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

PariDHT: integrazione con ConnectivityNIO

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

CORRELATORE: Dott. Paolo Bertasi

LAUREANDO: *Diego Lazzaro*

ANNO ACCADEMICO 2010-2011

Alla mia famiglia
per il loro sostegno e supporto.

Indice

Sommario

In questi ultimi anni con l'elevata diffusione della banda larga ed un corrispondente aumento nell'utilizzo di reti peer-to-peer o p2p [1], l'applicazione PariPari tenta di inserirsi in tale ambito, fornendo all'utente molteplici servizi in modo facile e sicuro. Tale piattaforma utilizza una struttura modulare e funziona mediante la creazione di una rete p2p serverless, multifunzionale ed anonima.
Il conseguimento/reperimento di risorse salvate nella rete, viene gestito grazie all'utilizzo di una DHT [2] creata dal plugin PariDHT, ma per lo scambio di dati si ha il bisogno inoltre di generare varie connessioni sincrone o asincrone tra i partecipanti della rete e per questo è stato introdotto recentemente in PariPari il plugin ConnectivityNIO, in sostituzione al "vecchio" plugin Connectivity. Nell'elaborato seguente viene trattato l'aggiornamento del plugin PariDHT allo scopo di utilizzare ConnectivityNIO.
Poiché non tutti i client hanno la possibilità di comunicare facilmente nella rete p2p a causa della presenza delle NAT[1], si è cercato di risolvere tale problema con il sottoplugin NAT traversal di ConnectivityNIO, ma per completare la sua realizzazione è stato necessario eseguire alcune modificare al plugin PariDHT, che saranno in seguito illustrate.

[1]NAT: Meccanismo sviluppato per ridurre il fabbisogno globale di indirizzi IP unici, permettendo a più utenti di connettersi alla rete tramite un solo indirizzo IP.

Introduzione

PariPari è un applicazione in via di sviluppo presso l'Università di Padova da circa 6 anni a cui lavorano attualmente circa 60 persone, tra studenti e laureandi, divisi in vari gruppi. Questa applicazione attraverso una rete distribuita di tipo peer-to-peer offre i più conosciuti servizi disponibili in internet in modo distribuito(Mail, DNS, web server, file hosting, chat) oltre ai comuni servizi forniti dalle reti p2p(file sharing, VoIP, calcolo distribuito) e tutti caricabili in base alle necessità dell'utente.
La rete p2p di PariPari è completamente decentralizzata e basa il suo funzionamento su una tabella di hash distribuita(DHT), al fine di mantenere i contatti dei nodi nella rete. Ogni nodo è a conoscenza solo di un numero limitato di nodi, per condividere/ricercare in breve tempo, risorse/servizi all'interno della rete.
Per fare ciò, si creano diverse connessioni tra i peer, gestite in passato attraverso l'utilizzo della libreria java.io, ma recentemente in PariPari, grazie all'inserimento del nuovo plugin ConnectivityNIO, è stata adottata la libreria java.nio. Tale novità consente di trasferire dati in modo sincrono o asincrono con il protocollo TCP o UDP, migliorando la gestione delle connessioni di rete. Questa introduzione implica che tutti i moduli che vogliono interfacciarsi con la rete P2P, dovranno utilizzare ConnectivityNIO, l'aggiornamento del modulo PariDHT verrà discusso in questo elaborato.
Particolare attenzione in una rete, va data ai nodi mascherati da NAT, i quali sarebbero irraggiungibili dai peer all'interno di PariPari, se non trattati adeguatamente. Tali nodi sono contattabili solo se si imposta correttamente l'indirizzo di destinazione nel pacchetto e vengono considerati dai nodi "reali" non mascherati, come nodi di sola consultazione.
Per gestire i nodi mascherati da NAT, parte del lavoro è già stato fatto in ConnectivityNIO, però per completare l'implementazione è necessario eseguire delle modifiche in PariDHT che verranno esposte in questo elaborato.

Capitolo 1

PariPari

Il progetto PariPari può essere definito come una piattaforma serverless di tipo DHT, multifunzionale, che garantisce l'anonimato dei nodi ed implementa un sistema di crediti.
Attualmente se si desidera utilizzare più applicazioni p2p contemporaneamente è molto probabile che vadano in contrasto tra loro (per esempio nell'uso della banda) e che quindi si abbiano prestazioni molto ridotte con ognuna, PariPari invece dà la possibilità di caricare solo i servizi che l'utente desidera e cerca di soddisfarli tutti in modo equo senza avere eccessivi cali di prestazioni, inoltre in futuro è previsto un plugin che permetta di scaricare contemporaneamente un file sia da torrent sia da mulo(per chiarimenti sui plugin vedi sezione 1.2), con un grosso vantaggio nella velocità di dowload.
PariPari non deve essere visto solo per il file sharing o messaggistica o VoIP, perché può essere adoperato anche per l'avvio di software di calcolo distribuito, così da sfruttare i cicli macchina non utilizzati di ogni peer, molti diffusi nei

Figura 1.1: Logo di PariPari.

computer domestici.

1.1 Caratteristiche Principali

Le caratteristiche principali di PariPari sono:

- **rete serverless**: basata su una particolare varianza di Kademlia [3], questa caratteristica rende la rete meno vulnerabile da attacchi informatici, come ad esempio il Denial of Service (DoS) [4], tipico attacco per rendere il server non più in grado di erogare il servizio al client. La rete PariPari quindi risulterà più robusta e sicura rispetto ai sistemi client-server;

- **architettura multifunzionale ed espandibile**: l'architettura di PariPari offre diversi servizi rendendoli accessibili anche dall'esterno della rete e si basa su una struttura modulare, che permette di semplificare lo sviluppo e rende il software facilmente espandibile, ad esempio con ulteriori moduli creati da privati. Tale struttura dà una maggiore sicurezza in quanto è sempre possibile eliminare i moduli privati ritenuti ostili, attraverso l'aggiornamento di PariPari;

- **anonimato dei nodi**: garantisce l'anonimato dei nodi e di conseguenza il rispetto della privacy, di chi mette a disposizione servizi o risorse e di chi ne usufruisce, proteggendo i dati sensibili di persone ed aziende;

- **sistema di crediti**: un sistema efficiente ed efficace che migliora le prestazioni generali di tutta la piattaforma, strutturato in modo tale che ogni plugin "paga" per avere un servizio e viene "pagato" qualora lo eroga.

PariPari avrà l'ulteriore vantaggio di un elevata scalabilità una volta che il numero di peer(nodi) connessi alla rete supererà una soglia minima, in questo caso l'uscita o l'entrata di un nodo della rete non influenzerà il comportamento e le prestazioni generali del sistema. Da questo ne trae beneficio anche la sicurezza, infatti i nodi malevoli possono essere rimossi dalla rete senza particolari conseguenze.

1.2 Plugin

Come già accennato, per semplificare lo sviluppo e l'organizzazione PariPari possiede un architettura modulare, cioè composta da vari moduli(o plugin). Pur essendo ognuno una struttura a sè stante, sono in grado di cooperare tra loro per mezzo dello scambio di messaggi.
Tra i vari moduli vi è una gerarchia, il gestore generale di tutta l'applicazione nonchè il plugin principale di PariPari è il Core, il quale si occupa di avviare l'intera piattaforma e di reperire e caricare i moduli richiesti dall'utente, gestendo internamente le richieste e le comunicazioni fra i vari plugin.
Tutti i plugin [5] vengono divisi in 2 gruppi, la cerchia interna, composta da tutti quei moduli necessari per la corretta inizializzazione e funzionamento di PariPari, la cerchia esterna, composta da tutti quei moduli che offrono servizi all'utente.
La cerchia interna è costituita dal Core e dai seguenti 4 plugin:

- **ConnectivityNIO**: gestisce i socket e le comunicazioni tra i vari nodi di PariPari, amministrando la banda disponibile in ingresso e in uscita e suddividendola in base alle esigenze dei plugin;
- **PariDHT**: definisce la struttura della rete stessa e permette la localizzazione delle risorse. Deriva da Kademlia e utilizza la metrica XOR;
- **Credits**: suddivisi in crediti interni ed esterni, rappresentano la moneta di scambio per qualsiasi transazione tra moduli (crediti interni) e con gli altri nodi della rete (crediti esterni). Ogni servizio ha un costo in funzione della quantità richiesta e del tempo di utilizzo, e un guadagno assegnato ogni qualvolta venga erogato;
- **Local Storage**: si occupa delle operazioni che avvengono su disco ad opera di altri moduli, ad esempio lettura e scrittura di file di configurazione.

I plugin della cerchia esterna, ciascuno dei quali offre un certo servizio sono:

- **Torrent**: l'obiettivo è fornire un client che implementi il protocollo BitTorrent, finalizzato alla distribuzione e condivisione di file nella rete;
- **Mulo**: fornisce le funzionalità di un qualsiasi client per la rete eDonkey2000 permettendo di scaricare e condividere file di ogni tipo e dimensione;

- **VoIP (Voice Over Internet Protocol)**: servizio che permette una conversazione telefonica sfruttando una connessione Internet cioè il protocollo IP;
- **IRC (Internet Relay Chat) e IM (Instant Messaging)**: due client per la messaggistica istantanea (chat) fra due o più utenti;
- **GUI (Graphical User Interface)**: fornisce l'interfaccia grafica;
- **Distributed Storage**: permette il salvataggio di file in modo distribuito e ridondante nella rete così da renderli sempre accessibili da ogni peer;
- **DBMS (Database Management System)**: ha come obiettivo quello di realizzare un DBMS distribuito sfruttando la struttura della rete di PariPari;
- **NTP (Network Time Protocol)**: crea un sistema di sincronizzazione capace di gestire un orario unico ed affidabile per tutta la rete;
- **Web server**: mette a disposizione un WEB server distribuito.

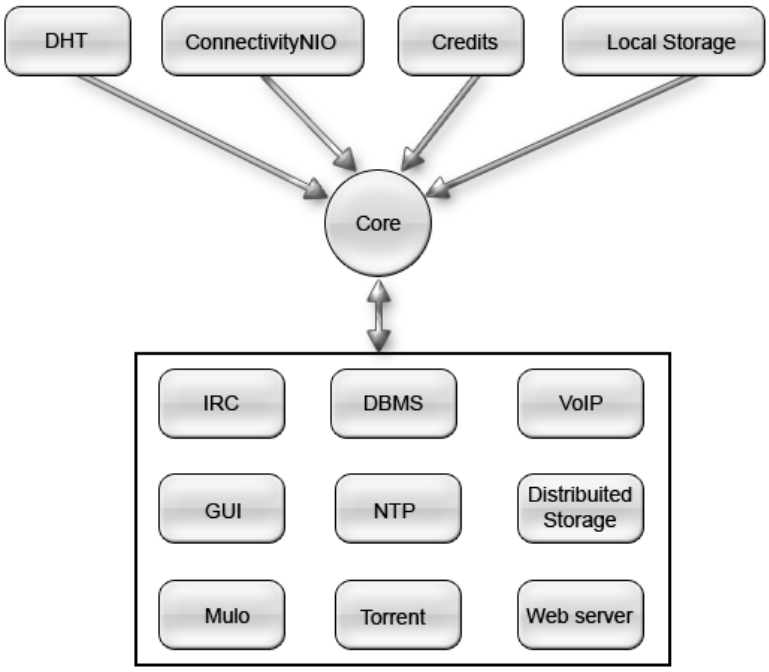

Figura 1.2: Struttura dei plugin di PariPari.

1.3 Java

Per semplificare la creazione e lo sviluppo dei vari plugin, il linguaggio di programmazione adottato è Java, in quanto gli studenti padovani hanno già familiarità con esso, perchè materia di studio nei primi anni universitari.
Le motivazioni cha hanno portato a tale scelta sono diverse:

- principalmente Java è un linguaggio di programmazione ad oggetti molto semplice perciò anche un programmatore con poca esperienza può creare il proprio plugin facilmente;
- PariPari può essere avviato facilmente tramite browser sfruttando la Java Web Start [6] che essendo inoltre compatibile con la maggior parte dei sistemi operativi, garantisce un facile distribuzione dell'applicazione per tutte le architetture. Inoltre permette facili aggiornamenti senza dover ricompilare il codice;
- si possono integrare diverse librerie già sviluppate e testate senza grandi difficoltà, in modo da scrivere poco codice;

L'unico aspetto negativo riguarda la velocità, che si riduce notevolmente in confronto a C++, soprattutto con una grande mole di calcoli da effettuare, come nella crittografia.
Ogni plugin descritto nella sezione 1.2 è un insieme di classi e interfacce java che comunica con gli altri plugin grazie a delle interfacce pubbliche chiamate API [7]. Queste funzionano come delle black box cioè se un plugin vuole utilizzare una funzione di un altro modulo, non ha bisogno di conoscere il reale funzionamento, ma è sufficiente che utilizzi le API e che richiami i metodi pubblici in modo corretto per ottenere quello di cui ha bisogno.

1.4 Gruppi

A capo del progetto PariPari ci sono i Professori Enoch Peserico e Paolo Bertasi e come già accennato nell'introduzione, gli sviluppatori sono studenti e laureandi dell'Università di Padova, divisi in Gruppi.
Tra gli studenti all'interno di un gruppo viene stabilito il capo gruppo, il quale si occupa della coordinazione dei lavori interni e fa da tramite con gli altri capi

gruppi, per discutere su idee future o per eventuali problemi/bug.
A ciascun gruppo è associato lo sviluppo di un particolare plugin e ogni studente svolge sviluppo e test del codice, in generale predilige la regola che nessun sviluppatore testa il proprio codice perchè questo rende molto difficoltoso il riconoscimento di eventuali bug.

Capitolo 2

Proprietà per lo sviluppo e Testing

In questo capitolo saranno discusse le proprietà generali che vengono seguite per lo sviluppo di ogni plugin in PariPari, compreso PariDHT, inoltre verrà approfondita l'attività di testing.

2.1 Proprietà principali per lo sviluppo

Nella realizzazione di un modulo o di eventuali sue parti, si cerca sempre di rispettare le seguenti proprietà:

- **Funzionalità**: l'obiettivo principale nello sviluppo di un modulo consiste nell'ottenere un plugin sempre funzionante, in grado di fornire in modo corretto i propri servizi senza provocare disguidi/problemi agli altri moduli;
- **Facilità di espansione**: nella scrittura del codice è rilevante utilizzare ciò che la programmazione ad oggetti mette a disposizione: interfacce,classi,sottoclassi ed ereditarietà, così da organizzare in modo migliore la struttura del plugin interessato. Nello specifico, è necessario suddividere le classi in base alle loro funzioni in modo da rendere il plugin facilmente espandibile;
- **Documentazione**: la documentazione composta dalla Javadoc [8] e dalla wiki del progetto [9], viene man mano creata con lo sviluppo del modulo stesso ed è indispensabile per facilitarne la comprensione. Inoltre grazie ad una buona documentazione i nuovi studenti che entreranno nel gruppo

riusciranno ad avere una maggiore familiarità con il codice già realizzato, cosicché, la loro integrazione sia la più semplice e veloce possibile;

- **Prestazioni**: nelle fasi iniziali di sviluppo di ogni modulo tale aspetto è stato poco rilevante rispetto alla funzionalità, in quanto viene preso in considerazione solo in periodi successivi. Comunque il suo fine risulta essere l'ottimizzazione dell'intero plugin con un corrispondente aumento delle prestazioni, in questi 2 possibili livelli: livello locale con la diminuzione del numero di risorse utilizzate dal computer e livello distribuito con la riduzione del numero di messaggi scambiati tra i peer e della dimensione dei pacchetti;

- **Testing**: questo aspetto è molto importante e verrà discusso in modo più approfondito nella prossima sezione.

2.2 Testing

L'eXtreme Programming [10] è una metodologia di programmazione agile utilizzata spesso in ambito aziendale per lo sviluppo di applicazioni e per una migliore gestione delle risorse umane così da incrementare la produttività delle stesse. Venne formulata nel 1999 da Kent Beck, Ward Cunningham e Ron Jeffries e adotta come principale procedura per la verifica del codice, la tecnica del testing. Una versione modificata di tale metodologia è stata adottata anche nel progetto PariPari, sfruttando soprattutto la parte riguardante il testing.

Nella scrittura del codice sorgente spesso è semplice rilevare la presenza di errori o mal funzionamenti. Per tale motivo in PariPari viene impiegato il testing come componente basilare nello sviluppo di un plugin, che permette un controllo nella correttezza del codice realizzato, una verifica del risultato ottenuto, una facile individuazione di eventuali bug o azioni non tenute in considerazione dallo sviluppatore, e ottenendo infine un risparmio di tempo e fatica rispetto alle operazioni di debugging [11].
La stesura dei test avviene di pari passo con la creazione del modulo stesso, teoricamente il test deve essere fatto dopo aver realizzato l'interfaccia e prima della scrittura delle classi. Come già accennato nel capitolo 1, vige la regola che nessun

sviluppatore testi il proprio codice.

Per creare un test in PariPari si utilizzano principalmente questi due strumenti software:

JUnit è un Framework[1] open source per effettuare gli unit test o test unitari [12], i quali consentono di esaminare ogni singolo componente di una classe, più in particolare per ciascun metodo viene controllato ogni sua piccola parte di codice. JUnit opera in modo molto semplice, fornisce una vasta gamma di asserzioni(controlli) che permettono di verificare la correttezza e il funzionamento della classe considerata. In questo modo quando viene eseguito un test, è esso stesso che valuta se le asserzioni vengono verificate, e se ciò non succede segnala l'eventuale fallimento del test che corrisponde alla presenza di errori.

Per una classe vengono realizzati diversi test ognuno dei quali deve essere effettuato in modo isolato rispetto agli altri, cioè utilizzando oggetti e variabili completamente nuove non modificate dai test precedenti.

EasyMock è una libreria open source utilizzabile solo quando risulta essere realmente necessaria, infatti un suo uso sconsiderato renderebbe i test difficili da capire e da mantenere. Tale strumento fornisce la possibilità di creare a partire dalle interfacce i mock object [13], oggetti semplici è già collaudati che permettono di entrare più in profondità e di rendere il testing più facile ed affidabile. Tali componenti sono oggetti fittizi, utilizzati per simulare e sostituire il comportamento degli oggetti reali che identificano classi esterne. In questo modo vengono resi invisibili gli errori presenti nelle classi non coinvolte nel test e si isola in modo completo il codice da testare.

Nel test corrispondente di una classe o di un metodo, verrà registrata l'attività dei mock, cioè come tali oggetti dovranno rispondere alle chiamate effettuate sugli oggetti reali simulati. Una volta effettuata tale operazione è possibile avviare il test, che verificherà se il comportamento della classe risulta essere quello atteso, in caso contrario significa che il codice sorgente esegue in modo o in numero errato le chiamate sugli oggetti reali.

[1]Framework: insieme di librerie già create e collaudate, utilizzabili con uno o più linguaggi di programmazione in modo che lo sviluppatore non debba riscrivere codice per compiti simili.

Per effettuare un testing esistono diverse regole da rispettare, come per esempio:

- l'utente deve sempre immettere il messaggio che verrà visualizzato in Console in caso di fallimento del test e quindi tutte le asserzioni devono avere tale messaggio espresso in modo chiaro;

- troppe asserzioni in uno stesso test causano un problema di leggibilità, teoricamente conviene associarne una per ogni test;

- è utile testare i possibili parametri in ingresso con valori scelti intelligentemente dall'utente e non casuali, spesso conviene utilizzare i valori limite;

- catene di oggetti mock troppo lunghe rendono i test difficili da gestire.

Non scendo ulteriormente nei particolari di come realizzare un test perché non è l'argomento principale di questo elaborato.

2.3 Conclusioni

Il testing è stato sviluppato in un capitolo a parte per mettere in rilievo la sua importanza, dato che gode di un'elevata responsabilità nel funzionamento del plugin stesso, essendo la linea guida per lo sviluppatore.

L'aver realizzato molti test per il plugin PariDHT, mi ha permesso di individuare diversi errori e procedere alla correzione degli stessi effettuando anche operazioni di refactoring. Quest'ultima tecnica consiste nel migliorare la struttura interna di ogni classe senza alterare le proprie funzionalità, minimizzando la presenza di bug ed ottimizzando la qualità del codice così da renderlo più comprensibile e gestibile.

Capitolo 3

PariDHT

In questo capitolo verrà introdotto il concetto teorico di DHT [2] e si spiegherà come questo aspetto è stato implementato in PariPari grazie al plugin PariDHT. Inoltre verranno esposte le varie scelte che sono state fatte nello sviluppo di PariDHT, tra le più importanti spiccà l'uso di Kademlia [3] cioè una tabella hash distribuita, decentralizzata, per reti peer-to-peer, progettata da Petar Maymounkov e David Mazières.

3.1 Introduzione

Il sistema tradizionale, utilizzato soprattutto in passato per le reti peer-to-peer, si basa sul concetto di client-server, ma come già accennato(vedi Capitolo 1) questo sistema è poco robusto, dal momento che, se il server si guasta, tutta l'intera rete p2p non può più funzionare finchè il server non viene ristabilito. Inoltre tale sistema è poco scalabile perchè le prestazioni per il trasferimento dei dati sono limitate alle prestazioni hardware del server o alla capacità della linea utilizzata per collegarlo alla rete(banda).
Per questi motivi il modello centralizzato client-server è stato man mano scartato, per passare ad un modello decentralizzato, cioè senza entità superiori che coordinano l'intera rete e il trasferimento di dati in essa.
In questo modo, tutti i nodi sono considerati realmente paritari cioè generici partecipanti alla rete, tutti con la stessa rilevanza, ed ognuno può comportarsi o come client o come server in base alle circostanze.
Inoltre a differenza del client-server, in questo sistema ogni risorsa è distribuita

nell'intera rete p2p, perciò un qualsiasi peer per ricercarla, utilizza particolari algoritmi di instradamento molto più complessi rispetto a quelli utilizzati nel sistema tradizionale centralizzato. Il conseguimento della localizzazione di una risorsa, alla fine coincide con la ricerca del peer a cui essa è associata.
Tra le reti P2P che si basano su un modello decentralizzato esiste la DHT, utilizzata anche in PariPari ed implementata nel plugin PariDHT.

3.2 DHT

Una DHT identifica un particolare sistema di immagazzinamento e reperimento delle informazioni sfruttando una rete di calcolatori.
Ogni partecipante alla rete viene chiamato nodo e gli viene associato un identificatore (ID) unico, che solitamente corrisponde ad un valore di 160bit creati tramite una funzione hash, non reversibile, tipo SHA-1 [14].
Anche le risorse vengono identificate da un ID unico in tutta la rete, facente parte dello stesso dominio dei nodi, questo ID viene ricavato a partire dalla risorsa stessa o da alcuni suoi parametri per mezzo di un hash sicuro. L'utilizzo di un hash sicuro permette un autocertificazione, più precisamente: i nodi client che ricevono una risorsa possono verificare la sua autenticità, confrontando l'hash ricevuto con quello calcolato a partire dalla risorsa stessa. Per il corretto funzionamento di questa tecnica l'hash della risorsa non deve variare nel tempo, ciò comporta che la risorsa in questione deve essere immutabile.
All'interno di una DHT, le risorse che possono essere dati o servizi, vengono memorizzate attraverso le coppie <chiave,valore>, dove la chiave identifica l'ID del dato e il valore coincide con la risorsa stessa. La differenza principale rispetto ad una classica tabella hash, sta nel fatto che, queste coppie <chiave,valore> sono distribuite equamente tra tutti i nodi della rete.
Per determinare la vicinanza tra i nodi e/o risorse, è doveroso fissare una metrica, cioè una funzione matematica che stabilisca come calcolare la distanza tra 2 oggetti(nel nostro caso nodi e/o risorse) facenti parte dell'insieme esaminato(la rete di calcolatori).
Definita la metrica, per rendere la rete operativa è necessario stabilire come i nodi sono in contatto tra di loro e come vengono distribuite le risorse nella rete, in modo equo.

Ogni peer conosce e mantiene in memoria dei contatti, non casuali, più in particolare un limitato numero di ID che corrispondono ai nodi vicini, cioè a distanza minima. Questi ultimi formano l'overlay network [2] e vengono organizzati in modo strutturato dando così forma alla topologia della rete.
Un simile approccio viene utilizzato per la distribuzione delle coppie <chiave,valore>, precisamente ogni risorsa viene associata al gruppo di nodi vicini(cioè con distanza minore possibile), i quali sono considerati i suoi responsabili, cioè manterranno in memoria la coppia <chiave,valore> che la identifica e risponderanno ad eventuali richieste di tale elemento. Per una maggiore disponibilità viene replicata la risorsa tra più nodi responsabili, così da renderla reperibile anche se alcuni di questi non risultano più connessi.
Ultimo ma rilevante concetto di una DHT consiste nell'algoritmo di instradamento, generalmente utilizzato per la localizzazione di risorse e nodi.
Tale procedura si basa sulle seguenti funzioni principali:

- un client che vuole ricercare una risorsa nella rete, invia tale richiesta al nodo, tra quelli a sua disposizione, più vicino alla chiave del dato;
- un nodo che riceve un messaggio di richiesta di una risorsa, può procedere in 2 modi:
    1. se tutti i contatti a sua disposizione sono più distanti dalla risorsa, allora lui è automaticamente il suo responsabile e deve rispondere a tale richiesta con il valore;
    2. altrimenti identifica tra tutti i nodi che conosce il più vicino possibile e gli inoltra il messaggio;
- un nodo che desidera condividere un nuovo servizio o risorsa, gli assegna un ID(chiave) e lo annuncia la rete, la quale, si assicurerà di renderlo disponibile agli altri partecipanti;
- quando un client richiede la rimozione di una risorsa, da lui messa a disposizione, tale richiesta viene soddisfatta dall'algoritmo di instradamento. Non necessariamente ciò deve avvenire subito;
- i nodi possono entrare ed uscire a proprio piacimento dalla rete. Quando un nodo entra gli vengono inviati le chiavi(ID) delle risorse di cui deve essere

responsabile. Quando un nodo esce, la sua responsabilità viene distribuita tra gli altri partecipanti della rete.

Questa è la struttura generale da seguire per l'implementazione di una DHT.

3.3 PariDHT caratteristiche principali

Il plugin PariDHT si incarica di implementare il protocollo DHT in PariPari, basandosi su Kademlia, più in particolare usufruendo di una sua versione modificata.
Il sistema utilizzato associa ad ogni nodo e risorsa, presente in PariPari, un identificatore o ID univoco nella rete, cioè un numero naturale a 256 bit calcolato tramite la funzine hash SHA-256. Non si utilizza il classico ID da 160bit, perchè si avrebbe una probabilità di collisione tra 2 identificatori casuali troppo elevata, rispetto alla versione a 256bit.
La metrica è basata sull'operazione XOR, quindi la distanza d tra 2 elementi x e y, che possono essere gli ID di nodi e/o risorse(dato che giacciono sullo stesso spazio degli indirizzi), viene calcolata con un XOR bit a bit:

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

il risultato è un numero a 256bit intero positivo.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 3.1: Tabella XOR.

Una volta specificato il dominio degli ID e la metrica utilizzata, è utile precisare come i nodi mantengono i contatti degli altri partecipanti e la topologia della rete creata.
PariDHT utilizza una struttura dati costituita da 256 insiemi, composti ognuno da k nodi. Ogni insieme viene detto k-bucket, dove k varia in base alla distanza

dei nodi partecipanti, dal nodo corrente. Più in particolare nell'i-esimo k-bucket ci sono i nodi, la cui distanza d è $2^i \leq d < 2^{i+1}$.
Come già accennato ogni partecipante alla rete conserva e quindi conosce solo gli ID dei suoi vicini, i quali vengono salvati nei vari k-bucket.
La dimensione di ogni k-bucket viene limitata ad un massimo di 20 nodi, questo per mantenere uno spazio occupabile in memoria per ogni utente dell'ordine di O(log N) e teoricamente dovrebbe almeno contenere 5 nodi, per alleviare i problemi introdotti da i churn dei nodi, cioè dalle disconnessioni improvvise dalla rete di tali partecipanti.
I nodi considerati vicini, vengono inseriti nei k-bucket in base al proprio identificatore, la procedura seguita risulta essere: il nodo corrente possiede un ID= $b_{255}, b_{254}, b_{253}, \ldots, b_2, b_1, b_0$,un i-esimo k-bucket intermedio contiene i nodi che:

- hanno lo stesso valore nei bit $b_{255}, b_{254}, \ldots$;
- hanno un valore differente nel bit b_i;

Se conosco più di 20 nodi che andrebbero inseriti in un k-bucket, allora memorizzo solo i 20 nodi la cui distanza dal nodo corrente, sia la minore possibile.

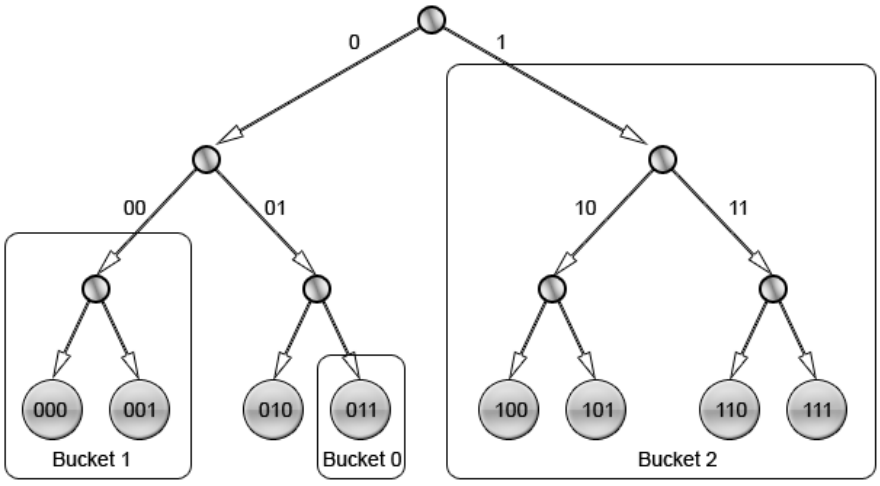

Figura 3.1: Esempio di partizionamento in bucket della rete di PariPari, basata su 3 bit, per il nodo con ID = 010.

Quindi la struttura topologia della rete così creata è a forma di albero, i nodi sono situati nelle foglie e se associamo uno 0 agli archi sinistri e un 1 agli archi

destri, il percorso dalla radice alla foglia, identifica un numero binario che corrisponde all'ID del nodo considerato.
La ricerca di un partecipante nella rete avviene contattando tra i nodi che conosco, quelli più vicini all'ID del peer cercato. I quali risponderanno con un elenco composto dai peer che conoscono più vicini al nodo cercato. In seguito, il nodo di partenza interrogherà tra i nodi ricevuti, quelli a distanza minore dal peer "obiettivo" della ricerca e tale ciclo si ripeterà fino ad arrivare al nodo richiesto.
Ogni hop, data la struttura ad albero, permette di dimezzare la distanza tra il peer iniziale e il peer cercato.
Riassumendo, la ricerca di un nodo nella rete procede dalla radice dell'albero verso il basso e iterativamente per ogni hop si esclude mezzo sotto-albero fino ad arrivare alla foglia cercata, tutto con una complessità computazionale dell'ordine di O(log N).

Capitolo 4

ConnectivityNIO

Il plugin ConnectivityNIO [15] è stato inserito solo recentemente in PariPari, deriva dal "vecchio" plugin Connnectivity [16] e in futuro andrà a sostituirlo completamente, quindi tutti i plugin che utilizzavano Connectivity per la gestione della trasmissione dei dati dovranno fare un aggiornamento a ConnectivityNIO. PariDHT fa parte di questi, infatti, in questo elaborato verrà descritto il processo di aggiornamento eseguito e gli eventuali vantaggi riscontrati.(vedi Capitolo 5)
L'obiettivo generale di entrambi i plugin Connectivity e ConnectivityNIO, non è cambiato: consiste nella gestione delle trasmissioni tra i nodi della rete, creando opportuni socket (derivanti dall'indirizzo IP e dalla porta utilizzata) e amministrando la banda disponibile in ingresso e in uscita per ogni nodo.
Questi plugin inoltre offrono una serie di servizi aggiuntivi come: Anonimato, Multicast, NAT traversal.
Di seguito verrà introdotta la tecnica NAT ed il motivo per cui si utilizza, successivamente verrà descritto in modo generale il plugin ConnectivityNIO e i vari servizi che offre.

4.1 Introduzione NAT

Negli ultimi anni il numero di computer connessi ad Internet, grazie alla diffusione della banda larga, è notevolmente aumentato e gli indirizzi IP da associare ad ogni partecipante della rete sono in via d'esaurimento. Questo problema è dovuto all'utilizzo dell'IPv4 [17], infatti tale versione dell'Internet Protocol crea indirizzi a 32 bit, i quali permettono di identificare univocamente solo circa 4

miliardi di host.
Per risolvere tale problema, la soluzione definitiva consiste nel utilizzate gli indirizzi IPv6 [18], composti da 128bit, i quali consentono di identificare univocamente circa 2^{128} host, numero sufficientemente alto per assegnare un indirizzo IP univoco a tutti i computer della terra e non solo.
Ancora oggi però si utilizza l'IPv4, quindi per risolvere temporaneamente il problema dell'esaurimento degli indirizzi, sono state inventate delle tecniche, una di queste è la NAT (Network Address Translator).
Quest'ultima permette ad una rete privata composta da più computer, di connettersi ad Internet tramite un singolo indirizzo IP pubblico, ed ogni computer che intende comunicare con l'esterno manda i pacchetti alla NAT, la quale sostituisce l'IP privato con l'IP pubblico avendo cura di memorizzare in una tabella la corrispondenza fra i vari indirizzi.
Tuttavia applicando questa tecnica nelle reti P2P, sorge il problema che le NAT non consentono connessioni in ingresso, cioè verso la rete privata, provenienti da host sconosciuti della rete pubblica. Per aggirare tali problemi in PariPari si utilizza il servizio di NAT Traversal, implementato in ConnnectivityNIO e aggiornato tramite le modifiche apportate la plugin PariDHT che verranno descritte in questo elaborato nel capitolo 5.

4.2 Caratteristiche generali

Il plugin ConnnectivityNIO gestisce la banda della trasmissione, risorsa limitata e molto contesa fra i vari plugin, utilizzando l'algoritmo del Token Bucket, che limita il throughput medio di ogni plugin al di sotto di un valore massimo, anche se è sempre possibile una situazione di burst cioè un elevato utilizzo della banda per un breve periodo di tempo, dovuta ad una particolare situazione.
Inoltre il plugin Crediti può influire sulla banda da utilizzare e sul numero di socket che ogni plugin può possedere, più precisamente, questi due servizi hanno un costo e quindi se il plugin interessato non ha i crediti sufficienti per il loro pagamento, di conseguenza non potrà usufruirne.
Grazie a questa struttura, PariPari è migliorata anche nel campo della sicurezza, impedendo a plugin malevoli di utilizzare la banda e i socket in maniera eccessiva ed incontrollata.

I servizi aggiuntivi offerti dal plugin ConnectivityNIO sono:

- **Anonimato**: garantisce l'anonimato del mittente e del destinatario anche in caso d'intercettazione della trasmissione, inoltre nasconde le informazione riguardanti i peer che fanno ricerche o pubblicazioni su DHT. Per implementarlo in PariPari si è preso spunto da TOR [19], cioè un sistema di comunicazione anonima per internet, basato sul protocollo Onion Routing, lo si è adattato aggiungendo tutte le funzionalità necessarie per il suo utilizzo all'interno di una rete P2P;

- **Multicast**: fornisce la possibilità di realizzare delle comunicazioni uno-a-molti cioè un mittente e più destinatari e molti-a-molti cioè più mittenti e più destinatari. Per implementarlo in PariPari sono state usate 2 strategie: SimpleConference utilizzata in caso di comunicazioni pochi-a-pochi, mentre AdvancedConference in caso di comunicazione molti-a-molti, quest'ultima non ancora implementata;

- **NAT traversal**: agevola le comunicazioni all'interno della rete PariPari in caso di nodi mascherati da NAT e/o protetti da firewall. Per questi peer che non riescono a comunicare direttamente e facilmente con la rete, vengono messi a disposizione dei servizi: IP Discover, STUN(Session Traversal Utilities for Network Address Translator) e UDP Hole Punching. L' IP Discover permette l'individuazione del proprio indirizzo IP pubblico, allo scopo di scoprire se si è mascherati o meno da NAT, lo STUN mette a disposizione una tecnica per individuare le porte della NAT/firewall che permettono traffico entrante anche da peer sconosciuti, l'UDP Hole Punching consente lo scambio di messaggi UDP in modo bidirezionale, tra due nodi entrambi mascherati da NAT. Tutti questi servizi per funzionare hanno bisogno di utilizzare un terzo nodo di supporto, chiamato "Server NAT", il quale ha il compito di rispondere alle varie richieste di IP Discover, STUN e coordina i peer durante l'UDP Hole Punching. Un nodo non può scegliere liberamente se fornire i servizi di NAT Traversal cioè se essere "Server NAT", ma deve prima capire se è in grado di poterli erogare, per esempio tutti i nodi non nascosti da NAT e/o firewall possono offrite tali serviz;

4.3 Differenze

4.3.1 Connectivity

Il plugin Connectivity [16] utilizza la libreria java.io basata sul concetto di Stream nel caso di comunicazioni TCP, cioè un flusso di dati tra una sorgente e un destinatario in un canale di comunicazione.
Quando un plugin si interfacciava alla rete tramite Connectivity, l'unica possibilità è di eseguire tutte le azioni in modalità bloccante cioè rimane in attesa senza poter effettuare altre operazioni, sia in caso di input(finchè si attendono i dati in ingresso), sia in caso di output(come la scrittura di dati su un socket).

4.3.2 ConnnectivityNIO

Il nuovo plugin ConnectivityNIO [15] utilizza invece java.nio dove la "n" stà per new ma anche per non bloccante e consiste in una collezione di API che offrono caratteristiche avanzate per le operazioni di input e output, come la possibilità di fare i vari tipi di richieste in modalità non-bloccante.
Questa funzionalità di I/O asincrono, permette l'esecuzione di altre operazioni, mentre sta terminando la fase di lettura o scrittura di un thread, ottenendo il risparmio di molte risorse del sistema, infatti, non ci saranno più thread che rimarranno in attesa di flussi I/O "lenti".

Il funzionamento di ConnectivityNIO si basa sui concetti di Buffer, Channel e Selector:

Buffer rappresenta un contenitore di dati di una certa dimensione, con a disposizione operazioni per inserire e rimuovere informazioni. In ogni nodo sono presenti due buffer separati, rispettivamente per i dati da inviare o da ricevere.

Channel rappresenta un canale, cioè veri e propri "tubi" tramite i quali è possibile trasferire dati sia in input che in output. Il concetto generale di Channel si diversifica rispetto allo Stream di java.io in molti punti, le differenze principali sono le seguenti:

1. è possibile sia leggere che scrivere sul Channel, mentre gli Stream sono a senso unico;
2. Il Channel può essere anche letto e scritto in modalità asincrona;

In java.nio esistono 4 tipi principali di canali:

- **FileChannel**: permette la creazione di un canale adeguato per la lettura e la scrittura di file, ed è l'unico che non dà la possibilità di essere impostato in modalità non-bloccante;
- **DatagramChannel**: permette l'invio e ricezione di dati via UDP;
- **SocketChannel**: permette l'invio e la ricezione di dati via TCP;
- **ServerSocketChannel**: permette di ascoltare la rete per le connessioni TCP entranti, come fa un server web. Per ogni connessione in entrata viene creato un SocketChannel;

Un ulteriore caratteristica importante dei Channel riguarda il fatto di essere "thread-safe", cioè più thread possono eseguire contemporaneamente operazioni di lettura e scrittura senza ottenere risultati anomali.

Selector permette di multiplexare e demultiplexare i Channels e di aspettare/riconoscere possibili eventi che possono sopraggiungere dalla rete. Ad ogni Selector si possono associare uno o più canali, a discrezione del programmatore. Un grande vantaggio è la possibilità di gestire con un singolo thread un elevato numero di Channels, migliorando notevolmente le prestazioni del sistema, poichè avrò un quantità molto minore di thread attivi per la gestione di diversi canali, rispetto al passato.

4.4 Vantaggi

Gli argomenti descritti in questo capitolo rappresentano un quadro generale delle innovazioni portate da ConnectivityNIO implementando la "nuova" libreria java.nio.
Tali rinnovamenti si andranno a riflettere su tutti i plugin che utilizzeranno i servizi offerti da ConnectivityNIO, più in particolare i plugin che avranno un elevato numero di connessioni di rete come Torrent e Mulo. Questi ultimi otterranno

un notevole miglioramento nelle prestazioni(velocità di esecuzione) per le operazioni di I/O, grazie alla riduzione del numero di thread creati e alla possibilità nei socket di operare in modalità non bloccante.
In questo modo si dà la possibilità, a chi possiede computer non recenti, di non avere cali di prestazioni causati dell'esecuzione di PariPari.

Capitolo 5

Implementazione

In questo capitolo verrà descritto il plugin PariDHT ancora più in dettaglio, inizialmente verrà esposto il protocollo DHTPP utilizzato, di seguito verranno spiegate le modifiche che sono state apportate alle classi di tale modulo.

5.1 Il protocollo di PariDHT

Il protocollo utilizzato dal plugin PariDHT è chiamato DHTPP(DHT PariPari Protocol) versione 0.1, gode delle caratteristiche principali già descritte nella sezione 3.3 "PariDHT caratteristiche principali", però è opportuno aggiungere le 6 primitive o funzioni disponibili, già implementate in questa versione:

- **Ping**: funzione molto semplice che permette di controllare se un nodo è ancora connesso o meno alla rete;

- **Find Node**: funzione che permette di trovare i nodi più vicini ad un ID di 256bit, argomento in ingresso per tale primitiva. Colui che riceve tale richiesta, restituisce k triple del tipo <ID,IP,Porta>, ciascuna tripla rappresenta un nodo connesso alla rete. Questi k nodi sono quelli a distanza minima all'ID cercato, tra quelli che il nodo ricevente conosce;

- **Find Resource Owner**: funzione per reperire le risorse dalla rete, molto simile alla precedente, ma differente in un punto. Durante la ricerca di un ID, se un nodo si accorge di essere il responsabile della risorsa cercata, restituisce i nodi che la possiedono al posto di quelli più vicini;

- **Store**: primitiva per inviare ad un nodo la richiesta di salvare una risorsa nella rete, in modo tale da essere reperibile in futuro. Il ricevente di tale richiesta memorizzerà la coppia <chiave,valore> associata alla risorsa e provvederà a renderla disponibile;

- **Pass Resource**: funzione che permette di passare la responsabilità di una risorsa, quindi la corrispondente coppia <chiave, valore>, ad un altro nodo. Come argomento in ingresso richiede un identificativo di 256 bit che rappresenta l'hash della risorsa. Un nodo esegue tale richiesta, quando viene a conoscenza di un qualsiasi altro partecipante della rete molto più vicino alla risorsa stessa;

- **Key note**: primitiva che permette di ottenere le note associate ad una certa chiave.

Per ognuna di queste primitive esiste un adeguato messaggio di richiesta e un altrettanto opportuno messaggio di risposta.
Tutti i messaggi inviati e ricevuti dal plugin PariDHT usano il protocollo UDP, poiché l'utilizzo del protocollo TCP ha un elevato costo, infatti con quest'ultimo solo per stabilire una connessione tra 2 nodi, i byte scambiati sono spesso superiori all'invio del messaggio stesso.
Ogni messaggio scambiato tra i peer, è costituito da un header o intestazione e da un body o corpo del messaggio.

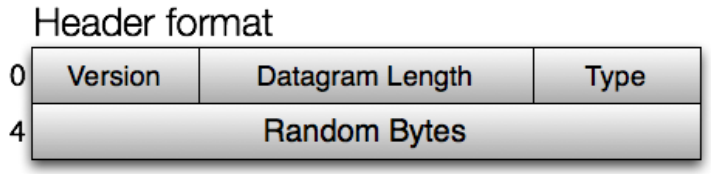

Figura 5.1: Header dei messaggi DHTPP.

Come si vede dalla figura 5.1 l'header di ogni messaggio è composto da 4 campi:

- **Version** (1 byte): rappresenta il numero della versione del protocollo DHTPP utilizzato;

- **Datagram length** (2 byte): corrisponde all'intera lunghezza del messaggio creato da PariDHT, cioè header più body, da sottolineare che questa lunghezza corrisponde con la grandezza del payload del pacchetto UDP, che successivamente verrà inviato nella rete;

- **Type** (1 byte): rappresenta il tipo di messaggio che viene inviato, per le varie possibilità (vedi tabella 5.1). Il byte corrispondente, può avere significati diversi in base ai primi 4 bit, che possono identificare o un messaggio di richiesta se sono tutti a 0 o un messaggio di risposta se sono tutti a 1, e in base ai 4 bit meno significativi che indicano rispettivamente il tipo di primitiva contenuta;

- **Random byte** (4 byte): identifica la sessione per ogni nuovo messaggio che viene inviato nella rete. Questa sequenza di 4 byte viene generata in modo random e viene posta nel messaggio inviante, la corrispondente risposta dovrà avere la stessa sessione.

Primitiva	type di richiesta	type di risposta
Ping	0x01	0x11
Find Node	0x02	0x12
Pass resource		0x16
Find resource owner	0x03	0x13
Store	0x04	0x14
Get note for key	0x05	0x15
Favourite child	0x07	0x17

Tabella 5.1: Tabella dei possibili type per un messaggio DHTPP.

Dopo l'intestazione è presente il body, costituito inizialmente dalla tripletta che identifica il nodo inviante. Tale informazione è costituita principalmente da 3 parti <ID,IP,porta>, con la successiva aggiunta del campo Info.

Come si vede dalla figura 5.2 la tripletta di ogni messaggio è composta da:

- **IP del nodo** (4/16 byte): attualmente si utilizza l'IPv4 (4 byte) però è già stato predisposto dello spazio libero per l'eventuale passaggio all'IPv6 (16 byte);

- **Porta UDP** (2 byte): utilizzata per la comunicazione in ingresso e in uscita con la rete, da parte del plugin PariDHT, di default 5335;

- **ID del nodo** (32 byte): cioè 256bit generati tramite la funzine hash SHA-256;

- **campo Info** (12 byte): lasciato disponibile agli altri plugin per inserire informazioni utili.

Ci sono inoltre 2 byte non utilizzati che possono essere sfruttati in futuro per altri scopi.

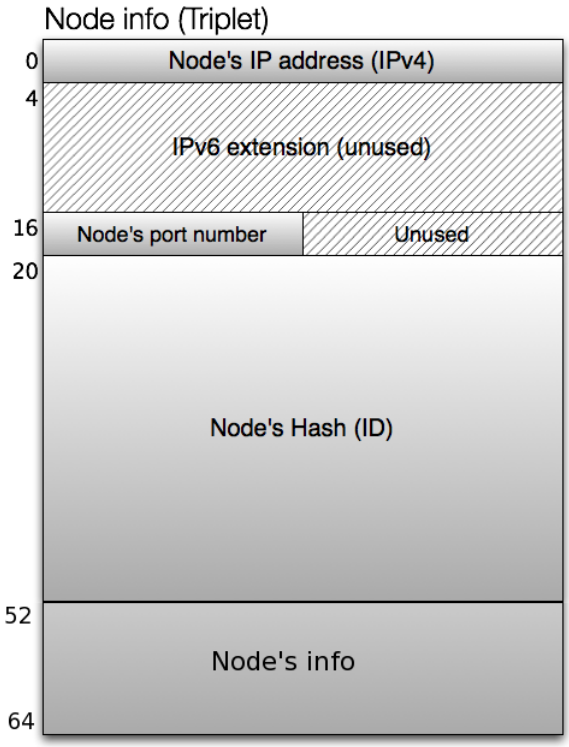

Figura 5.2: Rappresentzione di un nodo in un messaggio DHTPP.

Dopo tali informazioni, nel body è presente il contenuto vero e proprio del messaggio, che generalmente si differenzia in base al tipo di primitiva utilizzata.

La struttura delle classi di PariDHT, può essere divisa in 3 grandi gruppi:

1. **Classi principali e di interfacciamento con il Core**: inizializzano l'avvio dell'intero plugin a partire da un file di configurazione, e gestis-

cono tutte le richieste provenienti e verso gli altri moduli, passando tramite il Core;

2. **Classi di rete**: implementano il protocollo di comunicazione DHTPP e permettono di eseguire tutte le comunicazioni con gli altri nodi della rete;

3. **Classi del Kernel**: implementano gli algoritmi per la ricerca di informazioni nella rete e gestiscono tutti i nodi conosciuti e tutte le risorse di cui si è responsabili.

5.2 Integrazione con ConnectivityNIO

L'aggiornamento del plugin PariDHT all'utilizzo di ConnectivityNIO, aspetto necessario per il funzionamento futuro del plugin stesso, è stato eseguito tramite le seguenti operazioni: inizialmente è stato effettuato il passaggio all'utilizzo dei nuovi socket offerti da ConnectivityNIO, ed eliminando completamente l'impiego del "vecchio" plugin Connectivity.
Successivamente sono state implementate le nuove caratteristiche per il funzionamento dei nodi mascherati da NAT, più in particolare i passi generali seguiti sono: inserimento del StateNAT per ogni nodo connesso alla rete, la modifica di ogni messaggio inviato con l'aggiunta di un bit che identifica lo StateNAT del nodo mittente, i nodi nascosti da NAT vengono considerati come nodi in sola consultazione ed infine è stato modificato il messaggio verso un nodo mascherato da NAT, cambiando la porta UDP in modo adeguato.
Questi adattamenti verranno discussi in modo più approfondito e dettagliato, qui di seguito.

5.2.1 Aggiornamento socket

L'aggiornamento nel plugin PariDHT all'utilizzo dei socket UDP forniti da ConnectivityNIO, è stato implementato principalmente nella classe CoreProxy, il cui compito è quello di fornire l'accesso ai servizi resi disponibili dagli altri moduli come socket e/o file.
Tale classe fa parte dell'insieme delle "Classi principali e di interfacciamento con il Core".
Prima della creazione del socket, ConnectivityNIO a differenza di Connectivity

pretende la registrazione di ogni plugin che richiede eventuali suoi servizi, questo come espediente per limitare la banda utilizzata da ogni modulo. In passato per ogni socket aperto veniva applicato l'algoritmo del Token Bucket, ora invece tale algoritmo viene applicato sia per ogni socket sia per l'intera banda di ciascun plugin che utilizza ConnectivityNIO. Questa novità conferisce ai moduli la possibilità "dell'acquisto", attraverso i crediti posseduti, della quantità di banda che necessitano, indipendentemente da come poi verrà distribuita nei vari socket in loro possesso.

In CoreProxy, una volta stabilita la banda necessaria per l'intero plugin PariDHT, si effettua la registrazione a ConnectivityNIO e successivamente si procedere alla creazione del socket UDP utilizzando la libreria PluginSender.
Quest'ultima permette in modo semplice di utilizzare le API degli altri plugin di PariPari, interfacciandomi con il Core, in questo caso si utilizza tale libreria per richiedere un oggetto di tipo PPUdpNapAPI il quale mi identifica un socket UDP.
La chiamata effettuata è:

```
socket = pSender.requestSingleUDPSocketAPI(bandwidthin,
    bandwidthout, features, socketblockingmode, autorenew);
```

Come si può osservare, è possibile personalizzare la creazione del socket con i seguenti parametri espliciti: la banda in ingresso e in uscita richiesta per tale socket, il numero di crediti che il plugin Credits dovrà validare, l'opzione autorenew che permette di rinnovare le risorse che scadono in termini di tempo in modo automatico tramite PluginSender(l'opzione è abilitata) e per ultimo, l'aspetto più importante, cioè l'utilizzo di tale socket in modalità bloccante o meno.
Al momento PariDHT utilizza un solo socket UDP per la gestione di tutti pacchetti, sia in entrata che in uscita. A tale socket sono associati solo 2 thread, che gestiscono rispettivamente i pacchetti spediti e ricevuti. Visto lo scarso numero di processi necessari per la gestione del socket, è stato ritenuto opportuno e conveniente utilizzarlo in modalità bloccante, dato che non si avrebbero rilevanti miglioramenti in termini di prestazioni nel caso non-bloccante.

Se invece si utilizzasse una comunicazione basata su socket TCP, come per esempio Torrent o Mulo, per mandare informazioni tra 2 nodi sarebbe stato neces-

sario un collegamento diretto fra di loro e inoltre per ogni coppia di nodi connessi, un diverso socket. In questi casi, il numero di socket e thread attivi sarà molto maggiore rispetto a PariDHT, quindi l'utilizzo di socket non-bloccanti mi permetterebbe di ridurre notevolmente il numero di processi in attesa e perciò di risparmiare parecchie risorse del sistema.
Dato che PariDHT non utilizza e non gli conviene utilizzare una connesione TCP,(vedi sezione 5.1 "Il protocollo di PariDHT") molto probabilmente anche per il futuro non porterà nessun profitto l'utilizzo di socket in modalità non-bloccante, per tale ragione è opportuno utilizzare socket bloccanti.
Altre classi di PariDHT che sono state modificate a causa del cambiamento di socket, sono le seguenti (tutte facenti parte delle classi di rete):

- **NetInitializer**: inizializza la rete principale, nello specifico crea il socket tramite un'istanza della classe CoreProxy e gestisce i thread NetSender e NetReceiver;

- **NetReceiver**: thread che rimanere in ascolto sul socket UDP e gestisce tutti i pacchetti ricevuti, in particolare controlla la loro validità. Se non contengono errori, li invia alla classe inRequest che li smista opportunamente in base alla tipologia del messaggio ricevuto per esempio richiesta o risposta;

- **NetSender**: thread che preleva i pacchetti da inviare memorizzati in una coda condivisa e li inoltra ai rispettivi destinatari tramite il socket UDP.

Solo al momento della chiusura dell'applicazione, il socket e i relativi thread verranno chiusi.
Infine anche il file descriptor.xml nel quale sono menzionate le API offerte ed utilizzate, è stato modificato. Più precisamente, i servizi pubblici che PariDHT mette a disposizione non sono cambiati, mentre tra i servizi usufruiti è stato rimpiazzato l'uso del plugin Connectivity dall'plugin ConnectivityNIO.

5.2.2 Gestione dei nodi mascherati da NAT

StateNAT

Per differenziare in PariPari i nodi mascherati da NAT, si è introdotto il concetto di StateNAT, cioè ad ogni nodo connesso alla rete viene associato uno stato, il quale può assumere 3 valori:

- **TRUE**: identifica un nodo mascherato da NAT;
- **MAYBE**: identifica un nodo che non sa se attualmente è mascherato o meno da NAT;
- **FALSE**: identifica un nodo non mascherato da NAT, un nodo "reale".

Lo StateNAT è di tipo enumerativo ed è stato definito nell'interfaccia INode.
La corrispondente classe del Kernel Node che implementa tale interfaccia, rappresenta la struttura dati di un nodo connesso alla rete P2P, e necessita per la creazione di tale oggetto di 4 parametri principali in ingresso al costruttore, il suo IP, ID, la porta UDP e lo StateNAT, impostato di default su MAYBE.

Quando ConnectivityNIO attraverso il servizio di IPDiscover riuscirà a stabilire se il nodo è mascherato da NAT o meno, comunicherà tale informazione a PariDHT che imposterà il reale stato di quel nodo.
All'interno della classe Node sono stati aggiunti i metodi per conoscere lo stato di un nodo e per modificarlo, quest'ultimo verrà utilizzato quando si conoscerà il reale StateNAT.

NAT bit

Dopo l'aggiunta di tale caratteristica, si è ritenuto opportuno precisare in modo veloce ed efficace quando un nodo sia mascherato da NAT o meno, modificando la struttura del pacchetto che viene inviato nella rete.
Tale cambiamento consiste nell'aggiunta di 1 bit, NAT bit, impostato ad 1 quando lo StateNAT del nodo inviante è TRUE o MAYBE, a 0 quando lo StateNAT è FALSE. Tale bit è stato inserito nel body del messaggio, più precisamente nella tripletta <ID,IP,porta>, subito dopo la porta, dove erano presenti 2 byte non

utilizzati, in modo che la tripletta inglobasse tutte le caratteristiche principali del nodo inviante.

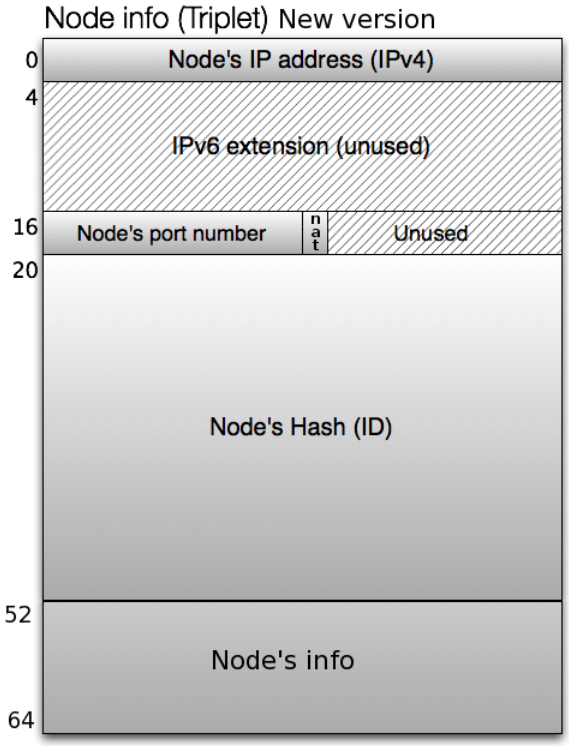

Figura 5.3: Nuova rappresentzione di un nodo in un messaggio DHTPP.

In questo modo, quando si riceve un messaggio da un qualsiasi altro nodo, è possibile individuare fin da subito se è mascherato o meno da NAT, grazie al NAT bit, e perciò comportarsi di conseguenza (vedi prossimo paragrafo).

Per implementare la nuova tripletta sono state apportate delle modifiche alle seguenti classi di rete:

- **DHTPacket**: rappresenta il pacchetto completo scambiato tra i nodi della DHT;

- **PacketFactory**: crea il DHTPacket adeguato da inviare sulla rete a partire dai parametri dati in ingresso al costruttore ed al metodo invocato, inoltre include la struttura dati del nodo inviante, cioè la tripletta e il NAT bit;

- **PacketParser**: si occupa di analizzare i pacchetti ricevuti e di riconoscerne la tipologia, estrapolando dal pacchetto i dati relativi del nodo inviante e creando l'oggetto Node associato.

Nodi in sola consultazione

I nodi con StateNAT uguale a TRUE o MAYBE, non sono considerati come veri e propri partecipanti attivi della DHT, ma vengono reputati come nodi in sola consultazione, cioè tali peer possono eseguire delle normali ricerche o dei salvataggi di risorse (coppie <chiave,valore>), ma non vengono mai considerati dagli altri nodi della rete, come responsabili di risorse o presi in considerazione per le operazioni di ricerca.
Nel caso generale di richiesta di una risorsa, salvata nella rete da parte di un qualsiasi nodo, si procederà con la Find dei suoi responsabili, cioè i nodi reali più vicini alla chiave cercata. Da questo è facile intuire che anche se un nodo nascosto da NAT compie una Store, sarà impossibile rintraccialo, per 2 motivi:

1. non sarà mai considerato fra i responsabili essendo mascherato da NAT;
2. in una rete molto popolata ogni nodo non è mai responsabile delle proprie Store, perché ci sono basse probabilità di vicinanza tra l'ID della risorsa e il nodo originale suo possessore.

Tali discriminazioni sono state introdotte per il motivo che il servizio di NAT traversal messo a disposizione da ConnectivityNIO e utilizzato per contattare i nodi mascherati da NAT è molto costoso in termini di tempo, quindi non vale la pena adoperarlo nelle semplici operazioni di Find e Store effettuate dal plugin PariDHT, perché provocherebbe solamente una riduzione eccessiva delle prestazioni globali di PariPari. Avrebbe più senso impiegarlo solo quando il nodo mascherato offre un certo servizio come server web, scambio di file ecc.
Riassumendo, i nodi con StateNAT uguale a TRUE o MAYBE, possono effettuare dei Find o degli Store, ma non verranno mai considerati come responsabili dei propri salvataggi di risorse, inoltre non saranno mai contattati dai nodi reali per operazioni di Store o eventuali ricerche nella DHT. Questo può essere implementato impedendo ad ogni nodo della rete, di memorizzare i peer mascherati da NAT tra i nodi conosciuti.

La struttura dati in PariDHT, che tiene traccia in modo ordinato dei peer conosciuti all'interno della rete e che viene utilizzata per tutte le operazioni di ricerca di nodi o responsabili di risorse, corrisponde alla classe del Kernel NodeStorer. I nodi vengono inseriti in tale struttura grazie alla classe del Kernel NSInsertionThread.
In quest'ultima è stato aggiunto un controllo, più precisamente viene verificato lo StateNAT: se i nodi sono reali si effettua l'operazione di immissione altrimenti vengono scartati. Ciò per impedire che i nodi mascherati da NAT siano introdotti nella NodeStorer, così da imporre che i risultati delle ricerche o i nodi su cui effettuare salvataggi di risorse, siano solo nodi reali, come si è stabilito inizialmente.
Esistono 2 ulteriori ed importanti classi del Kernel associate alla NodeStorer:

- **NSSaveThread**: salva periodicamente i nodi presenti all'interno della NodeStorer su file;
- **NSFillThread**: riempie la NodeStorer con i nodi già conosciuti, i quali vengono prelevati da file o ricercati direttamente nella rete.

In ragione a quanto detto sinora, i nodi provenienti dalla NodeStorer e recuperati da file, sono considerati come nodi non mascherati da NAT, quindi vengono inseriti nella struttura dati dalla classe NSFillThread, con lo StateNAT impostato su FALSE.
Anche i nodi prelevati dal file di configurazione, attualmente stabiliti dal plugin PariDHT, vengono ritenuti come nodi reali per facilitare l'inizializzazione della rete. In futuro quando non ci sarà più il bisogno di tali nodi, si procederà ad eliminarli o a considerarli con lo StateNAT uguale a MAYBE, che verrà modificato in seguito in base all'IPDiscover di ConnectivityNIO.

Cambiamento porta UDP

Ultimo ed importante aspetto attuato, è stata la modifica della porta UDP utilizzata per i messaggi di risposta ai nodi con StateNAT uguale a TRUE o MAYBE.
Un nodo mascherato da NAT quando deve inviare un pacchetto DHTPacket nella rete P2P imposta i seguenti dati che lo identificano:

- **IP pubblico della NAT**: ottenuto grazie all'utilizzo del metodo queryWhoAmIServer() presente nella classe DHTUtility. In futuro tale metodo

dovrà essere modificato in modo da renderlo un servizio distribuito poiché attualmente fa rifermento ad un server;

- **Porta UDP privata**: selezionata dall'utente del nodo mascherato;
- **ID**: calcolato come specificato precedentemente;
- **NAT bit**: con valore corrispondente allo StateNAT del nodo, in questo caso 1.

Una volta inviato il pacchetto dal computer dell'utente, il NAT lo filtrerà e procederà con l'aggiunta del header UDP, formato da queste principali componenti: il suo IP pubblico e la porta UDP utilizzata dalla NAT, che spesso differisce dalla porta impostata dall'utente.
Il procedimento può essere visto graficamente in questo modo:

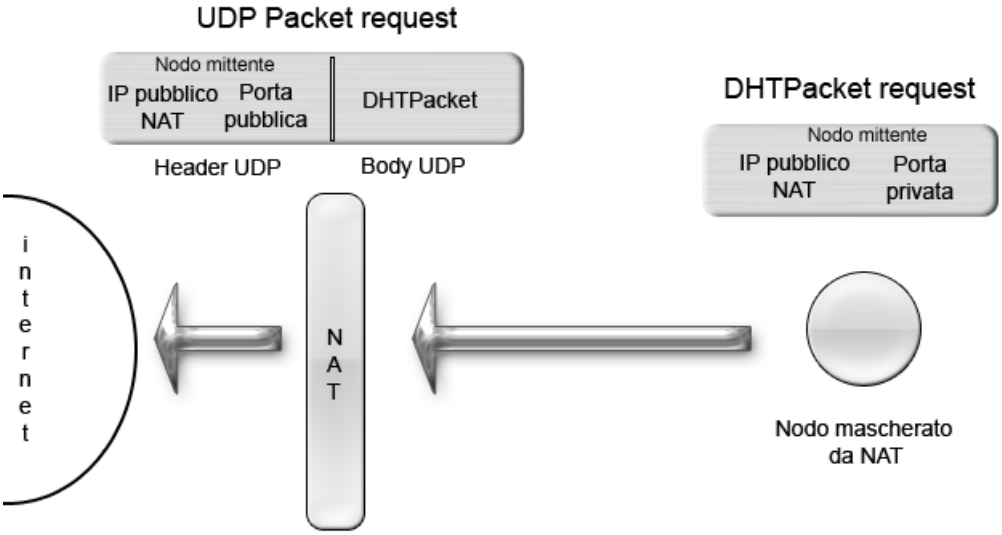

Figura 5.4: Messaggio inviato da un nodo mascherato da NAT.

Il nodo ricevente per rispondere a tale messaggio verifica come lo StateNAT contenuto nel DHTPacket sia impostato: NAT bit uguale a 1 indica lo stato su TRUE/MAYBE, NAT bit uguale a 0 indica lo stato su FALSE. In quest'ultimo caso risponderà normalmente, cioè considerando il nodo mittente come reale e quindi utilizzando l'IP e la porta trovati nel DHTPacket. Nel primo caso invece il nodo mittente risulta essere mascherato da NAT, quindi si procederà a rispondere servendosi dell'IP del DHTPacket ricevuto, che coincide con l'IP pubblico

della NAT, e come porta UDP viene sostituita la porta presente nel DHTPacket con quella presente nell'header UDP. Quest'ultima porta è quella reale adoperata dalla NAT per interfacciarsi con la rete.
Una volta realizzata la risposta voluta dal mittente, si può inviare il messaggio di ritorno e senza difficoltà la NAT riuscirà a farlo recapitare al nodo iniziale interessato.
Per implementare tale caratteristica è stata modificata la classe NetReceiver, sostituendo la porta del DHTPacket ricevuto, con la porta reale UDP utilizzata dalla NAT, nel caso in cui il NAT bit sia uguale ad 1.

Capitolo 6

Conclusioni e Sviluppi futuri

La prima parte di questo elaborato tratta l'eliminazione del vecchio plugin Connectivity e l'aggiornamento all'impiego del "nuovo" plugin ConnectivityNIO in PariDHT, passaggio chiave che deve obbligatoriamente essere compiuto da tutti i plugin all'interno di PariPari, se hanno la necessità di instaurare dei collegamenti di rete e quindi comunicare con gli altri peer.
Ciò in previsione dell'attivazione fra breve del PariPariSecurityManager, un modulo del Core che permetterà di gestire in modo migliore la sicurezza di tutto il progetto. Tale servizio, derivante dal Security Manager di java, creerà una politica di controllo sulle operazioni che verranno eseguite all'interno di PariPari verificando se un plugin ha le autorizzazioni necessarie per effettuarle.
In particolare, ConnectivityNIO sarà l'unico plugin che potrà instaurare delle connessioni di rete quindi di conseguenza il plugin Connectivity non potrà più essere utilizzato per tale scopo.
Nella seconda parte dell'elaborato sono stati trattati i cambiamenti effettuati nell'intero plugin per implementare le caratteristiche che permettessero ai nodo mascherati da NAT di comunicare più facilmente nella rete. Inoltre tali modifiche, necessarie ed indispensabili finché si continuerà ad utilizzare l'IPv4, hanno consentito di differenziare i nodi reali da quelli nascosti da NAT. In questo modo si è migliorata la ricerca/salvataggio di dati senza il rischio di errori o possibili complicazioni dovute alla presenza delle NAT.
Quando si effettuerà il passaggio all'IPv6, le NAT, il servizio NAT traversal implementato in ConnectivityNIO e tutte le caratteristiche introdotte nella seconda parte di questo elaborato diventeranno probabilmente obsolete e potranno essere

eliminate, anche se in futuro non è detto che potrebbero tornare utili.

Concludendo, ritengo l'intera esperienza molto utile, soprattutto perché mi ha preparato al lavoro di gruppo e quindi a tutti quegli aspetti rilevanti che devono essere tenuti in considerazione, come la gestione organizzativa, il coordinamento tra i partecipanti e le discussioni per le soluzioni di problematiche.
Grazie a questo progetto ho inoltre appreso nuove tecniche di programmazione utilizzate sia per lo sviluppo del codice sia per il test dello stesso. Ritengo opportuno che sia utile integrare ogni corso informatico con esperienze pratiche simili a questa, così da rendere lo studio meno teorico e più pratico, perciò molto più utile ai fini lavorativi futuri.

Per quanto riguarda gli sviluppi futuri di tali argomenti, è da ritenere opportuno l'aggiunta ad ogni nodo connesso alla rete P2P di PariPari di queste caratteristiche:

1. nel caso in cui un nodo riceva un messaggio da un peer sconosciuto nella porta utilizzata da PariDHT, allora tale nodo non sarà mascherato da NAT quindi si potrà cambiare lo StateNAT da MAYBE a FALSE;

2. nel caso generale di passaggio dello StateNAT da MAYBE a FALSE, il nodo interessato dovrà rieffettuare nuovamente la pubblicazione in DHT, così da farsi considerare nella rete come un nodo reale e quindi con tutte le caratteristiche già illustrate precedentemente. Teoricamente la ripubblicazione consiste in un insieme di Ping da effettuare ai nodi vicini.

Tuttavia entrambe queste caratteristiche sono comunque ancora in fase di pianificazione, quindi non implementabili attualmente poiché non sono stati definiti in modo chiaro i punti cardine da seguire.

Bibliografia

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

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

[3] Wikipedia. http://en.wikipedia.org/wiki/Kademlia

[4] Wikipedia. http://en.wikipedia.org/wiki/Denial-of-service_attack

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

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

[7] Wikipedia. http://en.wikipedia.org/wiki/Application_programming_interface

[8] Javadoc di PariPari. http://www.paripari.it/javadoc/

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

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

[11] Wikipedia. http://it.wikipedia.org/wiki/Debugging

[12] Wikipedia. http://it.wikipedia.org/wiki/Unit_test

[13] Wikipedia. http://en.wikipedia.org/wiki/Mock_object

[14] Wikipedia. http://en.wikipedia.org/wiki/SHA-1

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

[16] Wiki di PariPari. `http://www.paripari.it/mediawiki/index.php/Connectivity_en`

[17] Wikipedia. `http://it.wikipedia.org/wiki/IPv4`

[18] Wikipedia. `http://it.wikipedia.org/wiki/IPv6`

[19] Wikipedia. `http://en.wikipedia.org/wiki/Tor_(anonymity_network)`

Simone Giacon. *PariPari: DHT 2008.* Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Padova, A.A. 2008/2009.

Petar Maymounkov and David Mazières. *Kademlia: a peer-to-peer information system based on the XOR metric.* Lecture notes in computer science, 2002.

Loris Corona. *PariPari: NAT Traversal per ConnectivityNIO.* Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Padova, A.A. 2009/2010.

Giacomo Portolan. *PariPari: NAT Traversal.* Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Padova, A.A. 2008/2009.

F. Audet, C. Jennings. *RFC4787: Network Address Translation (NAT) Behavioral Requirements for Unicast UDP,* 2007.

A.S. Tanenbaum. *Reti di calcolatori - Quarta Edizione.* Pearson Education Italia, Milano, 2003.

Elenco delle figure

Elenco delle tabelle