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

PARIDHT: GESTIONE DEI CONTATTI

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Dott. Michele Bonazza

LAUREANDO: *Davide Zanin*

ANNO ACCADEMICO 2010/2011

Alla mia famiglia.

Prefazione

Negli ultimi anni l'aumento delle connessioni a banda larga ha portato alla diffusione delle reti *peer-to-peer*[1] (P2P, d'ora in avanti). Applicazioni quali *BitTorrent*[2], *eMule*[3], *Skype*[4] e *Tor*[5], permettono agli utenti di condividere file, conversare e tutelare la propria privacy proprio grazie alle reti P2P.

PariPari si inserisce in questo contesto offrendo questi ed altri servizi attraverso un unico software, ponendo rimedio ai conflitti che sorgono se si eseguono contemporaneamente applicazioni come quelle citate. PariPari vuole, inoltre, essere un servizio completamente decentralizzato, cioè un servizio che non necessiti di alcun server per funzionare. Questi, insieme alla facilità di utilizzo e alla garanzia di anonimato, sono i punti di forza del progetto.

Per raggiungere questi obiettivi è fondamentale basarsi su una rete che sia il più efficiente e sicura possibile, inoltre è di primaria importanza scegliere accuratamente quali sono i contatti della rete che verranno usati per reperire le informazioni desiderate. Senza alcuna precauzione basterebbero pochi utenti malintenzionati per danneggiare l'intero sistema, ad esempio oscurando una risorsa, fornendo risultati errati ad una richiesta oppure isolando un utente. Una cattiva gestione dei contatti influirebbe, inoltre, in maniera molto negativa sulle prestazioni, basti pensare a quanto tempo verrebbe sprecato cercando di comunicare con un contatto che non è più connesso alla rete.

Indice

Introduzione

In questo elaborato verrà inizialmente introdotto il progetto *PariPari*[6] esponendone gli obiettivi, l'architettura e l'organizzazione, ma senza soffermarsi troppo nei dettagli. Nel capitolo 2, verrà presentata l'evoluzione delle reti P2P fino ad arrivare alle odierne *DHT*[7]. Successivamente verrà descritta la rete *Kademlia*[8], rete che è stata presa come riferimento per lo sviluppo di *PariDHT*[9], la DHT di PariPari. L'ultima parte del capitolo è dedicata a PariDHT, in particolare verranno sottolineate le differenze presenti rispetto a Kademlia e descritti gli algoritmi di salvataggio e ricerca di una risorsa.

Dopo i due capitoli introduttivi, nel terzo, si parlerà della gestione dei contatti. Si descriveranno le differenze tra *contatto* e *nodo*, verranno esposti i requisiti funzionali ed infine verranno discussi i problemi affrontati e le tecniche adottate per risolverli. Il capitolo 4 è strettamente legato al capitolo 3: verrà infatti presentata l'implementazione dei meccanismi descritti nel capitolo precedente, senza però entrare troppo nei dettagli implementati ma cercando di dare un'idea generale di come si è deciso di sviluppare il codice.

Infine nell'ultimo capitolo verranno presentate le conclusioni, i problemi ancora aperti e gli sviluppi futuri.

Capitolo 1

PariPari

L'obiettivo principale del progetto PariPari è quello di creare un'applicazione multifunzionale che fornisca i più comuni servizi che si possono trovare su Internet, come *file sharing*, *VoIP*, *IRC*, *Web Server*, garantendo l'anonimato degli utenti e realizzando un sistemi di crediti transitivo.

PariPari è basato su una rete distribuita completamente decentralizzata di tipo *peer-to-peer*[1], cioè una rete in cui i nodi che la compongono non sono organizzati secondo un modello *client-server* (cliente-servente) ma seguendo un approccio paritario in cui ciascun nodo (*peer*) svolge sia il ruolo di cliente che di servente. La rete, inoltre, essendo completamente decentralizzata, non ha bisogno di un server per permettere ai vari nodi di tenersi in contatto tra di loro.

Disporre di una rete con queste caratteristiche è fondamentale per raggiungere l'obiettivo del progetto. La rete di PariPari è basata su una variante di *Kademlia*[8], ma di Kademlia si parlerà con maggior dettaglio in 2.3.

1.1 Architettura

Per migliorare la gestione, lo sviluppo e l'espandibilità, PariPari è fin da subito stato pensato come un'applicazione modulare. Ciascun modulo (*plug-in*) è a sé

Figura 1.1: Il logo di *PariPari*

stante e svolge un compito ben preciso, anche se, i vari plug-in, possono cooperare tra loro attraverso lo scambio di messaggi. *Core*[10] è il modulo centrale ed ha il compito di gestire il caricamento degli altri moduli e le comunicazioni tra questi.

L'insieme dei moduli indispensabili per il funzionamento di PariPari —detto *cerchia interna*— comprende, oltre che a Core, i seguenti plug-in:

Connectivity[11]. Gestisce le comunicazioni tra i nodi di PariPari e amministra la banda disponibile in ingresso e in uscita suddividendola tra i vari plug-in.

DHT[9]. Realizza una rete distribuita che collega i nodi tra di loro; permette agli altri plug-in di salvare e ricercare risorse senza la necessità di un server centralizzato.

Local Storage[12]. Rende disponibile l'accesso alla memoria di massa, applicando opportune regole che riguardano il numero e la posizione dei file che possono essere memorizzati.

Credits[13]. È la moneta di scambio tra i plug-in (*crediti interni*) e i nodi della rete (*crediti esterni*). Ogni servizio che viene richiesto ha un costo, quindi i plug-in fornendo servizi acquisiscono moneta che poi possono spendere per richiederne degli altri. Lo stesso ragionamento viene applicato anche ai servizi offerti e richiesti tra nodi della rete.

Gli altri plug-in formano la *cerchia esterna* e forniscono servizi più vicini all'utente. I principali sono:

Torrent[14]. Fornisce un client che implementa il protocollo *BitTorrent*[2].

Mulo[15]. Implementa le principali funzionalità di un qualsiasi client della rete *eDonkey2000*[16].

VoIP[17]. Permette di effettuare chiamate telefoniche attraverso Internet.

IRC[18] e IM[19]. Permettono a due o più utenti di comunicare i modo istantaneo (*chat*). I due plug-in implementano due protocolli diversi.

GUI[20]. Fornisce un'interfaccia grafica a tutti i plug-in.

Distribuited Storage[21]. Permette di salvare, ricercare e cancellare un file sulla rete.

Database[22] lo scopo di questo plug-in è quello di realizzare un *DBMS* distribuito utilizzando la rete di PariPari.

NTP[23]. Crea un sistema che permette a tutta la rete di sincronizzarsi su un unico affidabile orario.

Web Server[24]. Permette di disporre di un *Web server* distribuito.

1.2 Gruppo

PariPari è sviluppato da studenti dell'Università di Padova dei corsi di laurea triennale e magistrale, in maggior parte del Dipartimento di Ingegneria dell'Informazione . A capo del progetto si trovano i professori Enoch Peserico e Paolo Bertasi. Gli studenti sono divisi in gruppi, ognuno dei quali segue lo sviluppo e il testing di un plug-in. All'interno di ciascun gruppo si trova un *team leader* il cui compito è quello di gestire lo sviluppo e le comunicazioni con gli altri plug-in.

Per cercare di minimizzare il numero di errori presenti nel codice, è stata adotta una metodologia di programmazione agile chiamata *Extreme Programming*[25]. Gli aspetti principali di questa metodologia sono la verifica continua del programma durante lo sviluppo per mezzo di programmi di test e la frequente reingegnerizzazione del software, di solito in piccoli passi incrementali.

Per applicare al meglio questa metodologia sarebbe necessario dividere in due gruppi gli studenti assegnati a ciascun plug-in. Del primo gruppo farebbero parte gli *sviluppatori*, con il compito di sviluppare il codice, mentre del secondo farebbero parte i *tester*, con il compito di scrivere i test per il codice prodotto dagli sviluppatori. Purtroppo il numero di studenti che partecipano al progetto (circa sessanta) non permette questa distinzione, quindi ciascuno studente svolgerà sia i compiti di sviluppatore che di tester, ma con la regola che nessuno sviluppatore testerà il proprio codice, altrimenti il riconoscimento di eventuali bug risulterebbe più difficoltoso.

1.3 Implementazione

PariPari è interamente implementato in *Java*[26]; questo linguaggio è stato scelto principalmente per tre motivi:

- Propende fortemente verso una programmazione strutturata ed è orientato alla programmazione ad oggetti.
- Permette una portabilità immediata tra i vari sistemi operativi e permette l'integrazione dell'applicazione con i browser web attraverso *Java Web Start*[27].
- Viene insegnato agli studenti nei primi corsi di informatica all'Università di Padova ed ha una maggiore facilità di apprendimento rispetto ad altri linguaggi considerati, come ad esempio *C++*[28].

Tuttavia, a differenza di C++, Java risente di alcuni limiti come l'impossibilità di gestire direttamente le locazioni di memoria, il supporto non nativo della rappresentazione numerica priva di segno e le prestazioni non eccellenti in determinate operazioni[1].

[1] Ad esempio quando è necessario compiere una grande mole di calcoli, come in crittografia.

Capitolo 2

PariDHT

In questo capitolo si parlerà dell'evoluzione delle reti *P2P* fino a giungere alle attuali *DHT*; verranno quindi analizzate le reti *Kademlia* e *PariDHT*, la DHT alla base di PariPari.

2.1 Le reti P2P

Come già detto una rete P2P è una rete in cui i nodi[1] non sono divisi in *client* e *server* ma sono tutti equivalenti e svolgono compiti sia di cliente che di servente.

I vantaggi e svantaggi di una rete di questo tipo sono principalmente i seguenti:

- Non è necessario un server con potenzialità elevate, quindi non bisogna sostenerne i costi per l'acquisto e la manutenzione. D'altra parte, però, ogni computer della rete deve avere i requisiti per sostenere l'utente in locale e in rete, ma anche eventuali altri utenti che desiderassero accedere alle risorse del computer.

- Ogni utente condivide le proprie risorse, ed è quindi l'amministratore del proprio client-server. Questo ha però lo svantaggio di richiedere delle competenze ad ogni utente soprattutto per quanto riguarda la protezione.

- La velocità media di trasmissione dei dati è potenzialmente molto più elevata di una classica rete client-server, infatti l'informazione richiesta da un client può essere reperita da molti altri nodi, anziché da un unico server. Questo tipo di condivisione diventa tanto più efficace tanti più sono i client connessi, in contrapposizione alla rete tradizionale in cui un elevato numero di client connessi riduce la velocità di trasmissione dei dati per utente.

- La rete è più robusta. In una rete client-server è sufficiente che il server non sia più connesso affinché la rete non sia più accessibile; in una rete P2P, invece, finché continueranno ad essere connessi un certo numero di nodi la rete continuerà a funzionare.

[1] I nodi in questo contesto vengono anche chiamati *peer*.

Le prime reti P2P si basavano su un modello *centralizzato*, *Napster*[29] ne è un esempio. Napster permetteva agli utenti di condividere file musicali attraverso alcuni server utilizzati per tenere in contatto i nodi tra loro ma che non prendeva parte nei trasferimenti di file[2]. All'avvio del programma ogni nodo inviava al server la lista di tutti i file musicali che intendeva condividere. Per richiedere un file bastava, quindi, inviare una query al server e questo rispondeva con un elenco di nodi che possedevano i file desiderati. A questo punto i nodi comunicavano tra loro per iniziare il trasferimento del file richiesto. Napster era quindi un sistema ibrido in cui la ricerca avveniva attraverso un sistema client-server mentre il trasferimento era più simile ad una rete P2P. I limiti di questo sistema sono evidenti: da una parte la poca scalabilità[3] dall'altra la scarsa robustezza[4].

Successivamente venne introdotto il modello *distribuito non strutturato*, di cui l'esempio più famoso è *Gnutella*[30]. In questo caso tutte le operazioni, ricerca, avvio e trasferimento, avvenivano in modo distribuito, cioè senza l'uso di un server. Non essendo la rete strutturata le ricerche avvenivano in *broadcast*, veniva cioè inviato un messaggio a tutti i nodi che si conoscevano che, a loro volta, lo inviavano ai loro conoscenti fino a raggiungere il nodo o la risorsa desiderata. Per evitare che i messaggi circolassero per sempre sulla rete, a ciascuno di essi era associato un valore chiamato *TTL*[5] che indicava il massimo numero di nodi che poteva percorrere il messaggio. Ciascun nodo, inoltre, memorizzava temporaneamente l'identificativo dei messaggi che passavano attraverso di esso, così da scartare eventuali messaggi già ricevuti senza inviarli nuovamente ai propri conoscenti. Lo svantaggio di questa metodologia è di non garantire il determinismo di una ricerca e di non essere perfettamente scalabile.

Si giunse dunque allo sviluppo del modello *distribuito strutturato*, in cui la topologia della rete è ben definita, permettendo una maggiore scalabilità e affidabilità delle operazioni di ricerca e salvataggio. La strada seguita per ottenere questo risultato è l'utilizzo delle DHT[7], che porta i seguenti vantaggi:

Decentralizzazione. Non è necessario alcun coordinamento centrale, sono i nodi nel loro insieme a formare e gestire il sistema.

Scalabilità. Il sistema può essere utilizzato in modo efficiente anche in presenza di centinaia di milioni di nodi.

Tolleranza ai guasti. Il sistema, con i dovuti accorgimenti, è in grado di funzionare anche con nodi che si connettono e disconnettono continuamente dalla rete o che sono spesso soggetti a malfunzionamenti.

Protocolli che utilizzano le DHT sono, ad esempio, *Chord*[31], *BitTorent*[2] e Kademlia[8].

[2]Napster rimase attivo dal giugno 1999 fino a luglio 2001.
[3]Se gli utenti aumentassero sarebbe necessario potenziare il server.
[4]Se il server venisse disconnesso la rete non sarebbe più utilizzabile.
[5]*Time To Live*, tempo di vita.

2.2 Funzionamento di una DHT

DHT è un meccanismo che permette di ricercare e salvare delle risorse in una rete di calcolatori. In una DHT vengono memorizzate delle coppie <*chiave*, *valore*>, dove *chiave* è un identificatore univoco di *valore*, che rappresenta la risorsa che si vuole salvare nella rete. La ricerca avviene fornendo la chiave della risorsa da cercare. Sotto questo questo punto di vista una DHT è paragonabile ad un *hash table*[32] classica.

In una DHT, però, le coppie <*chiave*, *valore*> vengono distribuite il più uniformemente possibile tra i nodi della rete; agendo in questo modo si apre il problema della gestione di informazioni condivise ma, in compenso, si rende la rete più robusta e scalabile, e si evita che le informazioni e il lavoro per gestirle si concentri solo su pochi nodi. Sfruttando questa caratteristica e strutturando adeguatamente la rete si possono inoltre produrre algoritmi che permettono, nel caso medio, di trovare una risorsa in tempo logaritmico rispetto al numero di nodi di cui è composta la rete.

Quello che però caratterizza una DHT in maniera unica è l'*identificativo univoco* che viene assegnato a ciascun nodo. Tale identificativo viene anche detto *ID del nodo* e condivide lo stesso dominio delle chiavi delle risorse. Questa caratteristica permette di definire equivalentemente il concetto di vicinanza di un nodo rispetto ad una chiave o rispetto ad un altro nodo.

Per poter parlare in termini formali di vicinanza bisogna definire il concetto di *metrica* (o distanza). La metrica è una funzione matematica $d : X \times X \to \Re$ che soddisfa le seguenti proprietà:

1. $d(x, y) \geq 0 \quad \forall x, y \in X$
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) = d(y, x) \quad \forall x, y \in X$ *(proprietà di simmetria)*
4. $d(x, z) \leq d(x, y) + d(y, z) \quad \forall x, y, z \in X$ *(diseguaglianza triangolare)*

Nel caso delle DHT, X sarà il dominio delle chiavi delle risorse e quindi anche quello degli ID dei nodi. Con questa definizione quindi si può dire che un nodo con ID i è tanto più vicino ad una risorsa di chiave k tanto più piccolo è il valore di $d(i, k)$. Lo stesso ragionamento si applica per definire la vicinanza tra due nodi a e b, cioè i due nodi saranno tanto più vicini a mano a mano che il valore di $d(ID(a), ID(b))$ diminuirà. Data una risorsa r di chiave k, si definiscono inoltre, nodi *responsabili* della risorsa r, i nodi che si trovano il più vicino possibile a k.

Perché una DHT funzioni però bisogna strutturare la rete, bisogna cioè definire una regola che permetta a ciascun nodo di decidere quali nodi contattare per ottenere le informazioni che desidera o che vengono richieste da altri nodi. Questo aspetto è fondamentale per ottenere prestazioni logaritmiche.

Per ottenere tali prestazioni, le linee guida che un nodo n interrogato durante la ricerca di una risorsa di chiave k deve seguire sono le seguenti:

- Se n conosce un nodo m più vicino a k di quanto esso stesso non lo sia, allora invia le informazioni per contattare m al mittente.
- Se un nodo più vicino a k di n non esiste, allora n è responsabile di k, quindi n può restituire al mittente l'eventuale valore associato a k.

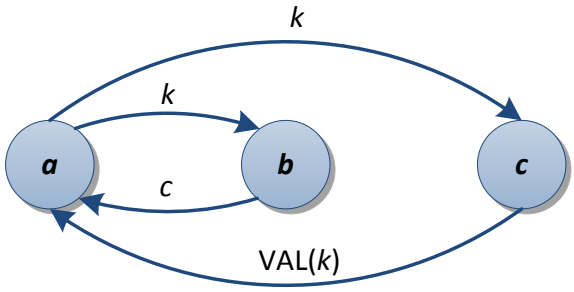

Figura 2.1: Ricerca di una chiave k.

Supponiamo che il nodo a (figura 2.1) voglia conoscere il valore associato alla chiave k. Per prima cosa sceglie il nodo che conosce più vicino a k, in questo caso b, e lo interroga. Il nodo b conosce un nodo, c, che è più vicino a k, allora invia ad a le informazioni necessarie per contattarlo. Una volta ricevute tali informazioni, a invia la richiesta a c; quest'ultimo, non conoscendo nodi più vicini a k di lui, è il responsabile, per cui invia ad a il valore associato a k. Ovviamente può capire che non ci sia ancora nessun valore associato ad una certa chiave o che i valori siano più di uno. I dettagli della procedura di salvataggio e ricerca di una risorsa o di un nodo verranno esposti in 2.4.3, dopo aver descritto le reti Kademlia e PariDHT.

Esiste un algoritmo detto *algoritmo del figlio prediletto*, introdotto nelle tesi di Paolo Bertasi[33] e Simone Giacon[34] e poi implementato in *PariDHT* da Nicola Celli[35], che permette di accelerare le ricerca. Questo algoritmo ha però dei difetti tali da renderlo un servizio *best effort*, quindi non potrà sostituire l'algoritmo precedentemente illustrato ma solo affiancarlo.

2.3 Kademlia

Kademlia[8], da cui PariDHT trae spunto, è l'esempio più famoso di DHT. Questo protocollo è stato ideato da Petar Maymounkov e David Mazières e usa come metrica la funzione XOR, si ha, quindi, che $d(x, y) = x \oplus y$.

Questa metrica fornisce una topologia ad albero binario della rete. Se si considerano i nodi della rete come le foglie dell'albero, e si assegna 0 ai rami sinistri e 1 ai rami destri allora il percorso dalla radice al nodo ne definisce il suo identificativo.

In Kademlia ogni nodo ha ID $= b_{m-1}, b_{m-2}, \ldots, b_1, b_0$, con $m = 128$, quindi formato da 128 bit. Ogni nodo mantiene una lista di m *bucket*, detti k-*bucket*, in

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 2.1: Tabella della verità dell'operazione XOR.

cui memorizza le informazioni per contattare un certo nodo. Il generico bucket $i \in [0 \dots m-1]$ contiene nodi che hanno distanza d con $2^i \leqslant d < 2^{i+1}$, cioè include i nodi che hanno ID con gli stessi valori nei bit $b_{m-1}, b_{m-2}, \dots, b_{i+1}$, e un valore diverso nel bit b_i. Il simbolo k presente nel termine k-bucket, rappresenta una costante, generalmente posta pari a 20, che indica il numero massimo di nodi che può contenere ciascun bucket.

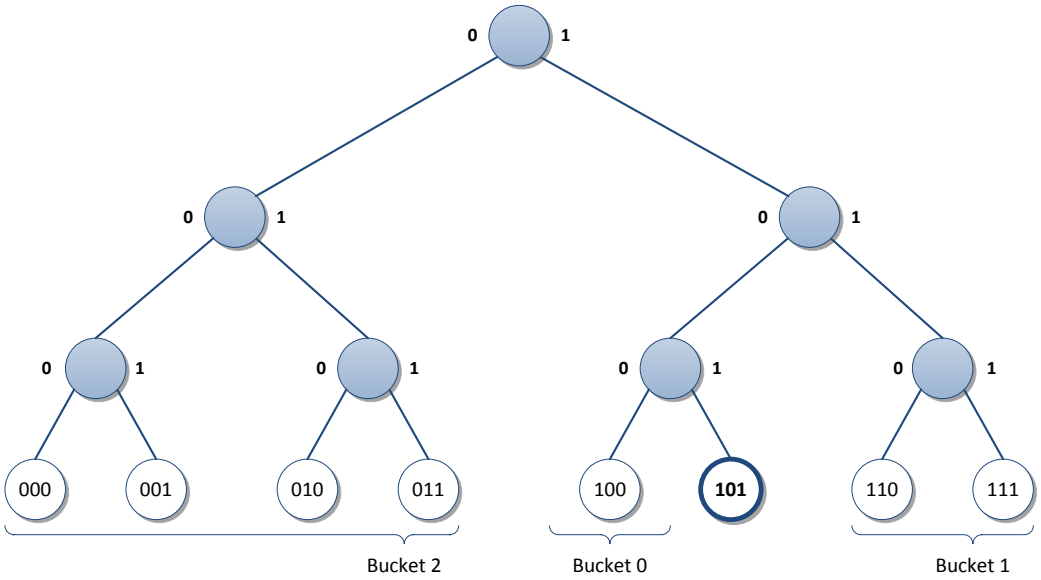

Figura 2.2: Partizionamento della rete rispetto ad un nodo con ID 101.

Siccome è noto[36] che nodi collegati da maggior tempo alla rete hanno maggior probabilità di rimaner connessi per un'altra ora, nei bucket di Kademlia vengono mantenuti i nodi connessi da maggior tempo; in questo modo aumenta la probabilità di avere nodi connessi nel futuro e rende la rete più stabile.

2.4 PariDHT

In questo paragrafo verranno esposti gli aspetti principali di PariDHT e gli algoritmi attualmente implementati per la ricerca e il salvataggio di una risorsa.

2.4.1 Requisiti funzionali

Nella realizzazione del plug-in si è cercato di rispettare il più possibile le seguenti specifiche:

Funzionalità. L'obiettivo principale è fornire un modulo DHT che dia la possibilità agli altri plug-in di pubblicare e reperire risorse dalla rete.

Espandibilità. Il plug-in deve essere il più modulare possibile in modo da rendere più semplici eventuali espansioni future.

Documentazione. Vista la frequenza con la quale i componenti del gruppo vengono cambiati, è molto importante che la documentazione sia di buona qualità. Per questo motivo si fa uso di *Javadoc*[37] per commentare il codice e della wiki[9] per spiegare come gli altri plug-in possono utilizzare PariDHT.

Prestazioni. Per una DHT è di fondamentale importanza valutare attentamente alcuni parametri come il numero e la dimensione dei messaggi scambiati tra i vari nodi. Il rischio è, altrimenti, che l'*overhead* generato dalla rete renda il plug-in lento e sconveniente per l'utente. Le prestazioni locali, valutate ad esempio come numero di cicli macchina o occupazione della memoria, possono essere inizialmente considerate di minor importanza e migliorate in fasi più avanzate dello sviluppo.

2.4.2 Struttura generale

In PariDHT un nodo è una tripletta <ID, IP, *porta*>. Da questo si ricava che non esiste una corrispondenza biunivoca tra nodi e macchine fisiche, infatti in una singola macchina potrebbero essere in esecuzione due istanze diverse di PariPari, anche se questo è abbastanza inutile, se non dannoso, per l'utente stesso. Per l'assegnazione degli ID viene usato l'algoritmo *SHA-256*[6][39]; in questo modo la probabilità di collisione[7] è estremamente bassa: gli ID infatti, sono composti da 256 bit[8]. Come in *Kademlia* viene utilizzato l'operazione di XOR come metrica.

Le risorse sono rappresentate con la tripletta <ID, *value, additionalInfo*>, in cui ciascun campo ha il seguente significato:

ID. Rappresenta la chiave della risorsa ed appartiene allo stesso dominio degli ID dei nodi.

value. È il valore, di qualsiasi tipo, che si vuole memorizzare nella rete.

[6]L'input dell'algoritmo è la chiave di una risorsa oppure le informazioni per la generazione dell'ID nel caso dei nodi. Quali siano queste informazioni verrà chiarito in 3.3 quando si parlerà di come calcolare l'ID di un nodo.

[7]Si ha una collisione quando a due nodi connessi diversi oppure risorse con chiavi diverse viene assegnato lo stesso ID.

[8]Con 256 bit è possibile generare $2^{256} \approx 1,16 \cdot 10^{77}$ valori diversi.

additionalInfo. È un campo che serve per meglio descrivere la risorsa. Attualmente viene inserita solo la scadenza della risorsa a cui fa riferimento, ma in futuro potrebbero essere inserite altre informazioni.

Il nodo che pubblica una certa risorsa viene detto *possessore* della risorsa, i nodi che hanno ID che è più vicino di tutti alla risorsa vengono detti *responsabili potenziali* della risorsa mentre quelli che effettivamente detengono la risorsa sono detti *responsabili effettivi* (o semplicemente responsabili).

Per svolgere le operazioni di una DHT, i nodi della rete comunicano tra loro con i seguenti messaggi:

- `STORE`. Viene utilizzato per richiedere al nodo destinatario di memorizzare una risorsa. Il destinatario può decidere se accettare o rifiutare la richiesta sulla base del proprio *load factor*[9].
- `FIND_NODE`. Richiede al destinatario di fornire la lista di nodi che conosce più vicini ad un certo ID.
- `SEARCH`. Chiede al destinatario di trasmettere la lista dei valori che hanno come chiave un certo ID. Se il nodo non è responsabile effettivo di quell'ID allora il risultato che fornisce è analogo a `FIND_NODE`. In questo modo il mittente può continuare la ricerca dei valori grazie ai nodi che gli sono stati forniti.
- `SEARCH_WITH_MASK`. Il comportamento è analogo a `SEARCH`, ma permette al destinatario di filtrare gli eventuali valori, applicando dei test, prima della restituzione.
- `PING`. Verifica se il nodo destinatario è ancora connesso alla rete.

2.4.3 Ricerca di un nodo e di una risorsa

Alla base del funzionamento di una DHT c'è la capacità di cercare un nodo nella rete. Si supponga infatti di dover salvare una risorsa. Si dovranno cercare i responsabili potenziali, cioè si dovranno trovare i nodi più vicini all'ID della risorsa. La ricerca di una risorsa avviane sostanzialmente allo stesso modo della ricerca di un nodo.

In PariDHT la procedura di ricerca dei nodi più vicini viene detta *look up*. L'equivalente di una look up in Kademlia restituisce un numero di nodi pari a k, cioè pari alla dimensione massima di un bucket, mentre in PariDHT il valore può essere specificato per ogni ricerca e quando non indicato viene usato quello di deafult, cioè $k/2$.

Sia k il numero di nodi vicini all'ID h che si vogliano ottenere, allora la procedura di look up si articola principalmente in tre passi:

[9] Il load factor è è dato dal numero di chiavi e valori che il nodo già memorizza.

1. Si crea una lista con i k nodi conosciuti più vicini ad h.
2. Si prendono i primi `ALPHA` nodi della lista, cioè i più vicini ad h, si contrassegnano e si invia a ciascuno di essi una `FIND_NODE(h)`. Ogni nodo quindi risponderà con i k nodi più vicini ad h che conosce.
3. Tra tutti i nuovi nodi ottenuti si inseriscono nella lista quelli opportuni[10]. Se nella lista ci sono ancora nodi non contrassegnati allora si ritorna al punto 2 altrimenti la procedura termina e si restituisce la lista.

Il parametro `ALPHA` rappresenta il numero massimo di richieste pendenti e viene utilizzato per limitare il traffico sulla rete.

La procedura di ricerca di una risorsa è molto simile, con la differenza che ai nodi vengono spediti dei messaggi di tipo `SEARCH` al posto di `FIND_NODE`.

2.4.4 Salvataggio di una risorsa

Anche per il salvataggio di una risorsa la procedura di look up riveste un ruolo fondamentale. Principalmente per la memorizzazione di una risorsa di ID h sono richiesti i seguenti passaggi:

1. Utilizzando la procedura di look up si ricavano i k nodi più vicini ad h.
2. Si selezionano i `RESPONSIBILITY_NODES` nodi più vicini ad h e si invia una `STORE(h)` a ciascuno di essi.

Come si può notare dal punto 2, il salvataggio di una risorsa viene eseguito su più nodi. Inoltre se una delle `STORE` dovesse fallire[11], deve essere scelto un altro nodo in modo tale che il numero totale di `STORE` eseguite con successo sia esattamente `RESPONSIBILITY_NODES`. Di default questo parametro è impostato a 3 ma può essere facilmente modificato attraverso il file di configurazione. Tale replica viene eseguita per permettere di trovare la risorsa anche nel caso in cui uno dei nodi responsabili si disconettesse.

[10] Non si inseriscono duplicati o nodi che non siano più vicini di quelli già presenti.

[11] La `STORE` fallisce se il nodo destinatario è disconnesso o non può accettare altre risorse.

Capitolo 3

Gestione dei contatti

Prima di parlare della *gestione dei contatti* bisogna precisare a cosa ci si riferisce con il termine *contatto*. Finora si è parlato di nodi della rete lasciando sottintendere che un nodo è caratterizzato dall'indirizzo di rete quindi da un IP e una porta. Però, a causa della scarsità di indirizzi IP, dovuta al protocollo *IPv4* [38], ciascun IP non è assegnato in modo univoco ad un dispositivo, ma viene scelto al momento della connessione ad Internet. Può quindi capitare che l'IP utilizzato da un certo utente venga poi assegnato ad un altro utente, nel momento in cui il primo si disconnette da Internet. Anche considerando la coppia <IP, porta> non si avrebbe la certezza di riuscire ad identificare un modo univoco un nodo.

Con il termine contatto ci si vuole quindi riferire ad una precisa esecuzione del plug-in. Se ad esempio viene avviato PariDHT e dopo un certo periodo di tempo si riavvia l'intero programma, allora le due esecuzioni definiscono due contatti diversi. Due nodi uguali, quindi, non è detto si riferiscano allo stesso contatto mentre quando si parla di *nuovo contatto* significa che si è connesso alla rete un nuovo utente che può anche avere lo stesso indirizzo di un vecchio contatto, oppure che lo stesso utente si è disconnesso e poi riconnesso.

I due attori principali impegnati nella gestione dei contatti sono il *nodo* e il *node storer*. Il nodo raccoglie tutte le informazioni conosciute che riguardano un certo contatto mentre il node storer immagazzina e fornisce i nodi. Un nodo viene quindi creato alla prima comunicazione con un contatto e finché non ci sarà motivo di credere che, anche se l'indirizzo di rete è lo stesso, il contatto è cambiato, il nodo raccoglierà tutte le informazioni di interesse che lo riguardano.

In questa introduzione non è stato considerato l'ID di un nodo perché, come si vedrà in 3.3, l'ID in generale non permette di capire se un contatto è cambiato: questa proprietà dipende, infatti, da come l'ID viene assegnato un nodo.

3.1 Requisiti funzionali

Per rendere la gestione dei contatti il più efficiente possibile sono stati individuati i seguenti requisiti funzionali:

- Deve essere possibile ottenere un nodo a partire dalla coppia <IP, porta>. Le informazioni contenute nel nodo devono essere valide cioè riferite al giusto contatto.
- Il nodo deve raccogliere tutte le informazioni note relative ad un certo contatto. Una qualsiasi altra entità che osservi una certo comportamento (ad esempio un contatto che non risponde ad una richiesta) deve informare il node storer di quello che è accaduto.
- Nel node storer devono essere presenti i k-bucket ed i nodi in esso contenuti devono essere sicuri cioè ancora connessi alla rete e non malvagi.
- Dato un identificativo *id* e un intero *n* deve essere possibile reperire la lista degli *n* nodi presenti nei k-bucket più vicini ad *id*; tale lista deve essere ordinata per distanza crescente da *id*.

Riassumendo, il node storer è una sorta di magazzino in cui vengono salvate tutte le informazioni conosciute riguardo una certa coppia <IP, porta>; queste informazioni formano il nodo. Nel node storer non possono esistere contemporaneamente due nodi con lo stesso indirizzo di rete, inoltre questi deve anche selezionare dei nodi che andranno a formare i k-bucket; tali nodi saranno poi usati per le operazioni fondamentali di PariDHT, come la ricerca di un nodo a partire dal suo identificativo.

3.2 Struttura

Per facilitarne la gestione, il node storer è stato diviso in due parti: i *bucket* e la *cache*. I bucket rappresentano i k-bucket come già sono stati descritti mentre la cache contiene tutti i nodi conosciuti che però, per qualche motivo, non fanno parte dei bucket.

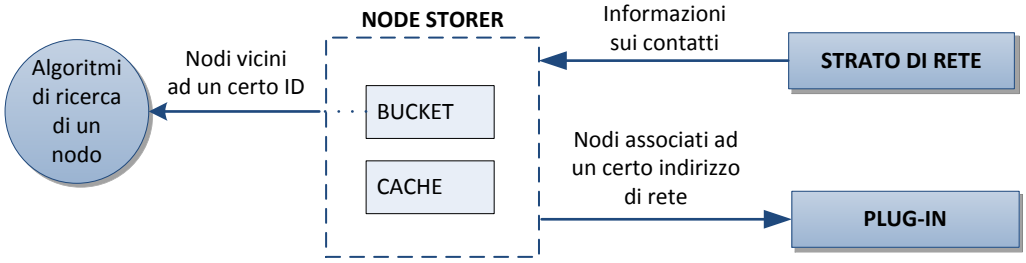

Figura 3.1: Struttura del node storer.

3.2.1 Bucket

Ciascun bucket è modellato come una lista in cui i nodi vengono sempre inseriti in coda. In questo modo in ogni momento un nodo è tanto più vicino alla testa

della coda tanto maggiore è il tempo passato da quando è stato inserito. Come già riportato, attualmente è prevista una dimensione massima di 20 per ciascun bucket.

Essendo gli identificativi formati da 256 bit si avranno 256 bucket; in ogni caso se in futuro sarà necessario aumentare il numero di bit dell'ID, e quindi il numero di bucket, dovrà essere possibile farlo con poco sforzo.

Siccome non possono esistere più nodi con lo stesso IP e porta ma diverso ID, è necessario disporre anche di un meccanismo per trovare velocemente un nodo dato il suo indirizzo di rete. Per questo motivo è presente anche una hash table che ha come chiave gli indirizzi di rete e come valori i nodi.

3.2.2 Cache

Come già accennato, nella cache vengono salvati tutti i nodi che per qualche motivo non possono entrare a far parte dei bucket.

È importante far notare fin da subito che la cache non può avere una dimensione infinita e che le informazioni raccolte, memorizzate nel nodo, avranno una scadenza, altrimenti si rischia di prendere decisioni basate su dati non più aggiornati. La cache sarà quindi una hash table con l'indirizzo di rete di un contatto come chiave e come valore il nodo che raccoglie tutte le informazioni conosciute.

3.3 Calcolo dell'ID di un nodo

Progettare adeguatamente l'algoritmo per il calcolo dell'identificativo di un nodo è fondamentale e per niente banale: è necessario, infatti, trovare un meccanismo che permetta di assegnare un ID diverso a ciascun contatto ma che non permetta ad un eventuale nodo malvagio di poter ottenere l'ID che preferisce. Se questo fosse possibile un nodo malvagio potrebbe scegliere una certa chiave da sabotare, ottenere un identificativo vicino alla chiave e decidere quali valori restituire alle richieste di valore che gli pervengono.

Purtroppo allo stato attuale non è ancora stata trovata una procedura che soddisfi completamente tutti i vincoli; è molto difficile trovare un algoritmo che sia sicuro e allo stesso tempo non troppo pesante, cioè che ad esempio non coinvolga comunicazioni sulla rete. Per rendere il problema più gestibile si è deciso di progettare due procedure per il calcolo dell'ID, una veloce ma debole, e una più lenta ma che offra maggiore sicurezza. L'ID calcolato con la prima si dice *debole* (*weak ID*) e quello calcolato con la seconda *sicuro* (*secure ID*).

3.3.1 ID sicuro

Il calcolo dell'ID sicuro è un problema ancora aperto. Nell'ultimo anno sono state fatte varie proposte ma tutte hanno almeno un difetto tale per cui non risultano essere sicure come si vorrebbe. L'ostacolo più difficoltoso da superare è che bisogna assegnare l'identificativo senza che il contatto interessato possa

influenzare il risultato a suo piacimento, ma che permetta anche agli altri nodi di verificarne l'autenticità, cioè permetta di capire quando un nodo malvagio sta fornendo un ID che non è quello che gli è stato assegnato. La procedura sicura dovrebbe anche essere in grado di dire se un certo nodo è già stato marcato come malvagio da altri nodi.

Una delle soluzioni più promettenti, proposta da Nicola Gobbo[40], è la seguente. Supponiamo che un certo nodo a si connetta alla rete e voglia quindi calcolare il proprio ID; al posto di generarselo da sé, a contatta un certo numero di nodi indipendenti, detti *garanti di primo livello*, che accordandosi tra loro gli forniranno un token T. A questo punto a userà T per calcolare il proprio ID seguendo un procedura prestabilita, individuerà i nodi vicini, detti *garanti di secondo livello*, e gli invierà il proprio ID e il token T; questi ultimi verificheranno la correttezza delle informazioni contattando i garanti di primo livello. Dopo questa fase i garanti di a saranno solo i garanti di secondo livello, quindi se un nodo b volesse verificare l'ID di a dovrebbe interrogare i vicini di a. Il motivo per cui tale procedura non è stata implementata è che il suo corretto funzionamento dipende dalla sicurezza della procedura di look up e, purtroppo, al momento non è ancora stato trovato un modo per impedire ad un nodo malevolo di prendere controllo della look up dirottandola verso altri nodi malvagi.

3.3.2 ID debole

Nelle primissime versioni di PariDHT l'identificativo di un nodo era calcolato come hash[1] ottenuto dalla combinazione di tre variabili: IP e porta del nodo, e un numero casuale generato al momento del calcolo. Questo sistema aveva il vantaggio che se un contatto si disconnetteva e se ne riconnetteva un altro con lo stesso ID e porta, allora gli identificativi dei nodi associati ai due contatti sarebbero stati sicuramente diversi[2]. Ogni volta che un nodo comunicava con un altro, veniva inviato anche il valore casuale generato in modo che il destinatario potesse calcolarsi l'ID del mittente. Questo metodo però aveva un grave problema di sicurezza: un nodo malvagio, infatti, poteva ottenere l'ID che desiderava semplicemente scoprendo un valore adatto da sostituire al numero generato casualmente.

Attualmente l'ID debole viene generato a partire dall'IP e dalla porta di un nodo. In questo modo è relativamente difficile per un nodo malvagio ottenere l'ID che desidera, perché dovrebbe cambiare la porta in uso da PariDHT fino ad ottenerlo. Questo algoritmo, però, assegna sempre lo stesso ID ad una certa coppia <IP, porta>: non permette, quindi, di distinguere due contatti diversi ma che condividano lo stesso indirizzo di rete.

[1] Tutti gli hash vengono calcolati con l'algoritmo *SHA-256*[39].

[2] Le probabilità che venga generato lo stesso numero casuale durante la procedura di calcolo dell'ID è praticamente pari a zero

3.3.3 Stato del ID

Per gestire efficacemente ID debole e ID sicuro è stato necessario caratterizzare i nodi conosciuti con uno stato che descrivesse il livello di sicurezza dell'identificativo noto. Gli stati possibili sono i seguenti:

- `SECURE`. L'ID del nodo è stato calcolato attraverso la procedura sicura. Questo stato, una volta acquisito, ha una scadenza pari a mezz'ora oltre la quale il nodo diventa `WEAK`. Solo nodi `SECURE` possono essere inseriti nei k-bucket.
- `WEAK`. L'ID del nodo è stato calcolato con la procedura debole, ma le informazioni necessarie per il calcolo sono state fornite dal nodo stesso.
- `UNKNOWN`. È come lo stato `WEAK` con la differenza che le informazioni necessarie per la generazione dell'ID (nella versione debole) sono state fornite da altri nodi. Quindi, nel caso questi dati si rivelassero falsi, non è da imputare nessuna colpa al nodo di cui si è calcolato l'ID.
- `UNRELIABLE`. Questo stato è riservato ai nodi che sono considerati non affidabili, cioè che hanno mentito sul proprio ID. Anche questo stato una volta assegnato ha una scadenza pari a mezz'ora.

Ora verranno analizzate le azioni da intraprendere sulla base dello stato attuale di un nodo e delle nuove informazioni ricevute.

Nodi `UNRELIABLE`

Qualsiasi comunicazione verso un nodo `UNRELIABLE` fallisce senza spedire alcun pacchetto e allo stesso modo qualsiasi pacchetto ricevuto da un nodo `UNRELIABLE` viene rifiutato. Come già accennato questa condizione è temporanea e alla sua scadenza il nodo viene cancellato dal node storer, quindi il contatto associato al nodo non viene più considerato malvagio, anche se poi potrebbe comunque essere riconfermato `UNRELIABLE`.

Nodi `UKNOWN`

In tabella 3.1 è riportato un riassunto della strategia adottata nel caso si acquisiscano nuove informazioni sull'ID di un nodo di stato `WEAK`. Nella prima colonna è presente il livello di sicurezza del —non necessariamente diverso— nuovo ID di cui si viene a conoscenza. Il nuovo ID sarà di tipo `UNKNOWN` se le informazioni per il suo calcolo sono state fornite da altri nodi, `WEAK` se le informazioni sono fornite dal nodo stesso e `SECURE` se l'ID è stato calcolato con la procedura sicura. Nella seconda colonna è indicato il risultato del confronto tra l'ID che già si conosce e quello appena ricevuto. Infine nell'ultima colonna viene riportata l'azione da intraprendere nel caso specificato.

Si analizzano ora i casi non banali.

ID ricevuto	Confronto	Azione
`UNKNOWN`	uguali	nessuna operazione
	diversi	nuovo nodo `UNKNOWN`
`WEAK`	uguali	il nodo diventa `WEAK`
	diversi	nuovo nodo `WEAK`
`SECURE`	uguali	il nodo diventa `SECURE`
	diversi	nuovo nodo `SECURE`

Tabella 3.1: Comportamento in caso di nodo di stato `UNKNOWN`.

CASO 1: l'ID fornito da altri nodi è diverso da quello che già si conosce.
Basandosi sulle informazioni disponibili non c'è modo per decidere quale ID sia corretto, viene quindi considerato giusto l'ultimo ID ricevuto, confidando sul fatto che quest'ultimo sia più aggiornato.

CASO 2: il nodo comunica un ID diverso da quello che già conosciamo.
Molto probabilmente sono stati gli altri nodi a fornire, volutamente o meno, informazioni non corrette, si crea quindi un nuovo nodo di stato `WEAK`.

CASI 3: l'ID sicuro è diverso da quello che già si conosce.
Analogamente al caso 2 si crea un nuovo nodo, ma questa volta `SECURE`.

Nodi `WEAK`

In tabella 3.2 è riportato come viene gestito un nodo `WEAK`. Di seguito si trova, inoltre, un'analisi dei casi non banali.

ID ricevuto	Confronto	Azione
`UNKNOWN`	uguali	nessuna operazione
	diversi	nessuna operazione
`WEAK`	uguali	nessuna operazione
	diversi	nuovo nodo `WEAK`
`SECURE`	uguali	il nodo diventa `SECURE`
	diversi	nuovo nodo `SECURE`

Tabella 3.2: Comportamento in caso di nodo di stato `WEAK`.

CASO 1: l'ID fornito da altri nodi è diverso da quello che già si conosce.
Si considera la nuova informazione come non corretta e quindi si mantiene l'ID che già si conosce. Il motivo di questa scelta è che gli altri nodi potrebbero avere informazioni non aggiornate oppure potrebbero inviare volutamente informazioni false in modo da screditare il nodo.

CASO 2: l'ID fornito dal nodo stesso è diverso da quello che già si conosce
A prima vista potrebbe sembrare un chiaro esempio di nodo che invia informazioni false e quindi da considerare `UNRELIABLE`. Il nodo che si conosce potrebbe però fare riferimento ad un contatto che si è disconnesso e l'ID ricevuto riguardi un nuovo contatto che ha lo stesso indirizzo di rete di quello conosciuto. Considerando inoltre che per entrar a far parte dei k-bucket l'ID del nodo viene verificato e, quindi, un'eventuale nodo malvagio verrebbe scoperto, si preferisce non considerare il nodo come `UNRIEABLE` ma creare un nuovo nodo di stato `WEAK`.

CASO 3: l'ID sicuro è diverso da quello che già si conosce.
Valgono le stesse considerazioni fatte al caso 2, quindi si procede allo stesso modo solo che il nuovo nodo sarà di stato `SECURE`. In questa circostanza c'è anche un motivo in più per procedere in questo modo: non si vuole sprecare l'informazione ottenuta che, essendo un ID sicuro, è onerosa da procurare.

Nodi `SECURE`

L'ultimo caso da analizzare, in tabella 3.3, è quello di nodo di stato `SECURE`. Come nei paragrafi precedenti, di seguito sono riportati, con maggior dettaglio, i casi non banali.

ID ricevuto	Confronto	Azione
`UNKNOWN`	uguali	nessuna operazione
	diversi	nessuna operazione
`WEAK`	uguali	nessuna operazione
	diversi	il nodo diventa `UNRELIABLE`
`SECURE`	uguali	nessuna operazione
	diversi	nuovo nodo `SECURE`

Tabella 3.3: Comportamento in caso di nodo di stato `SECURE`.

CASO 1: l'ID fornito da altri nodi è diverso da quello che si conosce.
Ovviamente un ID calcolato con la procedura sicura offre più garanzie, quindi si ignora l'informazione e si mantiene l'ID che già si conosce.

CASO 2: l'ID fornito dal nodo è diverso da quello che si conosce.
Questo è un chiaro segnale che il nodo sta fornendo informazioni false, quindi il suo stato diventa `UNRELIABLE`. Le probabilità che sia un nuovo nodo sono molto basse, se ciò si verificasse il nodo verrebbe considerato erroneamente malvagio per un certo periodo di tempo.

CASO 3: l'ID sicuro è diverso da quello che si conosce.
Come già visto negli altri stati, si interpreta il nuovo ID come un nuovo contatto, viene quindi creato un nuovo nodo di stato `SECURE`.

3.4 Nodi NATted e avvio del node storer

Un nodo è detto *NATted* se si trova mascherato da una *NAT*[3]. I nodi NATted sono un problema perché le NAT non permettono connessioni in ingresso provenienti da host sconosciuti[4]. Questa limitazione svantaggia fortemente i nodi NATted perchè tutte le richieste in arrivo da nodi da cui il nodo NATted non ha recentemente inviato un pacchetto, vengono scartate dalla NAT.

Per informare gli altri nodi se il nodo in cui è in esecuzione PariDHT (d'ora in avanti tale nodo sarà chiamato *myNode*) è NATted, in ogni pacchetto inviato è presente un flag impostato a `true` se il nodo mittente è NATted. All'avvio del plug-in myNode viene considerato NATted, questa condizione verrà confermata o smentita al termine della procedura di avvio. Se verrà stabilito che il nodo non è NATted allora verrà calcolato il suo ID e da quel momento in poi il node storer diventerà completamente operativo. Finché myNode è NATted non verrà inserito alcun nodo nei k-bucket. Per maggiori dettagli sulla gestione dei nodi NATted in PariDHT, si veda la tesi di Diego Lazzaro[42].

3.5 Prelevare i nodi vicini

Prelevare dai k-bucket gli n nodi conosciuti più vicini ad un certo identificativo id è il primo passo dell'algoritmo di ricerca dei responsabili di una risorsa (vedi 2.4.3). Questa operazione si può svolgere molto facilmente e efficacemente data la struttura dei k-bucket; basterà infatti calcolare il bucket in cui andrebbe inserito un nodo di identificativo id, inserire tutti i nodi contenuti in quel bucket, poi, se i nodi non risultassero ancora sufficienti, si procederebbe ad inserire i nodi contenuti nei bucket vicini fino a raggiungere n nodi oppure finché non terminano i bucket. Siccome è richiesto che i nodi siano ordinati in modo crescente rispetto alla distanza del nodo da id, si deve utilizzare una struttura ordinata come ad esempio una lista ordinata.

È importante far notare che non viene eseguito nessun controllo sui nodi quando vengono prelevati dai k-bucket perché, se un nodo è in un bucket, significa che è da considerarsi valido, cioè ancora connesso e non malvagio. Ovviamente sarà compito del node storer mantenere nodi validi nei k-bucket.

3.6 Inserimento di un nodo nei k-bucket

Prima di esporre la procedura di inserimento verrà illustrato quali sono le azioni che portano un nodo ad essere candidato a diventare parte dei k-bucket[5]. Si

[3] *Network Address Translation*[41], è un meccanismo che permette a più utenti di connettersi alla rete tramite un solo indirizzo IP.

[4] Da questa limitazione sono afflitti anche i nodi protetti da *firewell* non configurati adeguatamente.

[5] La procedura di inserimento di un nodo nei k-bucket non termina sempre con l'inserimento effettivo del nodo.

hanno due casi possibili:

CASO 1: viene terminata con successo una comunicazione con un nodo.
Affinché la comunicazione sia considerata valida deve essere stata avviata da chi andrà ad inserire il nodo e non dal nodo che si vorrebbe inserire. Un nodo malvagio, infatti potrebbe inviare un messaggio che non necessiti di una doppia risposta, come un ping, falsificando la sua identità[6]; in questo modo il destinatario tenterebbe di inserire un nodo associato ad un contatto che in realtà non esiste.

CASO 2: viene creato un nodo necessario.
Con il termine necessario si intende un nodo che, se inserito, occuperebbe un bucket non ancora pieno; il nodo però è stato creato, questo significa che non è detto ci sia stata una comunicazione con il nodo. Prima di procedere all'inserimento viene eseguito un ping al nodo; sarà poi lo strato di rete ad informare il node storer dell'esito della comunicazione.[7].

L'inserimento di un nodo nei k-bucket avviene in due fasi, nella prima viene verificato che tutte le condizioni necessarie per l'inserimento siano rispettate mentre nella seconda il nodo viene effettivamente memorizzato.

Prima fase

Per prima cosa viene calcolata la distanza del nodo da inserire da myNode e quindi grazie a questa informazione viene individuato qual è il bucket di destinazione.

Se il bucket non è pieno la procedura continua, se invece il bucket è pieno allora la procedura termina quasi sempre, cioè ci sarà una probabilità bassa che la procedura continui. Questo meccanismo è stato inserito per fare in modo che ci sia un certo ricambio dei nodi nei k-bucket infatti, se il bucket è pieno, per inserirne un altro sarà necessario rimuoverne uno. Il valore di probabilità dovrà essere scelto in maniera accurata (attualmente si pensa ad un valore intorno ad 1/20) in modo che i nodi all'interno di un bucket non cambino troppo velocemente.

A questo punto viene verificato l'identificativo del nodo da inserire[8]. Se dopo la procedura di verifica il nodo viene dichiarato `UNRELIABLE` l'inserimento termina mentre nel caso venisse creato un nuovo nodo, se l'identificativo è cambiato, verrebbe tentato, da capo, l'inserimento del nuovo nodo.

Seconda fase

Se il bucket di destinazione non è pieno allora il nodo viene inserito in coda: in questo modo in testa del bucket ci sono sempre i nodi che sono stati inseriti da più tempo.

[6] Questo tipo di attacco informatico viene chiamato *spoofing*[43].
[7] Se il nodo risponde al ping si ricade nel caso 1.
[8] Ovviamente se il nodo è già `SECURE` non viene effettuato alcun controllo.

Nel caso il bucket di destinazione fosse pieno allora si dovrebbe prima scegliere quale nodo rimuovere. Supponendo che il bucket contenga n nodi, si immagina di dividerlo in quattro gruppi A, B, C e D, ciascuno di $n/4$ nodi. Il gruppo A contiene gli $n/4$ nodi che sono stati inseriti da più tempo, il gruppo B contiene gli $n/4$ nodi inseriti subito dopo quelli contenuti in A e così via fino al gruppo D che contiene i nodi inseriti più recentemente.

A questo punto viene scelto casualmente un gruppo, in modo che la probabilità di scegliere un gruppo aumenti al diminuire del tempo passato da quando i nodi del gruppo sono stati inseriti nel bucket. Quindi il gruppo con maggior probabilità si essere scelto sarà D, seguiranno poi C, B ed infine, il meno probabile, A. Il nodo da rimuovere sarà selezionato in modo casuale dal gruppo scelto; questa volta, però, ogni nodo avrà la stessa probabilità di essere scelto. Il nodo rimosso verrà memorizzato nella cache e il nodo da inserire aggiunto in coda al bucket. I valori di probabilità assegnati a ciascun gruppo sono ancora in fase di studio, quelli assegnati attualmente sono riportati in tabella 3.4.

Gruppo	**Probabilità**
A	1/10
B	2/10
C	3/10
D	4/10

Tabella 3.4: Probabilità assegnata ai gruppi in cui è diviso un bucket.

3.7 Validità dei nodi

Come già accennato è fondamentale che i nodi inseriti nei k-bucket siano validi, cioè siano ancora connessi e non malvagi. Non rispettare questo requisito porterebbe ad un rallentamento delle operazioni di ricerca e salvataggio di una chiave e nel caso peggiore, a risultati sbagliati o al non effettivo salvataggio di una risorsa.

È chiaro, però, che i nodi non possono essere controllati costantemente ma bisogna scendere ad un compromesso tra affidabilità e prestazioni. Il problema dei nodi malvagi è già stato discusso nel paragrafo 3.3, basterà fare in modo che l'ID sicuro, di un nodo contenuto in un bucket, venga ricontrollato quando lo stato del nodo scade.

Resta quindi da discutere come massimizzare la probabilità che un nodo contenuto nei k-bucket sia connesso, senza però effettuare troppe comunicazioni nella rete.

3.7.1 Tipi di nodi

È statisticamente noto che maggiore è il tempo di connessione di un nodo, maggiore sarà la probabilità di trovarlo connesso in futuro[36]. Sfruttando questa

proprietà e seguendo l'implementazione di Kademlia in *eMule*[3][44], si è deciso assegnare ai nodi un tipo sulla base del *tempo di connessione*, dove con tempo di connessione non si intende il periodo di tempo passato da quando il nodo è connesso alla rete ma il tempo trascorso da quando un certo nodo ha comunicato per la prima volta con myNode.

Tipo	$\mathbf{T_c}$ (ore)	$\mathbf{T_s}$ (ore)
`EXPIRED`	-	0.5
`TYPE_2`	0	1.0
`TYPE_1`	1	1.5
`TYPE_0`	2	2.0

Tabella 3.5: Caratteristiche dei tipi di nodi.

Nella seconda colonna della tabella 3.5 è riportato il tempo minimo di connessione che un nodo deve avere per poter essere promosso al tipo corrispondente mentre nell'ultima colonna è riportato il periodo di tempo che deve passare dall'ultima comunicazioni prima di forzare il controllo del nodo. Se ad esempio per un'ora non ci sono più state comunicazioni con un nodo di tipo `TYPE_2` allora verrà effettuato un ping al nodo per verificare se è ancora connesso.

Se un nodo è `EXPIRED`, cioè da considerarsi disconnesso, ogni tentativo di comunicazione con quel nodo fallirà senza inviare nulla sulla rete; sono ovviamente accettati i pacchetti dal nodo; se ciò si verifica il nodo `EXPIRED` viene cancellato dal node storer e viene creato un nuovo nodo. Infine per un nodo `EXPIRED` il campo T_s (tempo di scadenza) della tabella indica per quanto il nodo deve essere considerato disconnesso, passato tale periodo di tempo il nodo viene rimosso dal node storer.

3.7.2 Verifica di un nodo

Ora che si è stabilito quando verificare un nodo bisogna decidere come comportarsi nel caso un nodo non risponda ad una richiesta. Rimuovere un nodo alla prima comunicazione mancata porterebbe a considerare molti nodi disconnessi anche quando questi in realtà non lo sono: una comunicazione potrebbe, infatti, fallire anche solo per un momentaneo problema alla rete.

Per risolvere questo problema si è aggiunto al nodo un flag che indica se il nodo è stato *segnalato*. Un nodo viene segnalato se non risponde ad una richiesta e, nel momento in cui questo accade, la scadenza del suo tipo viene impostata a due minuti[9]. Trascorsi i due minuti il tipo del nodo verrà degradato[10] e di nuovo la scadenza verrà impostata a due minuti. Una volta passati questi ulteriori due minuti verrà effettuata con un ping la verifica del nodo: se questa fallirà il nodo

[9]Eventuali altre segnalazioni prima della scadenza non verranno considerate.

[10]Se ad esempio era `TYPE_1` diventerà `TYPE_2`.

verrà segnalato, se invece completerà con successo, al nodo verrà assegnato una tipologia coerente al suo tempo di connessione e la nuova scadenza, conforme con il tipo appena assegnato. Questo meccanismo permette ad un nodo di tipo `TYPE_0` di avere 8 minuti per risolvere gli eventuali problemi di rete prima di essere considerato disconnesso.

Il procedimento sopra descritto ha però due eccezioni. La prima riguarda i nodi di tipo `TYPE_1` che hanno un tempo di connessione minore di un'ora. Questi nodi infatti, non avendo un tempo di connessione sufficiente, sono sempre stati `TYPE_1` e alla prima richiesta non risposta verrebbero considerati disconnessi. Per evitare che questo accada, e solo in questo caso, prima di declassare il nodo a `EXPIRED` viene effettuata un'ulteriore verifica; ovviamente se questa dovesse fallire allora il nodo verrà considerato disconnesso. L'altra eccezione riguarda i nodi con cui non c'è mai stata una comunicazione, tali nodi alla prima comunicazione senza risposta vengono considerati disconnessi.

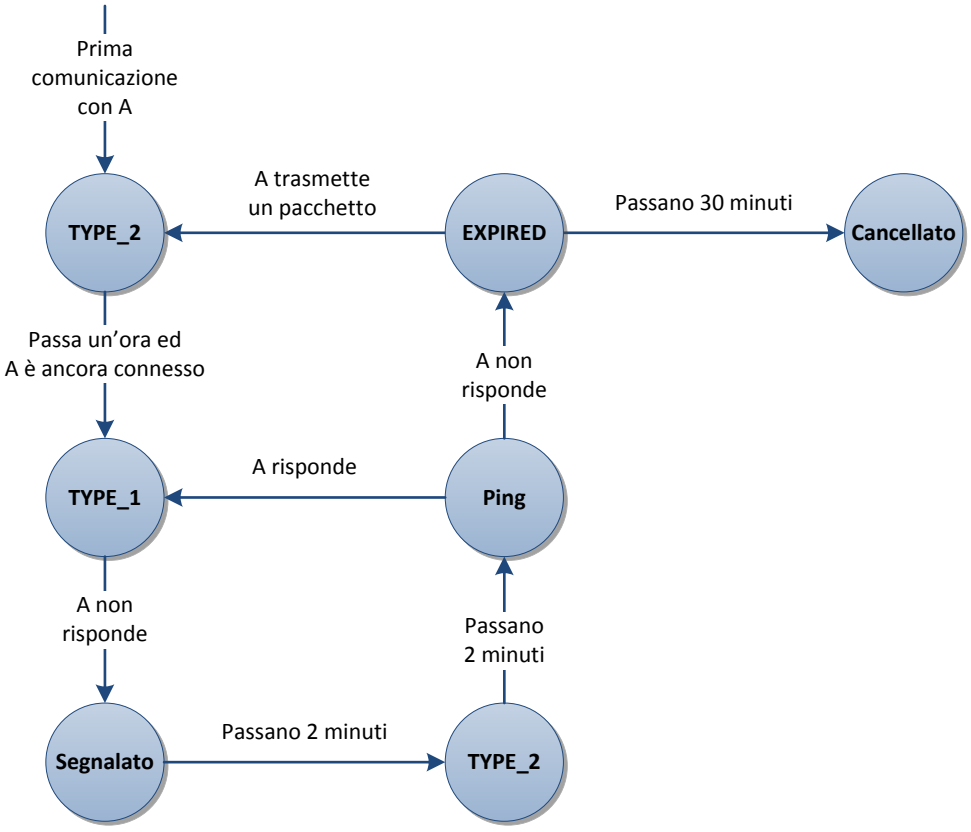

Figura 3.2: Esempio di evoluzione del tipo di un nodo A contenuto nei k-bucket.

Quanto appena descritto, di cui si può vedere un esempio nella figura 3.2, viene applicato anche ai nodi che non fanno parte dei k-bucket. Per tali nodi non viene però avviata nessuna verifica quando il tipo scade ma tutti gli aggiornamenti vengono effettuati solo sulla base delle informazioni fornite dallo strato di rete, cioè si sfruttano le comunicazioni completate o meno, avvenute per altri scopi. Sottoporre tutti i nodi alle verifiche sarebbe stato troppo oneroso in termi-

ni di comunicazioni sulla rete, comunque in questo modo è molto probabile che con il passare del tempo vengano inseriti nei k-bucket nodi di tipo `TYPE_0` che garantiscono maggiori affidabilità e meno verifiche.

3.8 Popolamento e salvataggio dei k-bucket

Anche con i meccanismi appena introdotti non c'è la certezza che un nodo contenuto nei k-bucket sia ancora connesso nel momento in cui viene interrogato. Per questo motivo è importante che, in ciascun bucket, ci sia un numero adeguato di nodi, in modo da permettere a tutte le operazioni di continuare anche se uno di essi si rivelasse disconnesso. Inoltre, come si è visto nel paragrafo 2.4.3, il salvataggio di una risorsa non viene mai effettuato solo su un singolo nodo ma esattamente su `DUPLICATED_REQUEST_NUMBER` nodi.

Per questo motivo ogni ora vengono selezionati tutti i bucket che contengono meno di due nodi e viene effettuata una ricerca di nodi adatti ad essere inseriti in ciascuno dei bucket selezionati. L'ID di partenza per la ricerca viene scelto tra quello dei nodi nel bucket oppure, se il bucket è vuoto, viene generato casualmente.

Per agevolare le future riconnessioni ogni ora viene anche salvata su file una lista di 200 nodi scelti casualmente tra quelli presenti nei k-bucket. Al prossimo avvio del plug-in verrà caricata la lista dei nodi e per ciascuno di essi verrà avviata la procedura di inserimento nei k-bucket.

La frequenza delle due operazioni, il numero minimo di nodi per bucket e il numero di nodi da salvare sono facilmente modificabili dal file di configurazione.

3.9 Validità e dimensione della cache

Affinché il meccanismo di cache funzioni correttamente è necessario che le informazioni che essa fornisce rappresentino il più possibile la situazione reale; ad esempio deve accadere il meno possibile che la cache ritorni un nodo `EXPIRED` quando in realtà il nodo si è riconnesso.

Per questo motivo, come già detto, lo stato `UNRELIABLE` e il tipo `EXPIRED` presentano una scadenza oltre la quale il nodo viene rimosso. Ogni volta che viene richiesto un nodo, prima che venga restituito, viene controllato se lo stato e il tipo sono scaduti: se così fosse il nodo verrebbe cancellato e il richiedente avvisato che non è presente nessun nodo con le caratteristiche richieste.

Un terzo controllo verifica se l'ultima comunicazione è avvenuta da più di mezz'ora, nel qual caso si procederebbe allo stesso modo con la rimozione del nodo.

Bisogna anche fare in modo che non vengano memorizzati troppi nodi nella cache. Per questo motivo dopo aver inserito un nuovo nodo viene verificato se la dimensione della cache è maggiore del massimo consentito, se così fosse verrebbe avviata la procedura di pulizia della cache.

La procedura si articola in due fasi ed ha come obiettivo portare la dimensione della cache al di sotto dell'80% del massimo consentito. Nella prima fase vengono rimossi tutti i nodi che hanno stato `UNRELIABLE` o tipo `EXPIRED` scaduto, oppure con i quali non c'è stata alcuna comunicazione nell'ultima mezz'ora. Se questo non fosse sufficiente ad eliminare almeno il 20% dei nodi, allora si procederebbe con la seconda fase che prevede di eliminare nodi, a partire da quelli con cui non c'è stata una comunicazione da più tempo, finché non si raggiunge un'occupazione minore dell'80% del massimo consentito.

3.10 Riepilogo

Di seguito e in figura 3.3 si trova un riassunto delle informazioni associate ad un nodo che il node storer deve gestire.

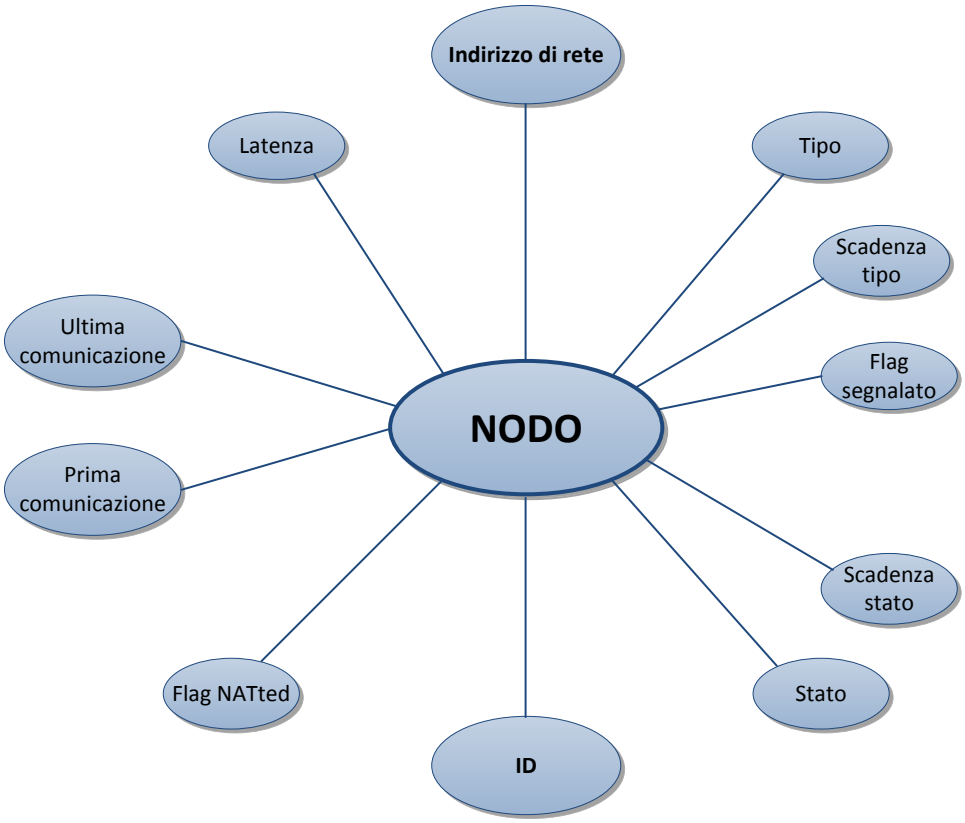

Figura 3.3: Caratteristiche che descrivono un nodo.

Indirizzo di rete. Composto da IP e porta specifica l'indirizzo di rete del contatto a cui il nodo si riferisce.

Flag NATted. Indica se il nodo è NATted.

ID. È l'identificativo univoco del nodo. Viene assegnato solo se il nodo non è NATted.

Tipo. Specifica, sulla base del tempo trascorso dalla prima comunicazione con il nodo, il suo tipo.

Scadenza tipo. Specifica quando dovrà essere verificato se il nodo è ancora connesso.

Flag segnalato. Indica se il nodo negli ultimi due minuti non ha risposto ad una comunicazione.

Stato. Indica il livello di affidabilità dell'ID del nodo.

Scadenza stato. Valido solo per gli stati `UNRIELIABLE` e `SECURE`, nel primo caso specifica quando il nodo non verrà più considerato malvagio mentre nel secondo quando l'ID dovrà essere ricontrollato.

Prima comunicazione. Indica quando è stata effettuata la prima comunicazione con il nodo.

Ultima comunicazione. Indica quando è stata completata con successo l'ultima comunicazione.

Latenza. Specifica la latenza media con cui il nodo risponde alle richieste.

Capitolo 4

Implementazione

Tutte le classi necessarie per realizzare la gestione dei contatti come descritta nel capitolo precedente, sono state inserite nei package `node`, `node.interfaces` e `node.exception`. Le classi principali sono sette: `Node`, `NodeStorer`, `Bucket`, `NodeCache`, `NSTask`, `PeriodicNSFill` e `NodeStorerSaver`.

In tutte le classi la sincronizzazione è gestita attraverso l'uso di metodi `synchronized`.

4.1 La classe `Node`

Come suggerito dal nome la classe `Node` implementa l'entità finora chiamata nodo. Questa classe contiene tutte le informazioni relative ad un contatto, fornisce dei metodi pubblici per accedere ai dati e dei metodi `protected`[1] che permettano al node storer di aggiornare le variabili.

`Node` estende la classe `DHTNode` ed implementa l'interfaccia `INode`. `DHTNode` è la visione che hanno i plug-in di un nodo, PariDHT escluso, quindi offre funzionalità minime: memorizza infatti solo l'indirizzo di rete, l'ID, il flag NATted e la latenza media. Tutte le classi di PariDHT che non fanno parte del package `node`, invece, utilizzeranno solo oggetti di tipo `INode`, in modo da permettere la sola lettura dei dati attraverso i metodi `get*()` e `is*()` presenti nell'interfaccia.

Tutti i costruttori della classe sono `protected`, in modo che una classe esterna al package non possa creare oggetti di tipo `Node` ma debba sempre richiederli al `NodeStorer`; sarà poi quest'ultimo che, se necessario, li creerà. La creazione di un oggetto `Node` avviene fornendo l'indirizzo di rete e l'ID, se il nodo non è NATted, in caso contrario è sufficiente l'indirizzo di rete. In ogni caso, una volta creato un `Node`, non è più possibile cambiare il suo ID, IP e porta, in accordo con il fatto che un eventuale cambiamento di una qualsiasi di queste tre variabili porterebbe ad un nuovo contatto, rendendo quindi le informazioni già conosciute non più valide. Con questi accorgimenti si vuole quindi obbligare il programmatore a creare un nuovo nodo ogni volta che IP, porta o ID cambiano.

[1] Il modificatore di accesso `protected` permette l'accesso solo a classi delle stesso package e alle sottoclassi[45].

La classe Node ha una variabile per ciascuna voce presente nel paragrafo 3.10; i metodi principali sono:

`protected void warn();`

Segnala che il nodo non ha risposto ad una richiesta. Se invocato su un nodo con cui non è mai stata instaurata una comunicazione allora il suo tipo viene impostato a EXPIRED, calcolata la sua scadenza e il metodo termina. Se il nodo non è già stato segnalato allora il flag warn[2] viene impostato a true e il tempo di scadenza del tipo a due minuti. Infine se il nodo era già stato segnalato e il suo tipo è scaduto allora viene degradato invocando il metodo degradeType.

`protected void elevateType();`

Se il tempo di connessione è sufficiente aumenta di un livello il tipo del nodo; ad esempio se il nodo è TYPE_2 ed è conosciuto da più di un'ora diventa TYPE_1. Se finora non era ancora mai stata completata una comunicazione con il nodo, questo metodo gli assegna il tipo TYPE_2 e imposta la variabile che memorizza l'istante della prima comunicazione, altrimenti imposta a false il flag warn. In entrambi i casi viene anche calcolato il nuovo tempo di scadenza del tipo del nodo. Questo metodo viene invocato dalla classe NodeStorer quando, su indicazione dello strato di rete, un nodo risponde ad una comunicazione.

`private void degradeType();`

Abbassa di un livello il tipo di un nodo, ad esempio se il nodo è TYPE_2 diventa EXPIRED. Imposta anche il flag warn a false e il tempo di scadenza a due minuti.

`protected void setIDStatus(IDStatus status);`

Imposta lo stato del nodo a status. Se lo stato è SECURE oppure UNRELIABLE viene anche determinata la sua scadenza.

`protected boolean isExpired();`

Restituisce true se il nodo è disconnesso, false altrimenti.

`protected boolean isUnrielable();`

Restituisce false se il nodo è malvagio, false altrimenti.

I metodi isExpired() e isUnrielable() verificano solo se, rispettivamente, il tipo è EXPIRED e lo stato è UNRIELABLE, ma senza verificare se sono già scaduti.

[2]La variabile booleana warn implementa il flag finora chiamato segnalato.

Il motivo di questa scelta è che, per evitare problemi di temporizzazione[3], in questi due casi le scadenze vengono controllate solo quando il nodo viene prelevato dalla cache[4].

Per gestire con più facilità il tipo e lo stato del nodo sono state creati due enum: `NodeType` ed `IDStatus`.

- `NodeType`. Questo enum, che rappresenta i possibili tipi di nodi, ha come valori `EXPIRED`, `TYPE_2`, `TYPE_1` e `TYPE_0`. A ciascuno di essi è associato un tempo minimo di connessione, che indica da quando il nodo deve essere conosciuto, e un tempo di verifica che specifica la frequenza con cui un nodo deve essere verificato[5]. È anche definita una variabile statica che indica il tempo di verifica nel caso il nodo non rispondesse ad una richiesta. Infine sono presenti due metodi: `degrade()` e `elevate(long connectTime)`. Il primo restituisce il tipo immediatamente più basso di `this` mentre il secondo restituisce il tipo più alto possibile che può essere assegnato ad un nodo conosciuto da `connectTime` millisecondi.

- `IDStatus`. Questa enum rappresenta gli stati che un nodo può assumere. I valori possibili sono: `UNRIELABLE`, `UNKNOWN`, `WEAK` e `SECURE`. Per gli stati `UNRIELABLE` e `SECURE` è anche definita una variabile che specifica il periodo di tempo per il quale lo stato è valido.

4.2 La classe `NodeStorer`

La classe `NodeStorer` contiene tutti i riferimenti ad oggetti di tipo `Node`. È divisa in due parti: i k-bucket, implementati dalla classe `Bucket`, e la cache, implementata dalla classe `NodeCache`.

Prima di illustrare il funzionamento della classe NodeStorer, è necessario descrivere brevemente le classi `NodeIndo` e `CommInfo`.

- `NodeInfo`. Questa classe raccoglie tutte le informazioni per la creazione di un nodo. Queste informazioni devono permettere anche di calcolare l'identificativo. Allo stato attuale è sufficiente l'indirizzo di rete e un flag che specifichi se il nodo è NATted.

- `CommInfo`. Un oggetto di questa classe descrive una comunicazione avvenuta con un certo nodo. All'interno si trovano due flag, uno per indicare se il nodo ha risposto alla richiesta e l'altro per specificare se la comunicazione è stata iniziata da myNode. Si trova infine una variabile che riporta il delay con con il nodo ha risposto.

[3] Ad esempio la cache potrebbe restituire un nodo con stato o tipo prossimi a scadere, quindi potrebbe succedere viene eseguito del codice sulla base di informazioni che subito dopo, magari all'interno dello stesso metodo, verrebbero considerate scadute.

[4] Nodi `EXPIRED` o `UNRIELABLE` possono essere presenti solo nella cache.

[5] I valori dei tempi di connessione e di verificano si trovano in tabella 3.5.

La classe `NodeStorer` implementa l'interfaccia `INodeStorer`, attraverso la quale fornisce dei servizi alle classi esterne al package `Node`.

I metodi principali, presenti nell'interfaccia, sono:

```
public void buildMyNode(NodeInfo info);
```

Questo metodo viene utilizzato per creare l'oggetto `Node` che rappresenta myNode. Come parametro vengono passate le informazioni necessarie per creare un nodo; grazie a queste è quindi anche possibile calcolare il proprio ID. Finché non viene invocato questo metodo myNode viene considerato NATted e nessun nodo viene inserito nei nei k-bucket. Questo metodo non può essere invocato più volte a meno che non venga eseguito il metodo `clear()`.

```
public void clear();
```

Questo metodo ha l'effetto di disconnettere dalla DHT, infatti cancella tutti i nodi dai k-bucket e dalla cache, e imposta myNode a NATted. Dopo aver invocato questo metodo è quindi necessario cercare nuovi nodi da inserire nei k-bucket, altrimenti il plug-in non potrà fornire nessun servizio.

```
public INode createNode(NodeInfo info)
  throws UnreliableNodeException, ExpiredNodeException;
```

Con questo metodo è possibile richiedere un oggetto `INode` al `NodeStorer`. Se un nodo con le caratteristiche richieste non è presente né nei k-bucket né nella cache, allora ne viene creato uno nuovo che prima di essere restituito viene inserito nella cache. Se il nuovo nodo ha un ID che corrisponde ad un bucket non pieno allora viene avviato un ping al nodo. In questo modo, se il ping terminerà con successo, verrà tentato l'inserimento del nodo nei k-bucket. Se il nodo è presente nella cache ma è `EXPIRED` allora viene lanciata l'eccezione `ExpiredNodeException`, mentre se è `UNRIELABLE` viene lanciata l'eccezione `UnrielableNodeException`. Con questo accorgimento si rende più facile per le altre classi la gestione il caso di nodi disconnessi o malvagi.

```
public INode notify(INode node, CommInfo commInfo);
```

Questo metodo permette alle classi di rete di fornire alla classe `NodeStorer` tutte le informazioni riguardante una comunicazione con `node`. Questo metodo restituisce un oggetto di tipo `INode` perché, alla luce delle nuove informazioni, `node` potrebbe aver cambiato ID e quindi, non potendolo modificare, è necessario creare un nuovo nodo.

Oltre a questi, in `INodeStorer` si trovano altri quattro metodi che restitui-

scono rispettivamente `myNode`, un istanza di `Bucket`, un'istanza di `NodeCache` e una di `NSThread`.

Nella classe `NodeStorer` si trovano due metodi `protected` fondamentali per il corretto funzionamento dello stesso.

```
protected Node notifyID(Node node, BigInteger ID,
  boolean isSecure);
```

Informa il node storer che è stato calcolato o trovato un nuovo identificativo `ID`, non necessariamente diverso, per `node`. La variabile booleana `isSecure` serve per specificare se l'ID è stato calcolato attraverso la procedura sicura. L'oggetto di tipo `Node` che viene restituito, rappresenta la versione aggiornata di `node` nel caso fosse necessario cambiare il suo ID. Il metodo implementa quanto descritto in 3.3 aggiornando tutte le strutture che lo necessitano; se, ad esempio `node`, alla luce delle nuove informazioni, risultasse essere `UNRIELABLE` allora andrebbe rimosso dai k-bucket e memorizzato nella cache.

```
protected void store(Node newNode);
```

Questo metodo inserisce `newNode`, che deve essere `SECURE`, nei k-bucket. Come si vedrà in 4.5, l'ID del nodo sarà stato verificato precedentemente. Una volta salvato `newNode` nei k-bucket il nodo verrà registrato affinché il suo tipo e stato vengano controllati periodicamente; `newNode` verrà inoltre rimosso dalla cache.

4.3 La classe `Bucket`

La classe `Bucket` implementa l'interfaccia `IBucket`, e memorizza i nodi che sono salvati nei k-bucket. All'interno di questa classe si trova anche l'ID di myNode.

I bucket sono implementati come un array di 256 liste ordinate, inoltre gli inserimenti avvengono sempre in coda. È presenta anche una hash table che permette di trovare velocemente un nodo a partire dal suo indirizzo di rete.

I metodi principali forniti attraverso l'interfaccia sono:

```
public SortedSet<INode> INode
  findNode(BigInteger id, int n) throws EmptyBucketxception;
```

Accetta come parametro un identificativo `id` e un intero `n`, e restituisce un insieme ordinato contenete gli `n` nodi più vicini ad `id` presenti nei k-bucket. L'insieme è ordinato in modo crescente rispetto la distanza da `id`, non è però garantito che si riescano a trovare `n` nodi. L'eccezione `EmptyBucketException` viene lanciata se i k-bucket sono tutti vuoti.

```
public INode findNode(BigInteger id)
  throws EmptyBucketException;
```

Simile al metodo precedente, con la differenza che restituisce il nodo contenuto nei k-bucket più vicino ad `id`. Anche in questo caso viene lanciata l'eccezione `EmptyBucketException` se i bucket sono vuoti.

```
public int getBucket(INode node)
  throws NATtedMyNodeException;
```

Questo metodo restituisce l'indice del bucket di `node` in è inserito, oppure andrebbe inserito. Questo metodo lancia l'eccezione `NATtedMyNodeException` se myNode è NATted e quindi, non avendo un identificativo, non è possibile determinarne la distanza da `node`.

```
public List<INode> getNodes()
  throws EmptyBucketException;
```

Ritorna una lista contenente tutti i nodi che fanno parte dei k-bucket.

Fanno parte dell'interfaccia anche i metodi `size()`, che ritorna il numero di nodi presenti nei k-bucket, e `isEmpty()`, che restituisce `true` se i k-bucket sono tutti vuoti.

Tra i metodi `protected` i più importanti sono:

```
public List<INode> getRandomNode(int n)
  throws EmptyBucketException;
```

Preleva dai k-bucket `n` nodi scelti casualmente. Se nei k-bucket sono presenti meno di `n` nodi allora li restituisce tutti mentre se i k-bucket sono vuoti allora viene lanciata l'eccezione `EmptyBucketException`.

```
protected List<BigInteger> getPartiallyEmptyBucket();
```

Ritorna una lista di ID. Ciascun identificativo è preso da un bucket che non contiene un numero sufficiente di nodi; se un certo bucket risultasse vuoto allora l'ID verrebbe generato casualmente.

```
protected void store(Node newNode);
```

Questo metodo inserisce `newNode` nei k-bucket. Una volta chiamato questo metodo il nodo viene sempre inserito perché tutti i requisiti sono già stati controllati precedentemente. La memorizzazione del nodo avviene seguendo quanto descritto in 3.6.

4.4 La classe NodeChache

La classe `NodeCache` implementa una cache per immagazzinare i nodi conosciuti che però per qualche motivo non possono fare parte dei k-bucket. Per memorizzare i nodi viene usata una hash table che usa come chiave oggetti di tipo `InetSocketAddress` cioè indirizzi di rete, e come valori oggetti di tipo `Node`. I principali metodi sono i seguenti:

`protected void put(Node node);`

Inserisce un nodo nella cache. Eventuali altri nodi con lo stesso indirizzo di rete verranno sovrascritti. Dopo aver inserito `node` controlla che il numero di nodi contenuti non abbia raggiunto il massimo consentito; se questo si verifica viene avviata la pulizia della cache.

`private boolean isInfoExpired(Node node);`

Verifica se le informazioni acquisite sul contatto devono essere considerate scadute. Restituisce `true` nei seguenti tre casi:

1. Il nodo è `EXPIRED` e il tipo è scaduto, cioè potrebbe non essere corretto considerarlo ancora disconnesso.
2. Il nodo è `UNRIELABLE` e lo stato è scaduto, cioè potrebbe non essere corretto considerarlo ancora malvagio.
3. Non si sono svolte comunicazioni con il nodo da più di mezz'ora, quindi `node` potrebbe raccogliere informazioni che riguardano un altro contatto.

In tutti gli altri casi il metodo restituisce `false`.

`protected Node get(InetSocketAddress sa);`

Restituisce l'eventuale nodo, con indirizzo di rete `sa`, contenuto nella cache. Restituisce `null` se non è presente nessun nodo con tale caratteristica o se il metodo `isInfoExpired`, applicato al nodo, ritorna `true`; in questo caso il nodo viene anche rimosso dalla cache.

4.5 La classe NSTask

La classe `NSTask` implementa un thread il cui compito è quello di avviare l'esecuzione di task assegnati. Al suo interno è presente una `DelayQueue`[46], cioè struttura dati che permette di bloccare un thread finché uno degli elementi contenuti nella coda, alla quale è associato un timer, non scade. Alla scadenza di task, questo viene rimosso e `NSTask` ne avvia l'esecuzione. I task vengono gesti-

ti tramite `WorkerManager`[6], in questo modo si ha la garanzia che non vengano eseguiti troppi task, e quindi thread, contemporaneamente.

I task sono di quattro tipi:

`STORE`. Tenta l'inserimento di un nodo nei k-bucket. Prima viene controllato che il nodo abbia tutte la caratteristiche per poter essere inserito, dopo di che il nodo viene inserito nei k-bucket invocando il metodo `store` della classe `NodeStorer`.

`ID_REFRESH`. Questo task calcola l'ID sicuro di un nodo. Viene utilizzato per verificare che i nodi contenuti nei k-bucket siano ancora sicuri.

`TYPE_REFRESH`. Alla scadenza del tipo di un nodo viene avviato un task questo tipo con il compito di verificare se il nodo è ancora connesso.

`CACHE_CLEANNER`. Questo task esegue la pulizia della cache seguendo quanto descritto in 3.9.

Nella `DelayQueue` si avranno quindi un task di tipo `ID_REFRESH` e `TYPE_REFRESH` per ciascun nodo contenuto nei k-bucket il cui timer corrisponde alla scadenza dell'ID per il primo e la scadenza del tipo per il secondo.

I task `STORE` e `CACHE_CLEANNER` vengono inseriti con scadenza immediata, in questo modo, appena si libera un thread del `WorkerManager` vengono eseguiti. Nella classe sono presenti dei metodi `protected`, generalmente invocati dalla classe `NodeStorer` per permettere l'inserimento dei task.

4.6 Thread periodici

Oltre a `NSTask`, sono presenti altri due thread, che però in questo caso sono periodici: `PeriodicNSFill` e `NodeStorerSaver`.

Il primo ha il compito di cercare nuovi nodi da inserire nei bucket che non sono sufficientemente pieni. L'implementazione prevede che venga invocato il metodo `getPartiallyEmptyBucket()` della classe `Bucket` che restituisce una lista di ID, uno per ogni bucket da riempire. Poi, per ciascun ID verrà eseguita una `LookUp`[7] ed eventuali nuovi nodi saranno, se possibile, inseriti.

`NodeStorerSaver` preleva in modo casuale 200 nodi dai k-bucket, invocando il metodo `getRandomNode` della classe `Bucket`, e li salva su file. Questo file verrà carico al prossimo avvio di PariDHT, ciascun nodo verrà verificato e se connesso inserito nei k-bucket.

Entrambi i thread vengono avviati a distanza di un'ora dall'ultima esecuzione o dall'avvio di PariDHT.

[6] `WorkerManager` è una classe implementata in PariDHT per la gestione dei thread. Per maggiori dettagli sulla gestione dei thread in PariDHT si veda la tesi di Nicola Gobbo[47]

[7] La classe `LookUp` implementa la procedura chiamata appunto, look up, descritta in 2.4.3.

Capitolo 5

Conclusioni e sviluppi futuri

In questo elaborato si è discusso della gestione dei contatti in PariDHT, analizzando problemi come il calcolo dell'identificativo di un nodo, l'organizzazione dei nodi in bucket, la ricerca di nuovi nodi e il controllo dello stato, inteso come connesso o meno, di un nodo.

In futuro il problema principale su cui bisogna concentrare l'attenzione è sicuramente il calcolo dell'ID. L'infrastruttura descritta e implementata prevede l'utilizzo di due procedure una sicura, più lenta, e una debole, più veloce. Soprattutto per quanto riguarda la procedura sicura non è ancora chiaro come debba essere realizzata, in futuro sarà però possibile aggiungerla senza dover stravolgere l'attuale implementazione. La gestione dei contatti, infatti, è fin da subito stata realizzata considerando le procedure di calcolo dell'identificativo nel modo più generale possibile.

Un altro aspetto che necessità di ulteriore attenzione è stabilire in modo più preciso i valori delle variabili che caratterizzano la gestione. È necessario ad esempio fissare la probabilità con cui continua l'inserimento di un nodo nel caso il bucket sia pieno e la probabilità con cui un nodo può essere rimosso da un bucket. C'è anche bisogno di stabilire con accuratezza le durate degli stati temporizzati come ad esempio lo stato `SECURE` o il tipo `EXPIRED`. Tutti i valori attuali, infatti, sono basati su ragionamenti qualitativi oppure ricavati da altri progetti, quindi potrebbero non essere adatti a PariDHT. In ogni caso, questi valori potranno sempre essere aggiustati con il tempo, basandosi anche sul numero di utenti che utilizzano la rete.

Un'idea molto interessante che aggirerebbe il problema del calcola dell'ID, che permetterebbe cioè una certa sicurezza nel salvataggio e ricerca delle risorse anche senza avere una procedura sicura per il calcolo, è stata proposta dal prof. Enoch Peserico. L'idea di fondo è di rendere l'ID di una risorsa variabile nel tempo, quindi a seconda dell'istante temporale in cui viene fatto il salvataggio, la risorsa finirebbe su nodi diversi. Ovviamente l'ID non può cambiare con una frequenza troppo elevata altrimenti una volta completato il salvataggio, la risorsa avrebbe già cambiato identificativo. Supponendo però di calcolare gli ID in modo che cambino ogni mezz'ora, un eventuale nodo malvagio potrebbe riuscire anche, con un po' di sforzo, a posizionarsi dove desidera nella rete ma i danni che pro-

durrebbe sulla risorsa che vuole sabotare sarebbe limitati nel tempo. I problemi principali di questo meccanismo sono che ogni mezz'ora tutte le risorse devono essere spostate in un altro nodo, generando un traffico non indifferente sulla rete, inoltre bisogna prevedere un sistema che permetta ai nodi di sincronizzarsi per il calcolo. Quest'idea è attualmente in fase di studio da Federica Bogo e Michele Bonazza.

Bibliografia

[1] **Wikipedia**
http://en.wikipedia.org/wiki/Peer-to-peer

[2] **BitTorrent**
http://www.bittorrent.com/intl/it/

[3] **eMule**
http://www.emule-project.net/

[4] **Sype**
http://www.skype.com/

[5] **Tor**
http://www.torproject.org/

[6] **Wiki PariPari**
http://www.paripari.it/mediawiki/index.php/Main_Page

[7] **Wikipedia**
http://it.wikipedia.org/wiki/Tabella_di_hash_distribuita

[8] **Petar Maymounkov**, **David Mazières**. Kademlia: A Peer-to-peer Information System Based on the XOR Metric.

[9] **Wiki PariDHT**
http://www.paripari.it/mediawiki/index.php/DHT_en

[10] **Wiki Core**
http://www.paripari.it/mediawiki/index.php/Core_en

[11] **Wiki Connectivity**
http://www.paripari.it/mediawiki/index.php/Connectivity_en

[12] **Wiki Local Storage**
http://www.paripari.it/mediawiki/index.php/Local_Storage_en

[13] **Wiki Crediti**
http://www.paripari.it/mediawiki/index.php/Econ_en

[14] **Wiki Torrent**
http://www.paripari.it/mediawiki/index.php/Torrent

[15] **Wiki Mulo**
http://www.paripari.it/mediawiki/index.php/Mulo_en

[16] **Wikipedia**
http://en.wikipedia.org/wiki/EDonkey_network

[17] **Wiki VoIP**
http://www.paripari.it/mediawiki/index.php/VoIP_en

[18] **Wiki IRC**
http://www.paripari.it/mediawiki/index.php/IRC

[19] **Wiki IM**
http://www.paripari.it/mediawiki/index.php/IM

[20] **Wiki GUI**
http://www.paripari.it/mediawiki/index.php/GUI_en

[21] **Wiki Distributed Storage**
http://www.paripari.it/mediawiki/index.php/Distributed_
Storage_en

[22] **Wiki Database**
http://www.paripari.it/mediawiki/index.php/Database_en

[23] **Wiki NTP**
http://www.paripari.it/mediawiki/index.php/NTP

[24] **Wiki Web Server**
http://www.paripari.it/mediawiki/index.php/NTP

[25] **Wikipedia**
http://en.wikipedia.org/wiki/Extreme_Programming

[26] **Java**
http://www.java.com/it/

[27] **Java Web Start**
http://www.oracle.com/technetwork/java/javase/tech/
index-jsp-136112.html

[28] **Wikipedia**
http://it.wikipedia.org/wiki/C%2B%2B

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

[30] **Wikipedia**
http://en.wikipedia.org/wiki/Gnutella

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

[32] **Wikipedia**
http://it.wikipedia.org/wiki/Hash_table

[33] **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.

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

[35] **Nicola Celli**. PariDHT: accelerazione. Tesi di Laurea triennale, Università degli Studi di Padova, 2010.

[36] **Daniel Stutzbach**, **Reza Rejaie**. Understanding Churn in Peer-to-Peer Networks Section 5.5 Uptime Predictability, Internet Measurement Conference, Rio de Janeiro, October, 2006.

[37] **Oracle**
http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[38] **Wikipedia**
http://en.wikipedia.org/wiki/IPv4

[39] **Wikipedia**
http://en.wikipedia.org/wiki/SHA-2

[40] **PariDHT Wiki**
http://www.paripari.it/mediawiki/images/5/5e/Admission_control.pdf

[41] **Wikipedia**
http://it.wikipedia.org/wiki/Network_address_translation

[42] **Diego Lazzaro**. PariDHT: integrazione con ConnectivityNIO. Tesi di Laurea triennale, Università degli Studi di Padova, 2011.

[43] **Wikipeda**
http://it.wikipedia.org/wiki/Spoofing

[44] **David Mysicka**. Reverse Engineering of eMule. An Analysis of the Implementation of Kademlia in eMule. Semester Thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2006.

[45] **Oracle**
`http://download.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

[46] **Oracle**
`http://download.oracle.com/javase/1,5.0/docs/api/java/util/concurrent/DelayQueue.html`

[47] **Nicola Gobbo**. PariDHT: sincronizzazione. Tesi di Laurea triennale, Università degli Studi di Padova, 2010.

Elenco delle figure

Elenco delle tabelle