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

TESI DI LAUREA

PARIKAD: KAD ROUTING TABLE

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

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: *Christian Piccolo*

Anno Accademico 2009-2010

Ai miei genitori,
per le matite e i colori.

Sommario

Kad è indubbiamente una delle poche DHT che si è estesa su una così ampia scala, anche grazie alla sua integrazione in programmi di file sharing di grande successo quali eMule/aMule e MLDonkey: ogni giorno Kad ha più di 1,5 milioni di utenti connessi simultaneamente.
L'obiettivo di questa tesi è quello di descrivere in dettaglio il funzionamento della struttura dati che sta alla base della rete Kad; si farà in particolare riferimento al progetto PariKad, il quale implementa in Java una rete Kad sviluppata per il progetto PariMulo di PariPari [1]. La struttura dati alla base di PariKad si basa fedelmente sull'implementazione di Kad in eMule 0.49c [2].

Indice

Capitolo 1

Introduzione: il progetto PariPari

PariPari è un progetto ambizioso, portato avanti dagli studenti dell'Università di Padova, il quale ha come obiettivo la creazione di un software libero scritto in Java, che raggruppi al suo interno applicazioni di file-sharing, messaggistica, video conferenza e permetta all'utente di usufruire di interessanti servizi tra cui quelli di WebServer, DBMS[1] e DHT[2].

L'idea innovativa di questo progetto è quella di far gestire da un'applicazione unica molti programmi che utilizzano Internet e che, fino ad ora, hanno sempre lavorato in modo indipendente l'uno dall'altro. Il fine è quello di utilizzare la rete nel modo più efficiente: l'utente finale potrà, ad esempio, parlare fluidamente in video conferenza, scaricare molti file dalla rete eDonkey e chattare con suoi amici, senza dovere per forza interrompere una di queste tre attività, come spesso invece accade con le applicazioni attuali.

PariPari è suddiviso in diversi plugin ognuno dei quali ricopre un compito preciso e permette il funzionamento di altri plugin o estende nuove funzionalità al programma. Il plugin **PariMulo**, in particolare, si occupa di riprodurre un client eMule permettendo di eseguire ricerche nella rete eDonkey per scaricare e condividere file.

In questo contesto si inserisce il nostro lavoro; PariMulo, infatti, non supportava fino ad ora l'utilizzo della rete Kad per la ricerca e la pubblicazione di file. La differenza sostanziale tra la rete eDonkey e la rete Kad sta nel fatto che, mentre la prima è una rete centralizzata, la seconda è una rete decentralizzata. Kad

[1] Database Management System
[2] Distribuited Hash Table

sfrutta la potenza di tutti gli utenti connessi a questa rete per condividere file e funzionare ininterrottamente, senza dover dipendere da un unico server centrale.

L'integrazione della rete Kad nel plugin di PariMulo ha portato alla nascita di **PariKad**. PariKad si può sostanzialmente dividere in due grossi blocchi: una prima parte legata allo scambio di messaggi tra utenti, necessario per il funzionamento globale di Kad, e una seconda parte che si occupa di gestire la struttura dati, tenendo organizzati i contatti della rete. Questa tesi si prefigge come obiettivo quello di descrivere in modo esauriente quest'ultimo punto.

Capitolo 2

Architettura Kad

L'obiettivo di questo capitolo è quello di spiegare i concetti che stanno alla base di ogni DHT (*Distributed Hash Table*) tra cui Kad, una DHT usata per reti peer-to-peer e ideata da Petar Maymounkov e David Mazières della New York University [4]. É necessario introdurre alcuni termini fondamentali:

- **peer:** un punto di connessione nella rete Kad.
- **client:** un'applicazione (per esempio come eMule/aMule) in grado di connettersi alla rete Kad; il computer su cui viene eseguita quest'applicazione risulterà essere un peer.
- **hash ID:** si intende un valore a 128 bit (16 byte) che identifica in maniera quasi univoca qualsiasi oggetto nella rete Kad, sia esso un oggetto passivo (keyword, file, etc.) o un peer. Per i peer, l'hash ID è creato in maniera pseudorandom all'avvio del client, mentre per gli oggetti passivi si ottiene attraverso il calcolo del MD4 su particolari caratteristiche dell'oggetto.
- **nodo:** si intende una particolare zona nella rete Kad in cui tutti gli oggetti/peer presenti hanno lo stesso hash ID; in ogni nodo ci possono essere più peer e oggetti passivi che possiedono lo stesso hash ID.
- **contact:** un peer conosciuto che viene tenuto in memoria e gestito per usi futuri.

Quindi un utente, che avvia il proprio client eMule, fa sì che il suo computer diventi un peer della rete Kad. Appena questo peer si mette in contatto con

altri peer della rete e viene accettato, i suoi dati vengono immagazzinati ed esso diventa un contact per l'utente che ne ha salvato i dati.

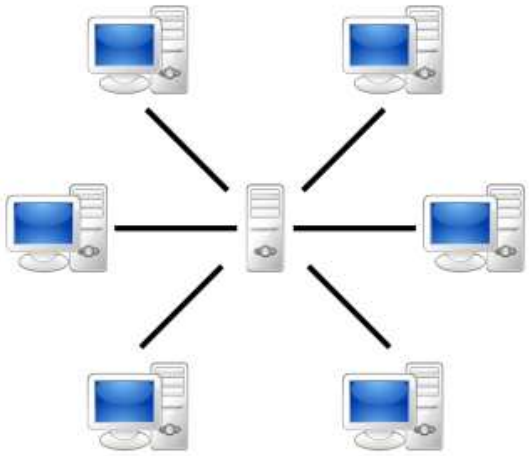

Figura 2.1: rete centralizzata

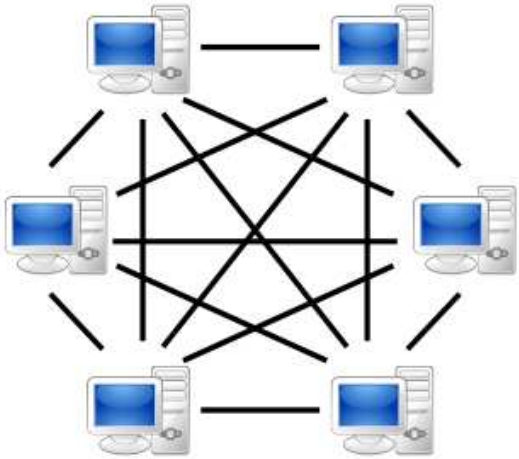

Figura 2.2: rete decentralizzata

I punti di forza di una DHT sono la **decentralizzazione**, cioè la possibilità di permettere lo scambio informazioni tra vari utenti senza l'uso di un server centrale di coordinamento (figura 2.1 e 2.2), la **scalabilità**, la quale permette alla rete di funzionare efficientemente con centinaia o milioni di nodi, e la **stabilità**, poiché la continua connessione di nuovi peer, la loro disconnessione dalla rete e i loro problemi di comunicazione non inficiano il funzionamento globale della DHT. Ne consegue che una buona DHT deve fornire un ottimo metodo per fare comunicare i vari client tra di loro e un'efficiente struttura dati che permetta di memorizzare i contatti ed eseguire velocemente azioni come la ricerca di file, keyword, etc.

Questa tesi non ha lo scopo di spiegare il meccanismo di comunicazione usato tra i vari peer nella rete Kad, tuttavia si è scelto di riportare, per completezza, una breve e semplice spiegazione a riguardo.

2.1 Network Protocol

Kad usa il protocollo UDP per gestire le comunicazioni tra i vari peer. Il protocollo UDP permette di scambiare messaggi in maniera molto veloce, con un minimo overhead e senza la necessità di eseguire un handshake prima dell'inizio della comunicazione. Se un client A vuole inviare un messaggio ad un suo con-

tatto B, è sufficiente che A invii il pacchetto all'indirizzo IP di B alla porta in cui B è in ascolto per i pacchetti UDP. Se B è attivo, allora legge il messaggio e se opportuno risponde, altrimenti A, non ottenendo repliche, può decidere di mandare il suo messaggio ad altri client, che forse sono in grado di soddisfare la sua richiesta.

Risulta evidente che uno dei primi problemi che Kad deve gestire è come permettere ai vari client di rendersi conto che un loro contatto si è disconnesso dalla rete. La soluzione semplice ed efficace è quella di usare un timer, il quale va a verificare dopo ogni intervallo di tempo prestabilito se il contatto è ancora attivo attraverso uno scambio di pacchetti UDP di tipo PING/PONG (se arriva un pacchetto PING bisogna rispondere con un pacchetto PONG). Se dopo alcune volte che si è cercato di comunicare con il contatto esso continua a non rispondere si procede alla sua eliminazione dalla nostra Routing Table, la struttura dati che si occupa di gestire i nostri contatti.

Ogni pacchetto UDP inviato nella rete Kad è contraddistinto dal primo byte chiamato ID. eMule/aMule così come PariKad utilizzano il byte 0xE4 per pacchetti standard e 0xE5 per pacchetti compressi. Il byte succcessivo definisce invece la funzione che ha quel pacchetto, cioè se è un pacchetto per richiedere file, per comunicare qualcosa, o altro. I byte che seguono dipendono ovviamente da pacchetto a pacchetto.

Per completezza è importante ricordare che negli ultimi anni eMule ha sviluppato una tecnica chiamata *Protocol Obfuscation* che ha l'obiettivo di criptare i pacchetti in uscita (e decriptare quelli in entrata) cosicché gli ISP[1] non siano in grado di bloccare i pacchetti della rete Kad[2]. Di conseguenza, il pacchetto criptato non inizia più con i byte 0xE4 o 0xE5, ma sembra una sequenza casuale di byte. Tuttavia questo pacchetto, una volta decriptato, torna ad avere la struttura classica sopra descritta.

Un client della rete Kad è quindi caratterizzato da due porte utilizzate per funzioni diverse. Una porta UDP che è utilizzata per scambiare messaggi relativi a operazioni standard (ricerca, pubblicazione file, ...) e di mantenimento (ping/pong, pacchetti di hello, ...) e una porta per pacchetti TCP utilizzata per

[1]Internet Service Provider.

[2]Il traffico generato da programmi peer-to-peer viene infatti spesso bloccato per evitare rallentamenti sulla rete Internet globale.

scaricare e condividere i propri files con sicurezza. eMule usa come standard la porta UDP 4672 e TCP 4662, ma questo valore può essere cambiato dall'utente.

2.2 Spazio a 128-bit

Ogni hash ID in Kad è di 128 bit, per questo si afferma che il protocollo Kad è caratterizzato da uno spazio a 128 bit, infatti ciò comporta che è possibile collocare in maniera univoca fino a $2^{128} \simeq 3.40 * 10^{38}$ oggetti differenti (è un numero molto grande se si tiene presente che l'universo osservabile contiene circa $7 * 10^{22}$ stelle). Ogni utente che vuole partecipare alla rete deve crearsi un valore random a 128 bit che in seguito rappresenterà il suo client ID; anche ogni file o keyword presente in Kad deve ottenere un ID a 128 bit, in modo che possa collocarsi in un punto preciso della rete. In aggiunta a ciò, tutti gli oggetti passivi devono anche trovare un *host peer* nelle vicinanze che si occupi di gestirli.

É importante comprendere fin da subito che client vicini geograficamente possono non esserlo nella rete Kad e viceversa. Dal momento che il client ID si può interpretare come l'indirizzo che un client sceglie di avere nella propria rete Kad, può accadere che due client diversi generino due valori random a 128 bit molto vicini tra loro e che, quindi, la loro distanza possa risultare molto piccola. In metafora possiamo dire che questi due client si ritroveranno ad essere vicini di casa nel mondo virtuale di Kad.

A questo punto è necessario spiegare in che modo calcoliamo la distanza tra peer e/o oggetti, o meglio, come calcoliamo la distanza tra due ID a 128 bit, siano essi associati a un peer o ad un oggetto passivo. Per far ciò Kad si avvale della metrica XOR (simbolo $\oplus$). Si riporta la tabella di verità della metrica XOR e un esempio per meglio comprenderne il funzionamento.

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 2.1: tabella di verità XOR

110100110	HASH ID A
010101001	HASH ID B
100001111	A $\oplus$ B

Tabella 2.2: distanza calcolata tra due hash ID secondo la metrica XOR

Come verranno gestiti i contatti con i loro ID è oggetto di discussione nel prossimo capitolo, il quale spiegherà in dettaglio le caratteristiche della Routing Table di Kad.

Capitolo 3

Routing Table

La Routing Table rappresenta il cuore della rete Kad poiché essa tiene in memoria tutte le informazioni riguardanti i contatti e permette di eseguire rapide ricerche su di essi, trovando quelli che più adatti su cui eseguire determinati compiti, quali la ricerca di un file, di una keyword o di altri contatti.

Per descrivere il funzionamento della Routing Table si utilizzerà un approccio buttom-up; verranno cioè introdotte per prime le classi più semplici che stanno alla base e si andrà via via costruendo la struttura dati con classi sempre più complesse. Le spiegazioni faranno spesso riferimento alle classi, ai metodi e alle variabili Java usate in PariKad, ma ciò non limiterà in alcun modo la comprensione del funzionamento della Routing Table anche a chi non padroneggia Java.

Le informazioni che verranno di seguito spiegate sono state ricavate in gran parte dalla lettura del codice sorgente di eMule 0.49c, che rappresenta l'unica fonte seria e affidabile per poter comprendere appieno il funzionamento della rete Kad ed in particolare della sua struttura dati. I numerosi documenti che si trovano in Internet sono invece risultati spesso approssimativi o imprecisi.

3.1 Int128

La classe `Int128.java`, la più semplice e fondamentale di tutta la struttura dati, è usata per rappresentare gli hash ID[1] dei peer, dei file e delle keyword. Questa

[1]Nel seguito della spiegazione si utilizzeranno in maniera intercambiabile le parole hash ID o Int128.

classe che rappresenta un valore a 128bit (16 byte) viene usata per rappresentare molte variabili ed è quindi necessario che sia efficiente nel consumo di memoria.

Per implementare questa classe si è sfruttata la classe BitSet di Java, che fornisce la possibilità di creare un array di bit booleani (true o false) di dimensioni a piacere ed è provvista di un metodo per fare lo XOR tra due BitSet.

tipo	**nome variabile**	**descrizione**
BitSet	`bitSet`	classe Java usata per rappresentare un Int128

Tabella 3.1: variabili di Int128

La classe `Int128.java` è stata quindi provvista di metodi che mappassero i valori booleani true e false nei corrispettivi 1 e 0[2] (vedi tabella 3.2), oltre ad alcuni metodi che permettessero la sua rappresentazione in stringa binaria ed esadecimale, lo shift di un Int128 e altre funzioni che tornavano utili.

indice	$\mathbf{0}_{MSB}$	**1**	**2**	**3**	**4**	$\cdots$	**124**	**125**	**126**	$\mathbf{127}_{LSB}$
Int128	1	0	0	1	1	$\cdots$	1	0	1	0
BitSet	true	false	false	true	true	$\cdots$	true	false	true	false

Tabella 3.2: relazione Int128 e BitSet

3.2 KadContact

Ogni peer ha una lista di peer conosciuti chiamati contatti, i quali vengono inseriti nella Routing Table.

La classe `KadContact.java` si occupa di gestire tutte le informazioni di un contatto. Essa è composta principalmente da metodi setter/getter che permettono di settare o ottenere il valore delle sue variabili. Si riporta in tabella 3.3 tutte le variabili di questa classe con una loro breve descrizione.

Il nostro Int128, usato per ottenere la variabile `contactDistance`, viene creato random durante il primo avvio di PariKad e salvato in un file, all'uscita dell'applicazione. Al successivo avvio, si riutilizza il valore dell'Int128 recuperato dal

[2]A differenza di altri linguaggi di programmazione quali C/C++, Java non permette il cast implicito di un byte/int in un booleano né il viceversa

tipo	**nome variabile**	**descrizione**
Int128	`id`	hash ID che identifica il contatto
Int128	`distance`	distanza tra il nostro Int128 e quello del contatto (distance = nostro ID XOR suo ID)
InetAddress	`ip`	indirizzo IP
byte	`type`	identifica la bontà del contatto: range [0,4]
byte	`kadVersion`	versione del protocollo Kad
int	`TCPPort`	porta TCP
int	`UDPPort`	porta UDP
long	`creationTime_millis`	quando è stato creato il contatto
long	`expirationTime_millis`	indica fino a quando si può considerare valido il valore `type` impostato
long	`lastTypeSetTime_millis`	quando è stato aggiornato per l'ultima volta il type del contatto
KadUDPKey	`kadUDPKey`	chiave utilizzata per inviare pacchetti offuscati al contatto
boolean	`IPVerified`	indica se l'IP del contatto è stato controllato e risulta valido
boolean	`receivedHelloPacket`	true se è giunto un pacchetto di HELLO da questo contatto, false altrimenti
boolean	`checkKad2`	segnala se è già stato fatto un controllo per verificare se il contatto supporta i pacchetti di tipo Kad2

Tabella 3.3: variabili di KadContact

file; questo comportamento coerente con quello di eMule dovrebbe portare ad una maggior stabilità alla rete Kad.

Delle variabili sopra riportate desidero spiegare in dettaglio la funzione di `kadVersion` e `type`.

Il byte `kadVersion` identifica quale versione del protocollo Kad utilizza il contatto. Il valore 0 significa che il KadContact utilizza la versione 1 di Kad, mentre ogni altro valore maggiore di 0 implica che il contatto utilizza la versione 2 di Kad. Questa differenza è molto importante poiché la versione di Kad utilizzata determina quale tipo di pacchetti possono essere spediti, quali caratteristiche il contatto supporta, etc.

Il byte `type`, invece, permette di valutare l'affidabilità di un contatto. Appena un contatto è aggiunto gli viene assegnato il valore `type=3`. Periodicamente il

client controlla se questo contatto è ancora attivo e lo promuove o degrada secondo la regola in tabella 3.4.

`type`	**descrizione**
4	il contatto deve essere cancellato (in genere poiché non è raggiungibile)
3	questo è il valore assegnato di default ai contatti che sono appena stati aggiunti alla Routing Table
2	contatti conosciuti da meno di 1 ora, ma che hanno dimostrato di essere attivi
1	contatti conosciuti da meno di 2 ore ma da più di 1 ora e che hanno dimostrato di essere attivi
0	contatti conosciuti da più di 2 ore e che hanno dimostrato di essere attivi

Tabella 3.4: descrizione variabile `type` di KadContact

Il `type` di un contatto viene memorizzato in un file alla chiusura dell'applicazione, affinché al successivo avvio il nostro client sappia fin da subito quali sono i contatti migliori che stanno connessi alla rete Kad da più tempo.

3.2.1 GeoLocator

Il GeoLocator è una funzionalità che è stata aggiunto alla classe `KadContact.java` per permetterle di riconoscere a che nazione appartengono i suoi contatti. Ci si è appoggiati alla libreria java InetAddressLocator [8].

tipo	**nome variabile**	**descrizione**
HashMap<String, Integer>	`contactsGeoLocation`	associa ad ogni Nazione X il numero di contatti che è in X

Tabella 3.5: variabile statica di KadContact

In teoria, eseguendo il GeoLocator su molti contatti, si dovrebbe trovare una distribuzione molto simile a quella presente nell'articolo [6]. Si riporta in figura 3.1 l'istogramma presente in quella pubblicazione, che è stato ottenuto attraverso un'analisi di quasi tutti i contatti presenti nella rete Kad il 30/08/2006.

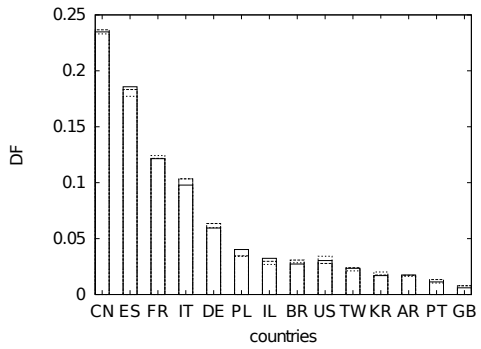

Figura 3.1: distribuzione geografica dei peers dell'intero spazio Kad al 30/08/2006

3.3 RoutingBin

I KadContact vengono a loro volta inseriti in delle liste di massimo 10 KadContact ciascuna (per questo vengono definite spesso 10-Bucket), gestite dalla classe `RoutingBin.java`, la quale si occupa di tenerli ordinati secondo un criterio di tempo.

tipo	**nome variabile**	**descrizione**
List<KadContact>	`m_listEntries`	contiene al massimo 10 KadContact

Tabella 3.6: variabile di RoutingBin

Appena un contatto si dimostra attivo (ha risposto a una delle nostre richieste o ci ha chiesto qualcosa), viene spinto in fondo alla lista. In questo modo i contatti più vecchi, in relazione alla nostra ultima comunicazione con loro, vengono a trovarsi in cima alla lista, mentre quelli più nuovi in fondo (si veda a proposito la tabella 3.7). Questo modo di gestire i contatti permette di ottenere una rete Kad più stabile, poiché i KadContact in cima alla lista vengono periodicamente contattati per vedere se sono ancora attivi e dopo un determinato numero di fallimenti vengono rimossi dalla nostra Routing Table (questo comportamento verrà descritto bene nella sezione relativa ai task di mantenimento 3.5).

RoutingBin List (10-Bucket)	
indice	**contactDistance dei KadContact aggiunti**
$0_{oldest\ contact}$	110101010101010...000001100010010
1	010000001100010...010101101010010
2	000011100010000...111111101010010
3	111111010101010...110100000000010
4	001010100100010...110101101010010
5	010101010101010...110011111101010
6	110101011111110...110101101010010
7	111111111111111...111111101010010
8	000000000000000...000000000000001
$9_{newest\ contact}$	111111111010110...110101101010010

Tabella 3.7: esempio di RoutingBin completamente riempita

3.4 RoutingZone

Come vedremo a breve, la Routing Table di una rete Kad è un albero binario proprio e questo significa che ogni nodo ha o 0 o 2 figli. Non è tuttavia un albero binario perfetto e quindi non è detto che tutti i nodi foglia abbiano la stessa profondità. Ogni nodo di questo albero binario è rappresentato da una RoutingZone.

Siamo quindi arrivati a descrivere la classe `RoutingZone.java` che, oltre a rappresentare un nodo della nostra Routing Table, è fornita di alcuni metodi statici che le permettono di eseguire dei task sulla Routing Table, la quale può essere pensata come l'insieme di tutte le RoutingZone.

Si riportano come sempre le variabili della classe `RoutingZone.java` con una loro breve descrizione (tabelle 3.8 e 3.9).

La funzione delle varie variabili è abbastanza intuitiva, eccetto forse per la variabile `m_uZoneIndex`. Essa ha il compito di tenere in memoria per ogni RoutingZone il prefisso comune dei contatti presenti nei suoi figli o nella sua RoutingBin. Se un KadContact ha la variabile `contactDistance` a 110001001011...01010001 allora si può trovare per esempio nel bucket gestito da una RoutingZone con `m_uZoneIndex` uguale a 0000...001100 oppure 0000...0011000100. Si noti che nel primo caso i 4 bit meno significativi di `m_uZoneIndex` tengono in memoria i 4 bit più significativi di `contactDistance`, mentre nel secondo caso `m_uZoneIndex`

salva gli 8 MSB di `contactDistance` nei suoi 8 LSB. In figura 3.3 è riportata una semplice Routing Table in cui si è evidenziato il valore della variabile `m_uZoneIndex` per ogni RoutingZone.

tipo	**nome variabile**	**descrizione**
RoutingZone	`m_pSuperZone`	riferimento al nodo padre
RoutingZone	`m_pSubZones[0-1]`	riferimento ad entrambi figli se ci sono, altrimenti null
RoutingBin	`m_pBin`	riferimento alla RoutingBin se presente, altrimenti null
int	`m_uLevel`	profondità del nodo nell'albero (la radice ha `m_uLevel`=0)
Int128	`m_uZoneIndex`	memorizza, partendo dal suo LSB, il prefisso comune della variabile `contactDistance` dei contatti presenti nei suoi figli o nel suo bucket
long	`m_tNextBigTimer_millis`	quando è necessario eseguire il task di mantenimento onBigTimer
long	`m_tNextSmallTimer_millis`	quando è necessario eseguire il task di mantenimento onSmallTimer

Tabella 3.8: variabili di RoutingZone

tipo	**nome variabile**	**descrizione**
HashMap<Integer, RoutingZone>	`zone_map`	mappa di tutte le RoutingZone presenti
RoutingZone	`root`	riferimento al nodo root della Routing Table
Int128	`uMe`	il mio Int128

Tabella 3.9: variabili statiche di RoutingZone

L'obbiettivo finale della Routing Table è quello di organizzare, in una struttura dati efficiente, i KadContact con i quali veniamo in contatto. Essa cerca, inoltre, di tenere in memoria molti KadContact a noi vicini nella rete Kad, mentre solo pochi tra quelli che ci sono lontani. In questo modo, la Routing Table permetterà di conoscere molto bene il nostro vicinato e quindi di rispondere con precisione alle richieste di file, keyword, etc. provenienti da altri contatti. In sostanza,

si viene a creare nell'intera rete Kad una struttura molto efficiente in cui ogni KadContact conosce molto bene gli altri contatti intorno a lui e si fa carico di smistare le richieste a loro destinate.

Facciamo un esempio. Sia A un file, una keyword o un contatto che desideriamo trovare e che si trova vicino a noi (secondo la metrica XOR). Per avere informazioni su A è allora sufficiente inviare una richiesta al contatto più vicino ad A. Dato che conosciamo bene il nostro vicinato sarà probabilmente necessaria una richiesta ad un solo KadContact per ottenere notizie riguardo ad A. Se invece A si trova molto distante da noi non dobbiamo fare altro che inoltrare la nostra richiesta ad un paio di contatti che sono un po' più vicini a ciò che vogliamo trovare. Questi ultimi, anch'essi abbastanza distanti da A, non trovando le informazioni che noi cercavamo su A, non faranno altro che ritornarci una lista dei loro KadContact, i quali si trovano ancora più vicini ad A. Noi, allora, inoltreremo le nostre richieste anche a questi ultimi e così via finché non troveremo un contatto abbastanza vicino ad A da poterci dare le informazioni che cercavamo.

Questo modo in cui viene gestita la ricerca di A si chiama **iterative lookup** (si veda la sezione 4 in [3]). Non è questa la sede per spiegare con precisione il suo funzionamento, ma si è comunque ritenuto interessante riportarne una semplice descrizione, al fine di chiarire come la Routing Table di ogni contatto risulti fondamentale per compiere ricerche nella rete Kad. Quello che si vuole sottolineare è come la conoscenza maggiore dei contatti a noi vicini e minore di quelli a noi lontani, ci permetta di creare una DHT molto efficiente, la quale distribuisce in maniera equa le richieste tra tutti i suoi contatti.

3.4.1 Struttura della Routing Table

Chiarita l'importanza della Routing Table, si inizia a spiegare in dettaglio la gestione delle RoutingZone, le quali saranno necessarie per costruire la struttura dati ad albero caratteristica della Routing Table di Kad.

Nella Routing Table ogni RoutingZone può essere o un nodo interno all'albero, oppure una foglia. Questa differenza è importante, poichè le RoutingZone che hanno funzione di nodi interni hanno un riferimento ai loro due figli, ma non ad una RoutingBin, mentre quelle che sono delle foglie non hanno un riferimento ai loro figli, ma hanno un riferimento ad una RoutingBin. Per chiarezza, si riporta questa distinzione nella tabella 3.10, mentre in figura 3.2 viene riportata una

rappresentazione grafica di una RoutingZone e delle sue variabili; quest'ultima verrà utilizzata spesso negli esempi successivi.

tipo di RoutingZone	**ha 2 figli**	**ha una RoutingBin**
nodo interno	✓	✗
foglia	✗	✓

Tabella 3.10: tipi di RoutingZone

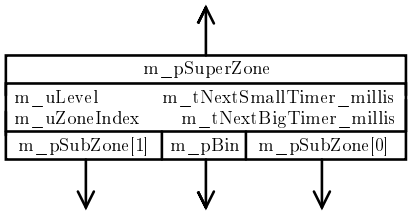

Figura 3.2: rappresentazione di una RoutingZone

È necessario ora comprendere in che modo i KadContact vengono inseriti nella Routing Table formata da tante RoutingZone. Quando si inizializza la Routing Table viene creata un'unica RoutingZone (che è il nodo `root`); essa non ha figli e quindi può contenere al massimo 10 KadContact nella sua RoutingBin. All'inserimento dell'undicesimo contatto il nodo `root` è costretto a fare uno **split**, cioè a generare due figli:

- `m_pSubZones[0]`: il figlio destro, che si occupa di tenere nel suo 10-Bucket i contatti che hanno il primo bit della variabile `contactDistance` a 0
- `m_pSubZones[1]`: il figlio sinistro, che si occupa di tenere nel suo 10-Bucket i contatti che hanno il primo bit della variabile `contactDistance` a 1

Dopo l'inserimento di altri KadContact, può accadere che anche uno di questi due figli si trovi ad avere la sua RoutingBin piena e quindi sia costretto a generare due nuovi figli. In tal caso queste due nuove RoutingZone, appena create, si occuperanno di gestire i KadContact che hanno il secondo bit della variabile `contactDistance` a 0 o 1 a secondo che siano il figlio destro o quello sinistro.

In generale, ogni nodo interno può essere paragonato ad un incrocio, il quale fa sì che i KadContact, con il bit di indice `m_uLevel`[3] della variabile `contactDistance`

[3] `root` ha `m_uLevel=0`, i suoi figli hanno `m_uLevel=1`, ...

uguale a 0, scendano per il percorso di destra, mentre quelli con il bit uguale a 1 scendano per il percorso di sinistra.

Per chiarezza, si riporta in figura 3.3 una semplice Routing Table in cui sono evidenziati i valori di alcune variabili relative a ciascuna RoutingZone.

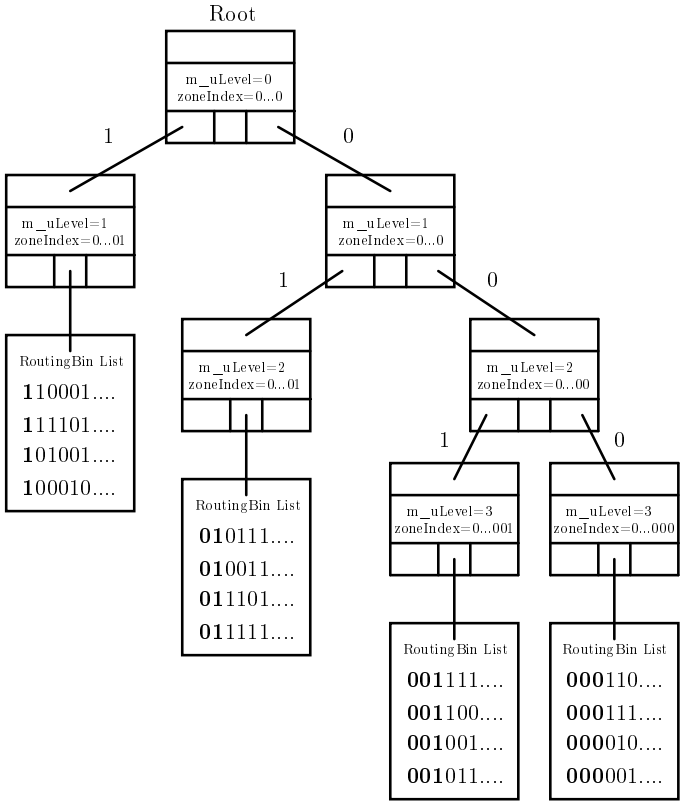

Figura 3.3: semplice RoutingTable composta da 7 RoutingZone; sono evidenziati in neretto i bit dei vari `contactDistance` che caratterizzano la rispettiva RoutingBin

Struttura Reale

Un altro termine associato ad ogni RoutingZone è quello di **posizione**. La posizione indica quanti altri nodi potrebbero idealmente esistere alla destra di una RoutingZone alla stessa profondità. Per esempio, il nodo `root` è in posizione 0, il suo figlio sinistro è in posizione 1 e se esso generasse un altro nodo sinistro, quest'ultimo sarebbe in posizione 3, poiché idealmente potrebbero esistere altre 3 RoutingZone alla sua destra a quella profondità. Ad ogni livello di profondità

possono esistere al massimo 2^{m_uLevel} RoutingZone e le loro posizioni varieranno nel range $[0, 2^{m\_uLevel} - 1]$.

Il concetto di posizione è stato introdotto per spiegare in che modo è strutturata una vera Routing Table in una rete Kad. Non tutte le RoutingZone hanno infatti il permesso di generare figli, poichè esistono dei vincoli che ogni RoutingZone deve rispettare per avere il permesso di fare uno split.

Le RoutingZone con `m_uLevel` uguale a 0, 1, 2 o 3 possono fare lo split senza problemi, mentre quelle con `m_uLevel`>=4 sono sottoposte ad un vincolo: se hanno una posizione minore o uguale a 4, allora possono fare lo split, altrimenti no. Infine, le RoutingZone con `m_uLevel`>=128 non possono fare lo split in nessun caso, poichè si trovano in fondo alla Routing Table e perciò gestiscono già l'ultimo bit della variabile `contactDistance` dei KadContact, rendendo inutile un ulteriore split. In questo modo, si tenderà a conoscere molto bene i KadContact che ci stanno vicini nelle rete Kad e meno bene quelli che ci stanno lontani. L'obbiettivo di queste due regole è quello di minimizzare lo spreco di memoria dovuto al salvataggio di un numero eccessivo di contatti troppo distanti da noi per esserci utili.

Nelle seguenti tre figure (3.4, 3.5 e 3.6) viene chiarito in che modo questi vincoli, appena descritti, influiscano sulla struttura dati ad albero della Routing Table.

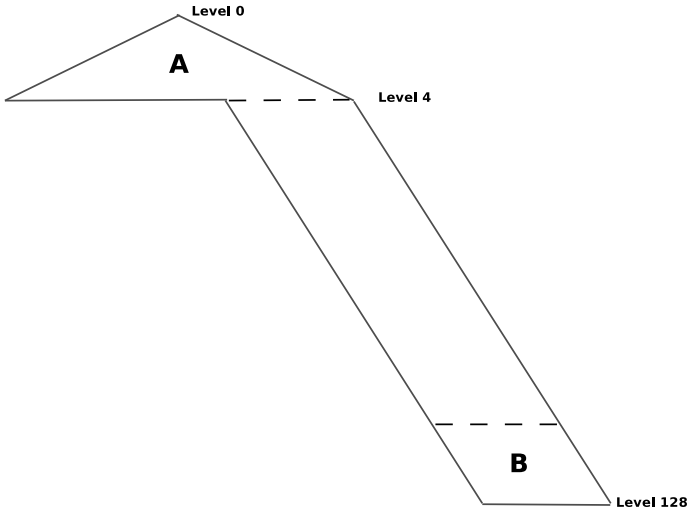

Figura 3.4: silhouette di una Kad Routing Table completa. La parte A e B sono presentate in dettaglio in figura 3.5 e 3.6

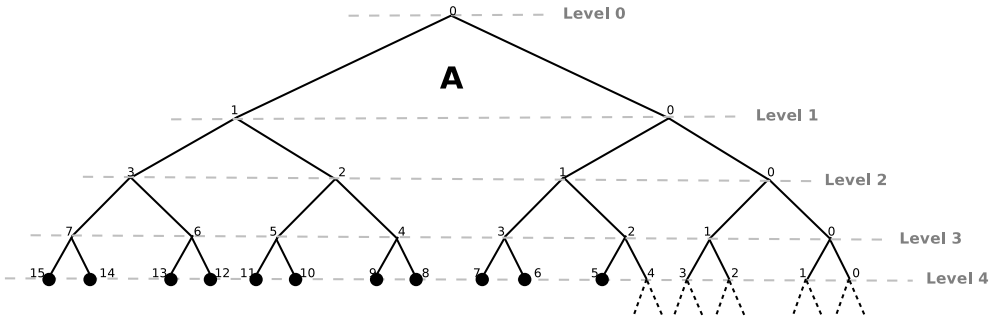

Figura 3.5: primi 5 livelli di una Routing Table. I puntini in neretto indicano la presenza di una RoutingBin. Al livello 4 solo i nodi in posizione minore uguale a 4 possono generare altri figli

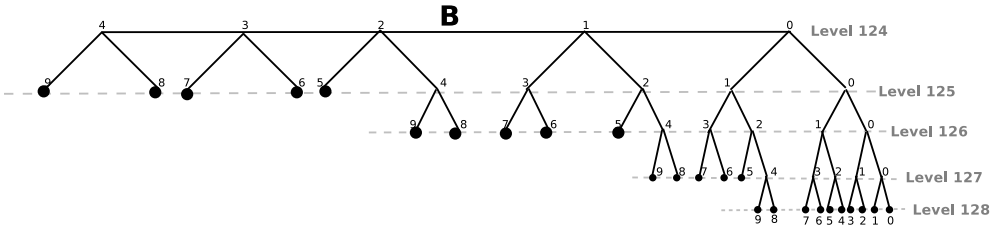

Figura 3.6: parte bassa di una Routing Table (in genere non si arriva mai così in basso)

3.5 Task di Mantenimento

Alla creazione di ogni RoutingZone vengono impostati i valori di due variabili (`m_tNextSmallTimer_millis` e `m_tNextBigTimer_millis`), le quali permettono di scadenzare l'esecuzione dei task di mantenimento `OnSmallTimer` e `OnBigTimer` su quella particolare RoutingZone.

La variabile `m_tNextSmallTimer_millis` viene settata ad un valore che dipende dagli ultimi 32 bit (4 byte) della `m_uZoneIndex` associata alla sua RoutingZone. In questo modo, i nodi più vicini a noi vengono mantenuti prima.

```
m_tNextSmallTimer_millis=System.currentTimeMillis() +
                         SEC2MILLIS(m_uZoneIndex.get32BitChunk(3));
```

Il task `OnBigTimer` viene invece eseguito molto meno spesso di `OnSmallTimer`, per questo non è necessario adoperare particolare accorgimenti che evitino l'affollarsi di task `OnBigTimer` nello stesso momento. Di conseguenza, la variabile `m_tNextBigTimer_millis` viene semplicemente impostata a 10 secondi dopo la creazione della RoutingZone:

```
m_tNextBigTimer_millis=System.currentTimeMillis()+SEC2MILLIS(10);
```

3.5.1 OnSmallTimer

La funzione del task `OnSmallTimer` è quella di fare pulizia dei contatti che non rispondo più alle nostre richieste. Esso viene eseguito ogni secondo su tutte le RoutingZone che soddisfano i seguenti requisiti:

- il valore di `m_tNextSmallTimer_millis <= System.currentTimeMillis()`
- è una foglia

Per controllare che il contatto sia ancora attivo si prova ad inviare un pacchetto KAD2_HELLO_REQUEST o KAD_HELLO_REQUEST al KadContact che sta in cima alla RoutingBin di ogni RoutingZone, la quale verifichi le condizioni sopra esposte. Se il contatto risponde, esso viene spinto in coda alla sua RoutingBin e il `m_tNextSmallTimer_millis` di quella RoutingZone viene incrementato di 1 minuto.

3.5.2 OnBigTimer

Il task `OnBigTimer` è chiamato ogni 10 secondi e ha l'obbiettivo di popolare la Routing Table di nuovi contatti. Questo viene eseguito solo sulla prima RoutingZone trovata che soddisfa i seguenti requisiti:

- il valore di `m_tNextBigTimer_millis <= System.currentTimeMillis()` oppure se è da almeno 15 minuti che non si aggiunge un nuovo contatto nella Routing Table
- è una foglia
- e verifica almeno 1 delle seguenti condizioni
  - ha una posizione minore o uguale a 4 e può quindi fare lo split per generare due nuovi figli
  - ha `m_uLevel < 4`, cioè si trova nella parte alta della Routing Table e può anch'esso fare lo split

– ha una RoutingBin con almeno 8 posti liberi; se le due ipotesi precedenti non sono verificate ma questa sì, vuol dire che la RoutingZone non può fare lo split, ma che a comunque una RoutingBin quasi vuota da riempire

Una volta eseguito questo task su una particolare RoutingZone, la rispettiva variabile `m_tNextBigTimer_millis` viene impostata ad 1 ora in avanti da quel momento.

3.5.3 Consolidate

Se una RoutingZone A ha due RoutingZone figlie (B e C), le quali sono delle foglie nella Routing Table e contengono meno di 5 KadContact insieme, allora questi due figli vengono cancellati e i loro KadContact vengono inseriti in una nuova RoutingBin collegata ad A (figura 3.7).

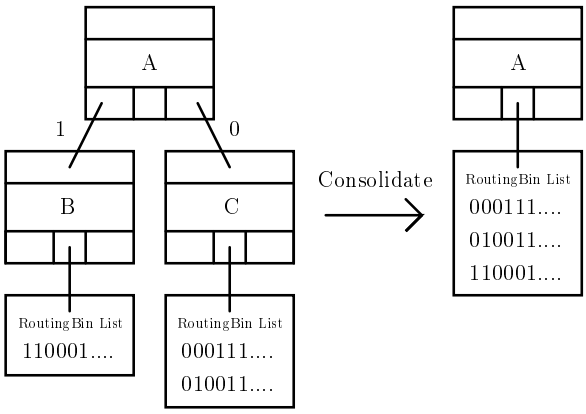

Figura 3.7: esempio di Consolidate prima e dopo

Il processo di consolidamento viene eseguito ricorsivamente iniziando dalle RoutingZone foglie e salendo fino alla radice. In questo modo, si compatta la Routing Table per renderla più leggera ed efficiente.

Questo task viene eseguito ogni 45 minuti.

Capitolo 4

PariKad

PariKad è un'implementazione scritta in Java di una rete Kad e si basa sull'implementazione in C/C++ di Kad in eMule 0.49c. È strutturata in modo tale che un unico thread principale si occupi di gestire i vari compiti necessari per il corretto funzionamento di PariKad.

4.1 Thread principale

Il thread principale svolge 5 funzioni fondamentali (di cui 2 non ancora completamente implementate):

- Permette di avviare e fermare Kad attraverso due semplici chiamate PariKad.start() e PariKad.stop(), le quali si occupano rispettivamente di inizializzare la struttura di Kad e di fare le dovute pulizie alla sua chiusura. All'avvio PariKad cerca di riempire la sua Routing Table attraverso contatti reperiti o dal file *boostrat.dat*, in cui sono salvati dei contatti alla chiusura del programma, oppure dal file *nodes.dat* presente nella sua cartella, oppure scaricandone uno direttamente da Internet. Alla chiusura, il thread principale si occupa di salvare il file bootstrap.dat con 200 contatti ed elimina la lista delle ricerche in corso e del publishining/indexing.

- Gestisce i task di mantenimento (vedi sezione 3.5) legati alla Routing Table.

- Si dovrebbe occupare di avviare l'UDPListener, un thread che permette di inviare e ricevere pacchetti UDP. In pratica questa funzione è delegata a

PariMulo che, oltre ad aspettare i suoi messaggi, resta in attesa pure di quelli di PariKad e glieli passa quando arrivano.

- Si occupa di avviare un thread che gestisce la pubblicazione dei nostri file. Non ancora completamente implementato.
- Si occupa di avviare un thread che gestisce il salvataggio delle keyword associate ai file presenti nella rete Kad. In tal modo, PariKad si prende carico di alcuni file che hanno un ID a noi vicino e può comunicare dove essi si trovano a chi ne è interessato. Non ancora implementato.

4.2 Sviluppi futuri

Dopo circa 6 mesi di lavoro sul progetto PariKad, siamo arrivati ad un buon punto di sviluppo: è possibile fare ricerche di file nella rete e quindi scaricare questi ultimi attraverso PariMulo; si gestiscono i contatti nella nostra Routing Table; si risponde, infine, alle principali richieste che vengono inoltrate da altri utenti. Al 20 Aprile 2009 gli sviluppi futuri riguardano:

- implementazione del *Find Buddy*, il quale permette di ottenere informazioni dalla rete Kad anche a chi si trova bloccato da un NAT (si veda il paragrafo 3.3.4 in [3]).
- modifica di alcuni metodi della RoutingZone, per aggiungere un sistema di protezione contro possibili attacchi che potrebbero venire lanciati contro la rete Kad. Si descrive sotto, un po' più in dettaglio, questo problema che deve assolutamente essere affrontato.
- test accurato dei metodi, delle classi e del funzionamento generale di PariKad. Al 20 Aprile 2009, i vari test coprono circa il 70% del codice scritto. Inoltre, anche i test che verificano il funzionamento globale di PariKad danno esiti molto positivi.

4.2.1 Vulnerabilità della rete Kad

È di cruciale importanza fornire a Kad un sistema per proteggere i vari utenti da possibili attacchi. Data la natura distribuita di ogni DHT, le sue vulnerabilità sono molteplici e possono causare seri problemi alla rete stessa.

L'attacco più pericoloso è senza dubbio il **Sybil Attack**. L'idea di questo attacco è quella di introdurre nella rete dei peer malevoli, i sybils appunto, i quali sono tutti controllati dalla stessa entità. I sybils, posizionati in modo strategico, permettono di ottenere il controllo di una frazione della rete peer-to-peer o perfino di tutta la rete. I sybils possono quindi monitorare il traffico generato dagli altri peer o possono interagire nella rete, instradando i pacchetti a peer sbagliati o ad altri sybils.

Come si può vedere in tabella 4.1, le ultime versioni di eMule/aMule hanno aggiunto sistemi sempre più avanzati per proteggere la rete Kad da questi possibili attacchi.

Clients	Flood protection	IP limitation	IP verification
eMule 0.48a / aMule 2.1.3	✗	✗	✗
eMule 0.49a / aMule 2.2.1	✓	✓	✗
eMule 0.49b / aMule 2.2.2	✓	✓	✓

Tabella 4.1: protezioni attivate nella rete Kad in varie versioni di eMule/aMule

Si sono infatti introdotti vincoli sempre più stringenti relativi al legame indirizzo ip - int128 dei vari client. Non si permette, cioè, a utenti che sono molto vicini geograficamente (stesso indirizzo ip, o comunque molto simile), di andare a posizionarsi in punti vicini della rete Kad, evitando così che client lanciati dalla stessa persona/azienda siano in grado di prendere il controllo della rete.

Per maggiori informazioni a riguardo si rimanda a [5] e [7].

Capitolo 5

Conclusioni

La rapida diffusione di connessioni Internet a banda larga ha portato ad una crescita esponenziale del fenomeno di *file-sharing*: applicazioni come eMule/aMule si appoggiano ogni giorno su una rete con più di 1,5 milioni di utenti connessi simultaneamente.

In questo contesto, in cui moltissimi software offrono mezzi per scaricare e condividere file tra utenti, anche PariMulo vuole trovare il suo spazio.

La necessità di rendere competitivo PariMulo ha portato alla nascita di PariKad, il quale integra, in PariMulo, una rete Kad per la condivisione di file e permette a PariMulo di competere allo stesso livello con altri software open-source più famosi.

A causa della scarsa letteratura sull'argomento, siamo stati costretti a studiare approfonditamente il codice di eMule, il quale si è rivelato l'unica fonte di informazioni attendibili. Nonostante le difficoltà incontrate nel comprendere il funzionamento di una rete Kad, il lavoro svolto in questi 6 mesi ha portato ad ottimi risultati.

Questa tesi nasce proprio sulla base delle conoscenze acquisite durante questa esperienza e dalla convinzione che questa materia necessiti una maggiore documentazione. A tal proposito, siamo convinti che il nostro lavoro sarà un utile punto di riferimento per chiunque cerchi informazioni più dettagliate sul funzionamento della Routing Table di una rete Kad.

Appendice A

Relazione PariKad - eMule

Si riporta di seguito la relazione che c'è tra alcune classi di PariKad con quelle di eMule 0.49c, dalle quali si è preso spunto. Infatti, nella struttura dati di PariKad si è cercato di usare una nomenclatura per le variabili, per i metodi e per le classi il più possibile uguale o simile a quella di eMule. L'obbiettivo evidente è quello di aiutare coloro che per la prima volta si appresteranno a leggere il codice di eMule e di PariKad.

PariKad	**eMule 0.49c**
`Int128.java`	`UInt128.h / UInt128.cpp`
`KadContact.java`	`Contact.h / Contact.cpp`
`RoutingBin.java`	`RoutingBin.h / RoutingBin.cpp`
`RoutingZone.java`	`RoutingZone.h / RoutingZone.cpp`
`PariKad.java`	`Kademlia.h / Kademlia.cpp`
`KadUDPKey.java`	`KadUDPKey.h / KadUDPKey.cpp`
`Bootstrap.java`	`ReadFile() in RoutingZone.cpp`
`unobfuscatePacket() in Packets.java`	`DecryptReceivedClient() in EncryptedDatagramSocket.cpp`

Tabella A.1: relazione classi PariKad e eMule 0.49c

Si è cercato di creare anche i metodi delle varie classi uguali a quelli di eMule. Ovviamente, dove possibile, sono state sfruttate le potenzialità che Java offre, come le classi standard per la gestione degli indirizzi IP e per le strutture dati, quali le Liste e le Mappe Hash.

Bibliografia

[1] PariPari. http://paripari.it.

[2] eMule 0.49c. http://www.emule-project.net.

[3] René Brunner. A performance evaluation of the kad-protocol. Master's thesis, Universität Mannheim, Germany, 2006.

[4] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. pages 53–65, 2002.

[5] Taoufik En-Najjary Moritz Steiner and Ernst W. Biersack. Exploiting kad: Possible uses and misuses. 2007.

[6] Moritz Steiner, Ernst W Biersack, and Taoufik En-Najjary. Actively monitoring peers in kad. In *IPTPS'07, 6th International Workshop on Peer-to-Peer Systems, February 26-27, 2007, Bellevue, USA*, 02 2007.

[7] Isabelle Chrisment Thibault Cholez and Olivier Festor. Evaluation of sybil attacks protection schemes in kad. 2009.

[8] InetAddressLocator. http://javainetlocator.sourceforge.net/.

Elenco delle figure

Elenco delle tabelle