavia non bisogna sottovalutare questa caratteristica dall'aspetto ambivalente. Infatti, si possono valutare le prestazioni a livello locale, come ad esempio le risorse in termini di disco, memoria e cicli macchina, oppure le prestazioni distribuite comprendenti i costi, in termini di numero di messaggi scambiati e dimensione di ciascun messaggio, per il mantenimento della rete e per effettuare le varie operazioni richieste. Per la natura di una DHT è più importante il secondo aspetto.

- **testing:** un altro aspetto da portare avanti assieme allo sviluppo del codice è il testing dello stesso, anzi per quanto affermato dalla versione modificata di *Extreme Programming*[35], tecnica adottata nello sviluppo di PariPari, il test dovrebbe precedere la stesura delle classi; più precisamente si dovrebbe scrivere il test di una classe subito dopo averne scritto l'interfaccia, prima ancora di iniziarne l'implementazione. Questo in pratica difficilmente avviene poiché spesso la classe da testare risulta essere piuttosto complessa ed è necessario avere certe informazioni per scriverlo adeguatamente.

- **range query:** attualmente non sono ancora sviluppate. Sono interrogazioni tipicamente usate all'interno delle basi di dati, permettono di richiedere al posto di un unico valore, come ad esempio un file, un intervallo di valori. Sarebbe possibile ottenere ad esempio tutti gli host aventi una banda massima di *2Mbit* oppure quelli nell'intervallo [*1Mbit, 2Mbit*].

3.2 Descrizione del protocollo

Il protocollo utilizzato dal plugin DHT è chiamato **DHTPP** (DHT PariPari Protocol), attualmente è alla versione 0.1 ed è una estensione del protocollo utilizzato da Kademlia. Con il termine nodo si tende ad identificare una qualsiasi macchina fisica connessa alla rete: sia essa un elaboratore di casa, un palmare o un potente server. Come già detto per i protocolli visti precedentemente, ad ogni nodo viene associato un GUID: un identificativo unico a livello globale. D'ora in avanti sarà semplicemente ID o identificativo. Un analogo identificativo, giacente cioè nello stesso spazio di quello dei nodi, è associato ad ogni risorsa che si vuole condividere all'interno della rete (in questo caso sarà spesso indicato anche con

il termine hash). Per l'assegnazione di questi identificativi si è deciso di usare l'algoritmo SHA-256 che fa uso di chiavi a 256 bit per rendere la probabilità di collisione altamente inferiore rispetto alla versione con chiavi a 160 bit. È molto importante che hash e ID giacciano sullo stesso spazio, al fine di poterli legare tramite una qualche metrica. Così facendo sarà possibile assegnare ad un nodo la responsabilità di tenere traccia delle risorse più "vicine". Come per kademlia la metrica è basata sull'operazione di XOR:

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

In questa versione sono implementate le seguenti primitive:

- **ping:** usata per controllare lo stato di un nodo, cioè se è ancora connesso alla rete;
- **find node:** come argomento in ingresso richiede un identificativo di 256 bit. Colui che riceve una richiesta di questo tipo restituisce k triple del tipo <ID,IP,Porta>, ciascuna tripla rappresenta un nodo della rete. Questi k nodi sono quelli più vicini all'ID che il nodo ricevente conosce;
- **pass resource:** come argomento in ingresso richiede un identificativo di 256 bit che rappresenta l'hash di una risorsa. È usata per passare le informazioni riguardanti una certa risorsa (come ad esempio la lista dei possessori) quando vengo a conoscenza di un nuovo nodo più vicino di me a questa risorsa;
- **find resource owner:** usata per reperire delle risorse sulla rete. Durante la ricerca, se un nodo si accorge di essere il responsabile della risorsa cercata, restituisce i nodi che la possiedono al posto di quelli più vicini;
- **store:** invia ad un nodo la richiesta di salvare una determinata coppia <chiave,valore>, in modo tale che questa possa essere reperita in futuro;
- **key note:** richiede le note associate ad una certa chiave.

Ogni messaggio deve contenere un intestazione come quella in figura 3.1, in cui i vari campi sono:

- **Version (1 byte):** rappresenta la versione del protocollo;

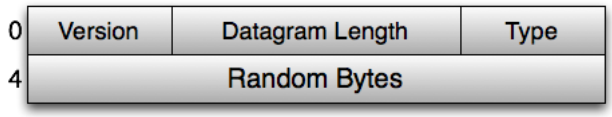


Figura 3.1: Intestazione dei messaggi DHTPP[17]

- **Datagram length (2 bytes):** rappresenta l'intera lunghezza del messaggio, attenzione però che questa lunghezza non corrisponde a quella datagramma UDP ma del solo payload;

- **Type (1 byte):** rappresenta il tipo di messaggio inviato, i 4 bit più significativi sono settati a 0 se il messaggio è una richiesta altrimenti a 1 se il messaggio è una risposta. I 4 bit meno significativi indicano la primitiva contenuta;

- **Random bytes (4 bytes):** questo campo è usato per identificare la sessione, quando viene creato un nuovo messaggio viene riempito con una sequenza creata in modo randomico, la risposta corrispondente dovrà avere la stessa sequenza;

Nella seguente tabella sono mostrati i valori del campo *tipo* attualmente in uso:

Primitiva	valore di richiesta	valore di risposta
Ping	0x01	0x11
Find node	0x02	0x12
Pass resource		0x16
Find resource owner	0x03	0x13
Store value	0x04	0x14
Get note for key	0x05	0x15

Tabella 3.1: Tabella dei possibili valori di tipo di pacchetto nel DHTPP

Dopo l'intestazione il messaggio dovrà contenere una rappresentazione del nodo inviante (vedi figura 3.2). Per identificare questo nodo è utilizzata una tripletta contenente indirizzo IP, numero di porta UDP e l'ID. È stato anche aggiunto un campo info di 12 byte utilizzato dagli altri plugin.

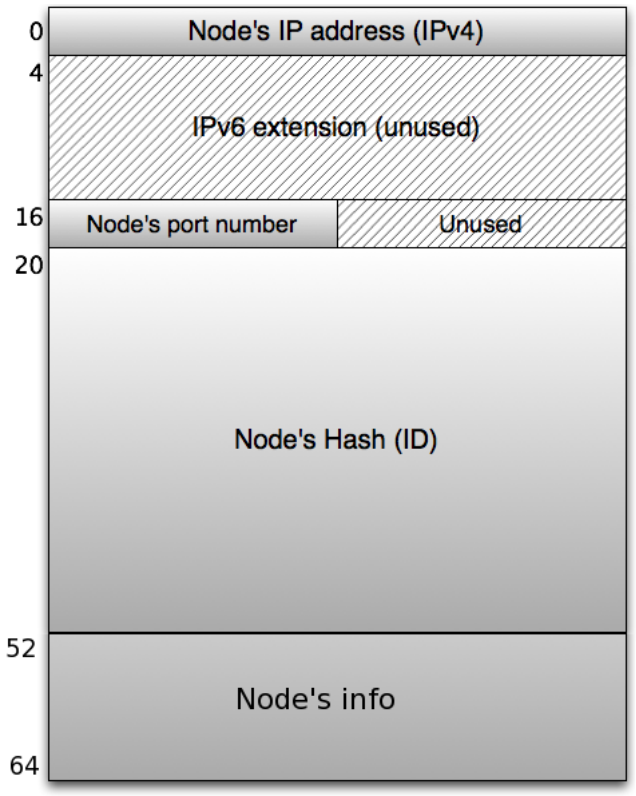


Figura 3.2: Rappresentazione di un nodo in un messaggio DHTPP[17]

3.3 Classi della rete

L'invio e la ricezione dei pacchetti dalla rete è gestita separatamente da due classi:

- **NetSender:** il quale preleva i pacchetti da una coda condivisa e li inoltra al destinatario;
- **NetReceiver:** gestisce i pacchetti ricevuti e li inoltra alla classe inRequest che li smista opportunamente;

Da notare che tutti i messaggi scambiati usano il protocollo UDP, poiché utilizzando il protocollo TCP il solo costo (in termini di byte scambiati tra due nodi) per stabilire una connessione è spesso superiore a quelli necessari per l'invio del messaggio voluto (ad esempio un ping).
Altre due classi importanti nella preparazione della comunicazione sono:

- **OutRequest:** una istanza di questo thread viene creata ed eseguita ogni qual volta un messaggio necessita di essere spedito nella rete. Dato che

il protocollo UDP non garantisce che il pacchetto giunga a destinazione, questa classe è dotata di un meccanismo che rispedisce il messaggio per un numero prestabilito di volte, qualora non avesse ottenuto risposta entro un certo tempo. Il tempo che intercorre tra due invii successivi cresce seguendo la nota tecnica (usata in TCP) di *backoff esponenziale*;

- **InRequest:** alla ricezione di un pacchetto dal NetReceiver ne verifica innanzitutto la tipologia. Se si tratta di una risposta, verifica se il pacchetto porta lo stesso numero di una delle richieste pendenti altrimenti viene scartata. Qualora la risposta sia valida viene inoltrata al corrispondente thread di OutRequest. Se invece si tratta di una request ne viene identificata la tipologia e inoltrata l'opportuna richiesta al DHTKernel.

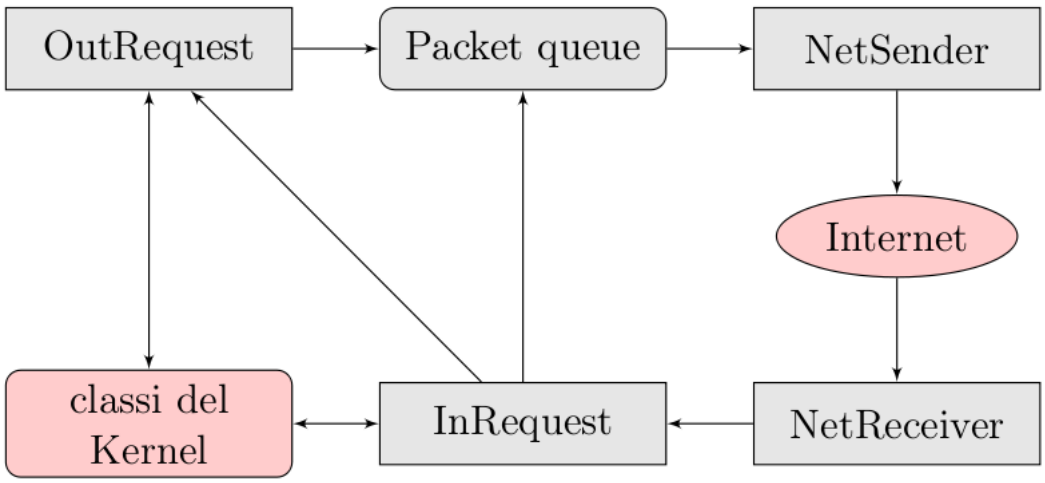


Figura 3.3: Struttura della rete di DHTPP[37]

Nella figura 3.3 è riportato uno schema che rappresenta le principali classi della rete e come esse interagiscono, tra loro e con l'esterno. Esistono però altre classi non indispensabili ai fini strutturali, ma necessarie per una buona realizzazione del modulo. Si tratta in particolare di:

- **DHTPacket:** rappresenta il pacchetto scambiato tra i nodi DHT;
- **NetInitializer:** inizializza la rete principalmente creando e gestendo NetSender e NetReceiver;
- **PacketFactory:** crea il DHTPacket adeguato da inviare sulla rete, a partire dai parametri dati in ingresso al costruttore ed al metodo invocato;

- **PacketParser:** si occupa di analizzare i pacchetti; ne riconosce la tipologia e altri campi;
- **PendingPacket:** rappresenta il tipo di oggetto contenuto nel pending buffer, che è devo sono tenuti i messaggi inviati che non hanno ancora ricevuto una risposta.

È stato inoltre creato un livello di classi intermedie, le quali si occupano della creazione del pacchetto adatto per implementare una certa primitiva.

Capitolo 4

La ricerca di un nodo e l'algoritmo del figlio prediletto

In questo capitolo si inizierà spiegando l'algoritmo di ricerca di un nodo utilizzato nel modulo DHT di PariPari, per poi introdurre l'algoritmo del figlio prediletto e infine illustrarne l'implementazione.

4.1 La ricerca di un nodo

La ricerca di un nodo è una funzione fondamentale delle DHT. Implementandola correttamente ha una complessità logaritmica, questo sta ad indicare il numero massimo di nodi (rispetto al numero totale di nodi che potrebbero essere presenti sulla rete) da visitare per giungere ad un risultato certo. La primitiva implementata è quella precedentemente denominata *FIND NODE*, a cui in ingresso viene passato un ID e il cui obiettivo è trovare i nodi più vicini al dato ID. In pariDHT la classe **LookUp** si occupa di questo.
In Kademlia il numero di nodi che si tende a recuperare è pari alla dimensione massima di un bucket, mentre PariPari permette di parametrizzare questo valore passandolo in ingresso al costruttore della classe LookUp; qualora non specificato viene usato il valore di default contenuto nel file di configurazione. D'ora in poi si indicherà con k il numero di contatti, vicini al nodo avente identificativo h, che si vogliono reperire.
L'algoritmo di ricerca è realizzato per mezzo di un thread implementato nella classe LookUp, i suoi passaggi principali sono:

1. ottiene i k nodi più vicini prelevandoli dal *NodeStorer*[1] locale. Se ve ne sono meno di k allora li preleva tutti;

2. con i risultati ottenuti precedentemente crea una *NodeList*;

3. prende i primi *ALPHA*[2] nodi non marcati della lista, li marca e invia ad ognuno una richiesta di tipo *FIND NODE* , ciascuno di essi restituirà k nodi prelevandoli dal proprio NodeStorer.

4. tra tutti i nodi ottenuti inserisce quelli opportuni nella NodeList. Se non vi sono più nodi marcati l'algoritmo termina mettendo a disposizione i nodi contenuti nella lista creata;

Ora si descriverà in modo dettagliato cosa fa, e come è stato implementato, ciascuno dei punti precedentemente elencati.
La **ricerca nel NodeStorer** avviene invocando il metodo:

Listing 4.1: Metodo utilizzato la ricerca nel Nodestorer locale

```
1 public synchronized SortedSet<INode> findNodes(BigInteger $h$, int howMany)
2     throws EmptyNodeStorerException
```

che opera in modo sincronizzato sulla struttura dati e lancia una *EmptyNodeStorerException* qualora essa fosse vuota. In ingresso basta specificare l'identificativo vicino al quale devono essere reperiti i nodi e quanti ne deve trovare. In genere gli vengono passati come parametri quelli dati in ingresso alla creazione della classe LookUp.
All'interno del metodo, per trovare i nodi desiderati viene dapprima individuato il bucket che li dovrebbe contenere, sia esso l'i–esimo, se questo bucket contiene abbastanza nodi allora vengono selezionati solo gli *howMany* più vicini; altrimenti vengono presi tutti i nodi del bucket in questione per passare poi a cercarne in altri bucket. Si può facilmente vedere che il secondo bucket da prendere in considerazione è l'(i −1)–esimo poi l'(i −2)–esimo e così via. Solo dopo aver raggiunto inutilmente il primo bucket si passa all'(i + 1)–esimo. Questa graduatoria dipende dalla metrica XOR la quale si basa sulla diversità di bit: i bucket successivi all'i–esimo differiscono da quello trovato originariamente per un bit più

[1]Struttura composta di bucket in cui vengono memorizzati i nodi, è simile a quella utilizzata in Kademlia

[2]Parametro contenuto nel file di configurazione che limita il numero di richieste contemporanee che è possibile inviare

significativo, pertanto presentano una maggiore distanza da questo. Se dopo aver scandito tutto il NodeStorer non si fossero trovati un numero sufficiente di nodi, allora vengono restituiti quelli presenti. Questo caso è possibile soltanto se si è entrati da poco nella rete e non si sono ancora conosciuti abbastanza nodi o se la rete è formata da pochi nodi. I risultati vengono restituiti in un SortedSet<INode> che è una struttura dati nei quali i nodi vengono inseriti ordinatamente e non permette l'inserimento di un nodo già contenuto (cfr. [1]). La **NodeList** è una struttura dati d'ausilio contenente un determinato numero di nodi, tale numero è passato in ingresso e solitamente è pari a k, il numero di nodi da reperire. Al suo interno i nodi sono disposti in ordine crescente di distanza da h, proprio per questo è necessario passare in ingresso al costruttore anche l'identificativo cercato. Ultima caratteristica peculiare è la capacità di marcare particolari nodi, in modo tale da distinguerli dagli altri. I metodi più importanti che possiede sono:

- **insert(INode):** scandisce la lista fino a determinare la posizione dove inserire il nodo specificato, e lo inserisce. Se la lista è piena l'inserimento del nodo comporta la cancellazione di quello in ultima posizione. Ovviamente il nodo non viene inserito qualora dovesse essere più lontano di quello che già occupa l'ultima posizione;

- **mark(INode):** marca il nodo specificato, se presente;

- **getFirstUnmarcked():** restituisce il primo nodo della lista a non essere stato segnato. Da notare il fatto che il nodo ritornato viene marcato. Se non ce ne sono, allora restituisce null.

La terza fase consiste nell'**invio delle richieste** verso la rete esterna. Vengono inviate ALPHA richieste ai primi nodi della Nodelist non marcati. ALPHA indica il numero massimo di richieste pendenti, serve sia a limitare il traffico sulla rete, sia a limitare la portata in uscita da un nodo in un limitato periodo di tempo. ALPHA è un parametro modificabile su file di configurazione e di default vale 3.

L'ultima fase è quella in cui vengono **valutate le risposte** e si verifica se si è giunti al termine. Nel messaggio di risposta si possono ricevere al più k nodi. Una volta ricevuti li si vanno ad inserire uno alla volta nella NodeList. In questo modo si sfruttano due proprietà della NodeList:

- non inserisce duplicati;

- inserisce un nodo solo se è più vicino dei k che già conosce;

Una volta inseriti si valuta se ci sono ancora nodi non marcati nella NodeList, in caso affermativo l'algoritmo riprende dal punto 3 altrimenti il thread di LookUp termina, quindi entra nello stato PERFORMED e imposta i risultati. A partire da questo momento, sarà possibile prelevarli per mezzo dell'apposito metodo:

Listing 4.2: Metodo utilizzato per prelevare i risultati della LookUp

```
1 public SortedSet<INode> getResults()
```

che li restituisce in un set ordinato.
Nel caso in cui nessuno dei contatti presenti nella NodeList risponda alla richiesta, vengono rimossi i nodi non più in linea dal NodeStorer e lo stato del LookUp viene impostato a PERFORMED_WITH_ERROR forzando così il termine del thread. Ulteriore particolare da curare è la verifica che:

- uno dei contatti ottenuti potrebbe essere proprio il nodo che desidera trovare h, in questo caso non ha senso inoltrare la richiesta a se stessi;

- uno dei contatti ottenuti potrebbe non avere lo stesso ID, ma lo stesso indirizzo ip e porta, del nodo che desidera trovare h. In questo caso è stata restituita una vecchia istanza non più valida, un contatto di questo tipo è certamente da scartare.

4.2 L'algoritmo del figlio prediletto

L'algoritmo del figlio prediletto, introdotto già nelle tesi di Paolo Bertasi[36] e Simone Giacon[37], permette di velocizzare la ricerca di un nodo nella DHT. Questo algoritmo non va a sostituire il metodo di ricerca visto precedentemente, ma si affianca ad esso. Questo perché il figlio prediletto potrebbe fallire e non terminare la ricerca.
Questa tecnica delega agli altri il compito di reperire le informazioni ed è molto simile a quella usata in *Chord*. I passi eseguiti dall'algoritmo sono i seguenti:

1. il nodo n vuole trovare i k nodi più vicini ad un certo ID h, come primo passo sceglie come prediletto il nodo p, ovvero il nodo che, localmente, tra quelli conosciuti è il più vicino ad h;

2. il nodo n invia una richiesta particolare al nodo prediletto p comunicandogli che è il suo prediletto e l'ID h che sta ricercando;

3. una volta ricevuta la richiesta, il nodo p verifica, localmente, quale fra i nodi conosciuti è il più vicino all'ID h; Chiamiamo quest'ultimo p';

4. p verifica se p' è più vicino di lui all'ID h;

   se p' è più vicino di p, p invia un messaggio a p' comunicandogli l'ID h da ricercare e a chi inviare gli eventuali risultati, ovvero il nodo n; a questa punto il nodo p' può essere rinominato p e ricominciare l'algoritmo a partire dal punto 3;

   altrimenti se p è più vicino di p' significa che è il nodo più vicino all'ID h; p risponderà ad n comunicandogli i k nodi che conosce più vicini a quell'ID (compreso se stesso) e quindi termina l'algoritmo;

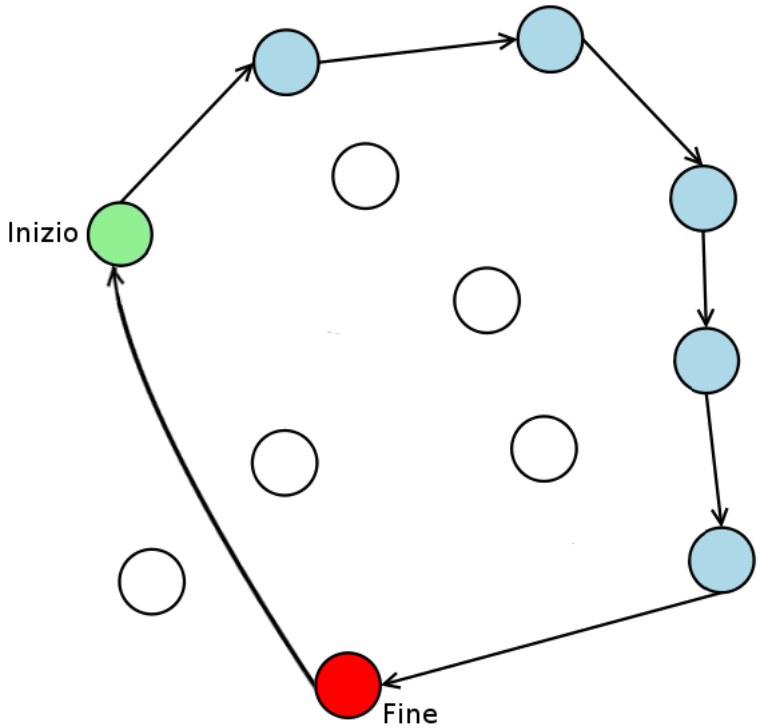
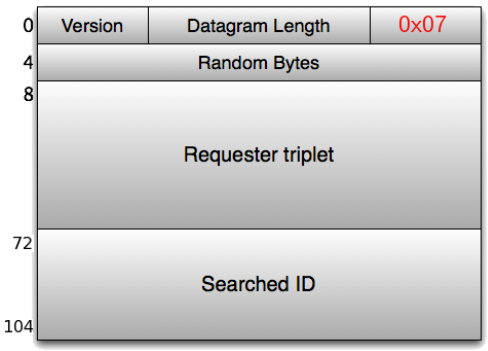


Figura 4.1: Il sistema del prediletto[36]

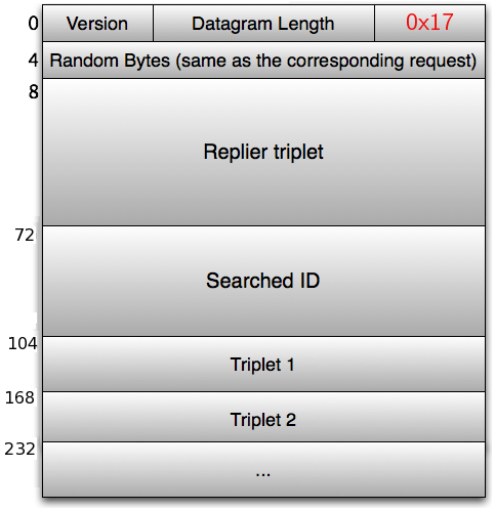
4.3 Realizzazione del sistema del prediletto

Per permettere la realizzazione delll'algoritmo del figlio prediletto è stato necessario aggiungere due tipi di pacchetto:

- **FAVOURITE_CHILD_REQUEST**(figura 4.2a): a questo pacchetto è stato assegnato il codice 0x07, l'intestazione è quella utilizzata per tutti i pacchetti e già analizzata quando si è visto il protocollo DHTPP; come in ogni messaggio il nodo deve aggiungere la tripletta <ID, IP, Port> che lo rappresenta seguita poi dall'ID che si sta cercando;
- **FAVOURITE_CHILD_REPLY**(figura 4.2b): a questo pacchetto è stato assegnato il codice 0x17 (in accordo con la regola prevista dal protocollo DHTPP che prevede che i messaggi di risposta abbiano un 1 nei 4 bit più significativo del campo tipo), anche in questo l'intestazione è quella utilizzata per tutti i pacchetti e il nodo deve aggiungere la tripletta <ID, IP, Port> che lo rappresenta; a ciò viene aggiunto l'ID che è stato cercato e le triplette <ID, IP, Port> che identificano i nodi più vicini a quell'ID;



(a) Favourite child request



(b) Favourite child reply

Una volta creati i pacchetti l'algoritmo di ricerca è realizzato per mezzo di un thread implementato nella classe **FavouriteChild**. Questa classe permette di parametrizzare il numero di nodi che si vuole recuperare passandolo in ingresso al costruttore; qualora non specificato viene usato il valore di default contenuto nel file di configurazione. Come già detto questo algoritmo va integrato con

il metodo di ricerca già implementato, poiché l'algoritmo del figlio prediletto potrebbe terminare. L'algoritmo termina senza giungere a nessun risultato nel caso in cui un nodo n scelga come prediletto un nodo p che n contiene ancora nel proprio NodeStorer, ma che in realtà si è già scollegato dalla rete. Per evitare ciò all'interno di PariPari esiste un thread periodico (la lunghezza del periodo è fissata attraverso un parametro del file di configurazione) che si occupa di andare a verificare che i nodi contenuti nel NodeStorer siano ancora in vita.
La classe LookUp che implementava la ricerca nel modo classico è stata rinominata **LookUpSearch**. Questo perché si è voluto assegnare il nome **LookUp** alla classe che implementa, attraverso un thread, l'integrazione fra la ricerca classica e l'algoritmo del figlio prediletto. La nuova classe LookUp avvia entrambi i thread di ricerca e utilizzando il manager delle sincronizzazioni implementato nel plugin DHT si mette in attesa di quella che termina per prima. Quando il primo dei due termina, verifica che la ricerca sia andata a buon fine, in tal caso setta i risultati, termina l'altro thread e finisce correttamente entrando nello stato PERFORMED. In qualsiasi caso la ricerca normale finisca, sia correttamente che non, il figlio prediletto viene terminato. Questo perchè la ricerca classica è più consistente e se utilizzando quella non si sono ottenuti risultati non è possibili ottenerli attraverso il figlio prediletto. Quindi se la ricerca classica non è andata a buon fine lo stato della LookUp viene settato a PERFORMED_WITH_ERROR.
L'algoritmo del figlio prediletto è implementato, attraverso un thread, all'interno della classe FavouriteChild. D'ora in poi si indicherà con k il numero di contatti, vicini al nodo avente identificativo h, che si vogliono reperire. La prima operazione effettuata è la ricerca in locale dei k nodi più vicini all'ID h. Questa, come per la ricerca classica, avviene con il metodo *findNodes* illustrato precedentemente (4.1); Una volta ottenuti, i risultati vengono inseriti in un TreeSet (cfr. [2]) che è una struttura dati che permette di mantenere i nodi ordinati a seconda della distanza rispetto all'ID che si sta ricercando. È simile alla NodeList precedentemente analizzata, a differenza di questa non permette di marcare i nodi, ma questa funzionalità non era necessaria nella realizzazione di questa classe. Nel TreeSet viene anche inserito il nodo che sta avviando la ricerca. Il primo nodo di questa struttura dati è il più vicino all'ID ricercato e sono quindi possibili due casi:

1. il primo nodo è proprio quello che sta facendo la ricerca; in tal caso il

thread termina correttamente, entra quindi nello stato PERFORMED e la struttura dati TreeSet viene restituita come risultato della ricerca;

2. il primo nodo è un altro nodo della rete; in questo caso viene inviata una FAVOURITE_CHILD_REQUEST;

Si vuole ora approfondire il secondo caso. Per venire inviato il messaggio è necessario creare un istanza della classe OutRequest, la quale viene creata ed eseguita ogni qual volta un messaggio necessita di essere spedito nella rete. Durante la creazione di una istanza della classe OutRequest è possibile settare, attraverso dei parametri del costruttore, il tempo massimo di attesa del messaggio di risposta ed il numero di invii da effettuare. In questo caso il tempo di attesa è settato ad un valore molto alto, ciò è dovuto a due ragioni:

- non è possibile stabilire la lunghezza di una catena, cioè quanti nodi attraversa, e nemmeno sapere quanto tempo impiega ad ogni salto, quindi sarebbe stato soltanto possibile fare una sovrastima;
- si sfrutta il fatto che il thread verrà comunque terminato dalla classe LookUp quando la ricerca classica fosse finita.

Dato che si è lasciato un tempo molto ampio si è deciso di effettuare un unico tentativo, cioè non sono previsti invii successivi al primo. Una volta che un nodo riceve una FAVOURITE_CHILD_REQUEST, egli capisce di essere un figlio prediletto e inizia a svolgere i propri compiti:

1. effettua la ricerca in locale dei nodi più vicini all'ID h; per far ciò utilizza il metodo emphfindNodes già illustrato nella porzione di codice 4.1; inserisce i risultati nella struttura dati TreeSet ed inserisce anche se stesso;

2. verifica chi è il primo nodo del TreeSet:

   se lui è il primo nodo invierà una FAVOURITE_CHILD_REPLY in risposta al nodo che aveva avviato la richiesta; nella risposta verranno inseriti le triplette <ID, IP, Port> rappresentanti i nodi presenti nella struttura TreeSet;

   altrimenti, in caso contrario, continua la catena di prediletti inviando una FAVOURITE_CHILD_REQUEST al primo nodo della struttura dati TreeSet; Per continuare la catena nel pacchetto di richiesta (in figura 4.2a)

nel campo in cui deve inserire le informazioni riguardanti il nodo che invia la richiesta inserisce la tripletta <ID, IP, Port> che identifica il nodo che ha iniziato la ricerca. Il nodo che riceverà il messaggio capirà di essere il prediletto e ricomincerà dal punto 1.

Una volta che il nodo che ha avviato la ricerca riceve la risposta i nodi vengono settati come risultati e il thread termina entrando nello stato PERFOMED. I risultati saranno a questo punto disponibili utilizzando il metodo *getResults* costruito in modo analogo a quanto visto precedentemente nella porzione di codice 4.2.

Capitolo 5

Conclusioni e sviluppi futuri

In questo capitolo si analizzerà l'"accelerazione" portata dall'algoritmo del figlio prediletto, le attuali pecche e i possibili sviluppi futuri da implementare.

5.1 Conclusioni

Riprendendo uno studio effettuato da Simone Giacon nella sua tesi[37], si supponga che il nodo n desideri ottenere i nodi più vicini ad un certo identificativo h e che invii una sola richiesta alla volta. Utilizzando l'algoritmo di ricerca classico è possibile quantificare il numero di trasmissioni da effettuare in funzione al numero di nodi presenti nella rete. Con uno spazio di 2256 possibili nodi, supponendo sia occupato da solo 232 nodi, è ragionevole ipotizzare, in presenza di una distribuzione uniforme all'interno dello spazio, un numero medio di nodi da visitare pari a 6. In figura 5.1 sono mostrati i messaggi inviati fra i nodi, si è indicato con una freccia nera doppiamente orientata tra il nodo n e ogni nodo m_i (con $1 \leq i \leq 5$), l'operazione di invio di una richiesta dal nodo n a cui il nodo m_i risponderà con una lista di nodi tra cui il nodo m_{i+1}, che il nodo n contatterà al passaggio successivo. Da ciò si conclude che saranno necessarie 12 comunicazioni pari cioè a due volte il numero di nodi da visitare.

Utilizzando invece l'algoritmo del figlio prediletto, che è simile a quello della ricerca di Chord in cui ci affida nelle mani di un altro nodo per far continuare la ricerca, si eviterebbero i messaggi di ritorno, come mostrato dagli archi rossi in figura 5.1, impiegando quindi 7 messaggi per concludere la ricerca. Supponendo che i messaggi impieghino circa tutti lo stesso tempo per giungere a destinazione,

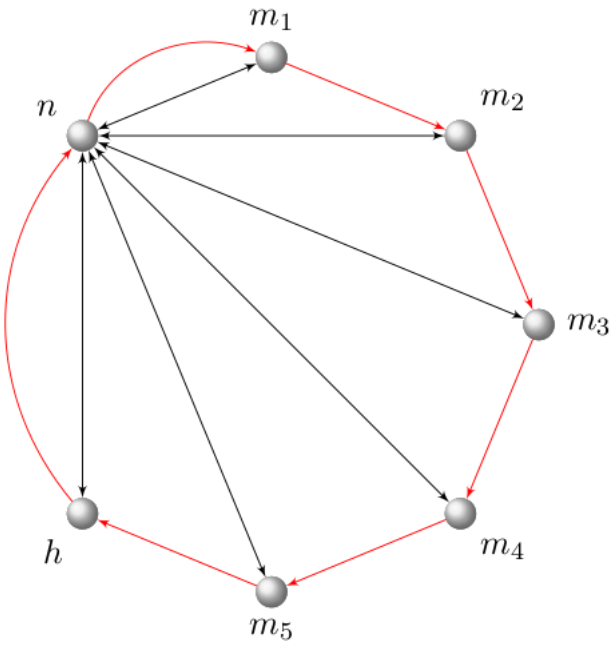


Figura 5.1: Il sistema del prediletto a confronto con la ricerca normale[37]

il sistema del prediletto permette di arrivare a destinazione in quasi la metà del tempo (per precisione un risparmio del 41,66%). Con il presente lavoro si è quindi implementato questo metodo nella sua forma più semplice, poiché ogni nodo sceglie un unico prediletto ma portando una buona "accelerazione" nelle ricerche. Nonostante ciò questo metodo soffre di alcuni punti deboli che verranno approfonditi nella prossima sezione, nella quale si spiegheranno i possibili rimedi che verranno implementati.

5.2 Sviluppi futuri

Per come è implementato attualmente l'algoritmo del figlio prediletto soffre di due punti deboli:

1. possibile interrompimento della catena di prediletti, con conseguente fallimento della ricerca;

2. sicurezza;

Per quanto riguarda il **primo punto debole** il problema si pone quando un nodo n sceglie come prediletto un nodo che n contiene ancora nel proprio NodeStorer,

ma che in realtà si è già scollegato dalla rete. La richiesta inviata da n non riceverà quindi nessuna risposta e la catena di prediletti si spezzerà.
Per risolvere questo problema è necessario che un nodo scelga più prediletti, ipotizziamo ne scelga un numero costante k. Non è possibile però che ad ogni passo dell'algoritmo ogni nodo scelga k prediletti altrimenti la rete verrebbe inondata da questi, dato che crescerebbero in maniera esponenziale. Una miglior soluzione è che il primo nodo n scelga k prediletti ed ogni nodo intermedio n_i faccia continuare la catena di figli prediletti scegliendone soltanto uno. In questo modo si diramerebbero k catene, a partire dal nodo n , che comunque rimarrebbero un numero costante senza inondare la rete e rendendo così più difficile la non riuscita dell'algoritmo, dato che sarebbe necessario l'interrompimento non più di un'unica catene ma di k catene. Un problema però è il mantenimento di queste catene disgiunte altrimenti se tutte andassero a finire sugli stessi nodi non porterebbe alcun vantaggio avere k catene.
La metrica XOR è *unidirezionale*, cioè per ogni punto x e per ogni distanza $\triangle > 0$ esiste uno e un solo punto y tale che d(x, y)[1] $= \triangle$ [38]. Questo assicura che tutte le ricerche per una chiave *convergono attraverso lo stesso percorso* con conseguente indesiderato congiungimento delle catene. Una possibile soluzione al problema era quella di distinguere il metodo con cui scegliere il prediletto all'interno delle tre catene. Ad esempio, scegliendo k=3, si potrebbe far in modo che:

- la catena numero 1 scelga come prediletto il nodo che localmente conosce più vicino all'ID ricercato;
- la catena numero 2 scelga il secondo più vicino;
- la catena numero 3 scelga il terzo più vicino;

In questo modo anche se due catene andassero a finire sullo stesso nodo al passo successivo andrebbero a saltare su due nodi diversi. L'attuazione di più catene farà aumentare in maniera costante il numero di messaggi totali inviati dall'algoritmo (non quelli necessari per giungere alla destinazione della ricerca) rendendo però l'algoritmo del figlio prediletto più robusto. Per quanto riguarda l'implementazione è allo studio un nuovo protocollo di rete che permetterà l'aggiunta di un campo tipo in cui, utilizzando dei codici numerici, sarà possibile comunicare ad un nodo come scegliere il proprio prediletto.

[1] distanza da x a y

Il **secondo punto debole** è la sicurezza. Per come è attualmente implementato il nodo che avvia la ricerca si affida totalmente nelle mani degli altri nodi, ma non tutti i nodi sono "buoni' e potrebbe esserci qualche nodo intenzionato a non far concludere correttamente la nostra ricerca. Nella prima colonna della tabella 5.1 sono illustrate le possibili operazioni che un "cattivo" nodo intermedio potrebbe effettuare.
Per poter risolvere questo tipo di problemi, come è spiegato nel testo "Security considerations for peer-to-peer Distributed Hash Tables"[41], l'unica soluzione è permettere al nodo che avvia la ricerca di osservarne i progressi. Questo significa che comunque un nodo intermedio sceglierà il proprio prediletto a cui inoltrerà la ricerca, però dovrà anche comunicare al nodo che ha avviato la ricerca a chi la inoltrerà, come mostrato in figura 5.2. Utilizzando il sistema così migliora-

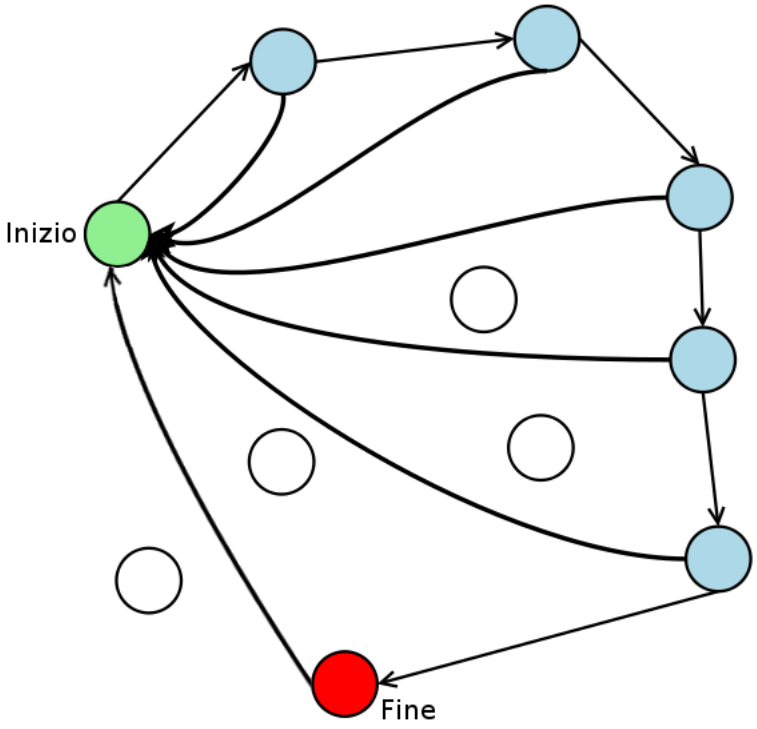


Figura 5.2: Il migliore sistema del prediletto[36]

to, il nodo n che avvia la ricerca potrà risolvere i problemi causati da un nodo malintenzionato nella maniera illustrata in tabella 5.1.

Tabella 5.1: Tabella dei possibili problemi arrecati da un nodo cattivo con relative possibili soluzioni

Azioni intraprese da un nodo "cattivo"	**Possibili soluzioni attuabili da** n
non inoltra il messaggio o lo inoltra a nodi non esistenti;	in questo caso la catena viene spezzata, il problema è parzialmente risolto utilizzando il più possibile catene disgiunte come visto precedentemente;
inoltra il messaggio ad un nodo che in realtà è più distante di lui dall'ID ricercato;	n potrà calcolare le distanze di entrambi i nodi notando ciò, potrà quindi decidere se spezzare la catena o ripartire dal nodo precedente a quello cattivo;
inoltra il messaggio ad un nodo che in realtà è più distante di lui dall'ID ricercato, ma comunica ad n di aver inoltrato il messaggio ad un altro nodo più vicino	n inizialmente giudicherà corretta la scelta del prediletto, ma una volta ricevuta la risposta parziale del nodo successivo a quello malintenzionato noterà che non è lo stesso che gli era stato comunicato; a quel punto n potrà quindi decidere se spezzare la catene o continuare la ricerca o riprenderla dal nodo precedente a quello cattivo;

Tabella 5.1: continua nella prossima pagina

Tabella 5.1: continua dalla pagina precedente

Azioni intraprese da un nodo "cattivo"	**Possibili soluzioni attuabili da** n
restituisce come risultato una lista di nodi non connessi;	n dovrà verificare la bontà dei risultati ricevuti, valutando se i nodi sono connessi; nel caso non lo fossero il messaggio verrà scartato e n potrà quindi decidere se spezzare la catena o ripartire dal nodo precedentemente a quello malintenzionato;
restituisce come risultato una lista di nodi lontani dall'ID ricercato;	n potrà confrontare i risultati ricevuti con quelli parziali da lui posseduti e verificare se i nodi ricevuti sono più distanti dall'ID ricercato rispetto ai nodi componenti i risultati parziali; nel caso fossero più lontani i risultati finali il messaggio verrà scartato e n potrà quindi decidere se spezzare la catena o ripartire dal nodo precedente a quello malintenzionato;

Tabella 5.1: si conclude dalla pagina precedente

Per poter attuare il nuovo sistema del prediletto è allo studio un nuovo protocollo di rete che permetterà una volta inviata una richiesta la ricezione di più messaggi di risposta , quali ad esempio i risultati parziali di una ricerca.

Bibliografia

[1] JavaDoc della struttura dati SortedSet. http://download.oracle.com/javase/1.4.2/docs/api/java/util/SortedSet.html.

[2] JavaDoc della struttura dati TreeSet. http://download.oracle.com/javase/1.4.2/docs/api/java/util/TreeSet.html.

[3] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Core_en.

[4] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/ConnectivityNIO_en.

[5] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Econ_en.

[6] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/DHT_en.

[7] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Local_Storage_en.

[8] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Torrent.

[9] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Mulo_en.

[10] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/VoIP_en.

[11] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/IRC.

[12] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Distributed_Storage_en.

[13] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Database_en.

[14] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/NTP.

[15] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/Web_en.

[16] Wiki di PariPari. http://www.paripari.it/mediawiki.

[17] Wiki di PariPari. http://www.paripari.it/mediawiki/index.php/DHTPP.

[18] Wikipedia. http://it.wikipedia.org/wiki/Peer-to-peer.

[19] Wikipedia. http://en.wikipedia.org/wiki/Chord_%28DHT%29.

[20] Wikipedia. http://it.wikipedia.org/wiki/Kademlia.

[21] Wikipedia. http://it.wikipedia.org/wiki/BitTorrent.

[22] Wikipedia. http://it.wikipedia.org/wiki/EDonkey.

[23] Wikipedia. http://it.wikipedia.org/wiki/Voip.

[24] Wikipedia. http://it.wikipedia.org/wiki/Internet_Relay_Chat.

[25] Wikipedia. http://it.wikipedia.org/wiki/Interfaccia_grafica.

[26] Wikipedia. http://it.wikipedia.org/wiki/Database_management_system.

[27] Wikipedia. http://it.wikipedia.org/wiki/Network_Time_Protocol.

[28] Wikipedia. http://it.wikipedia.org/wiki/Server_web.

[29] Wikipedia. http://en.wikipedia.org/wiki/Java_Web_Start.

[30] Wikipedia. http://it.wikipedia.org/wiki/Napster.

[31] Wikipedia. http://it.wikipedia.org/wiki/Gnutella.

[32] Wikipedia. `http://en.wikipedia.org/wiki/Pastry_%28DHT%29`.

[33] Wikipedia. `http://it.wikipedia.org/wiki/Secure_Hash_Algorithm`.

[34] Wikipedia. `http://it.wikipedia.org/wiki/Algoritmo_greedy`.

[35] Wikipedia. `http://it.wikipedia.org/wiki/Extreme_Programming`.

[36] Paolo Bertasi. Progettazione e realizzazione in java di una rete peer to peer anonima e multifunzionale. Master's thesis, Università degli Studi di Padova, 2005.

[37] Simone Giacon. PariPari: DHT 2008. Master's thesis, Università degli Studi di Padova, 2009.

[38] Petar Maymounkov and David Mazières. Kademlia: a peer-to-peer information system based on the XOR metric. 2002.

[39] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[40] Antony Rowstron and Peter Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[41] Emil Sit and Robert Morris. Security considerations for peer-to-peer Distributed Hash Tables. 2002.

Elenco delle figure

Elenco delle tabelle