UNIVERSITA' DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

PARIPARI: NAT Traversal per Connectivity NIO

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Loris Corona

A.A. 2009 - 2010

Indice

Sommario V

Introduzione 1

Capitolo 1 : PariPari 3

1.1 Il Progetto 3

1.1.1 Architettura 4

1.1.2 Implementazione 6

1.2 Il Gruppo 6

Capitolo 2 : Connectivity NIO 7

2.1 Il Plugin 7

2.2 La versione NIO 8

2.2.1 Assegnazione NAP 9

2.2.2 Gestione flussi di dati 11

Capitolo 3 : NAT Traversal 13

3.1 IP Discover 15

3.2 STUN 17

3.3 Server NAT 19

3.3.1 Login 21

3.4 UDP Hole Punching 22

3.5 Package 27

Conclusioni 29

Bibliografia 31

Elenco figure 33

SOMMARIO

Negli ultimi anni stanno prendendo sempre più piede software e tecnologie che sfruttano le potenzialità della rete Internet. Oggi i più usati sono applicazioni di reti peer-to-peer (P2P) per il file-sharing, come eMule e BitTorrent, applicazioni per la telefonia e videotelefonia via web, esempio VoIP ed altri programmi che offrono servizi di chat e messaggistica istantanea. L'utente medio che utilizza questi prodotti non ne conosce il reale meccanismo di funzionamento e non possiede competenze avanzate in ambito informatico per poterli configurare al meglio.
Per questo motivo ognuno di noi vorrebbe che questi strumenti siano i più semplici possibili da utilizzare senza doverci preoccupare di impostare/aprire porte, impostare limiti di banda, preoccuparci della presenza di NAT (Network Address Translation), senza doverci preoccupare di gestire molti software che la maggior parte dei casi entrano in conflitto tra loro limitandosi a vicenda. Per questo motivo si è deciso di realizzare un unico prodotto che integri tutte queste applicazioni di uso comune rendendo la vita più semplice possibile agli utilizzatori futuri.

Il progetto **PariPari** entra in quest'ottica implementando una piattaforma che unisca la maggior parte delle tecnologie web di uso comune e che sia di facile utilizzo anche all'utente più inesperto.

Lo scopo di questa tesi consiste nel presentare l'evoluzione del plugin Connectivity e di presentare una tecnica di NAT Traversal per facilitare la comunicazione all'interno della rete PariPari, senza che l'utente si debba preoccupare se un nodo sia o meno mascherato da NAT.

Introduzione

Una delle caratteristiche fondamentali per il funzionamento di una rete informatica è la presenza di un meccanismo che identifichi in modo univoco ogni computer o dispositivo collegato alla rete. Di conseguenza abbiamo bisogno di uno schema di indirizzamento locale, nel caso di una rete domestica, o globale, nel caso della rete Internet, in cui non esistano due host identificati dallo stesso indirizzo. Basti pensare, per esempio, al servizio postale. Se un indirizzo postale mi permettesse di raggiungere dieci persone diverse ed io devo spedire mille euro ad una di esse, come fa il postino a sapere a chi consegnare il denaro ? per essere sicuro che il destinatario riceva il pagamento potrei inviare mille euro ad ognuna delle persone identificate ma si capisce subito che questa soluzione non è molto vantaggiosa. Quindi abbiamo bisogno di un indirizzo univoco che identifichi uno ed un solo individuo, che nel nostro caso sarà un host o un dispositivo collegato alla rete.
Questa proprietà di unicità, nella rete web, è affidata al protocollo IPv4[1] che mette a disposizione uno spazio di indirizzamento a 32 bit con il quale possiamo identificare univocamente circa 4 miliardi di host. L'idea originaria era quella di fornire ad ogni nuovo utilizzatore di Internet un indirizzo IP univoco statico cioè un indirizzo che fosse sempre lo stesso ad ogni connessione. In breve tempo, grazie al progresso tecnologico, alla diffusione della banda larga e hai prezzi non più proibitivi dei computer, il numero di utenti collegati in rete aumentò a dismisura portando ad un veloce esaurimento di tali identificatori. Si può capire molto facilmente che se venissero allocati tutti gli indirizzi disponibili in modo statico non ci sarebbe più la possibilità di collegare in rete nessun altro dispositivo per la mancanza di identificatori univoci.
Una soluzione definitiva a questo problema si avrebbe con l'entrata in uso del protocollo IPv6[2], un'evoluzione del protocollo IPv4, che metterebbe a disposizione uno spazio di indirizzamento a 128 bit con la possibilità di identificare all'incirca 2^{128} host, numero sufficiente per assegnare un indirizzo IP statico a tutti i computer della terra per moltissimi anni[3]. Fino ad allora si dovranno sviluppare tecniche che ci permettano di risparmiare questi identificatori ormai in via di esaurimento.

---

[1] http://it.wikipedia.org/wiki/IPv4
[2] http://it.wikipedia.org/wiki/IPv6

[3] *"Lo spazio di indirizzamento di IPv6 sarebbe in grado di fornire più di 1500 indirizzi per ogni piede quadrato di superficie terrestre, un valore che probabilmente ci sarà sufficiente anche quando i tostapane sul pianeta Venere avranno indirizzi IP"*, cfr. [3]

Alcune di queste tecniche attualmente adottate sono: il CIDR[1] (Classless InterDomain Routing), il Subnetting e il NAT (Network Address Translator).
L'idea di base del NAT è il fatto che tutti gli host che possono comunicare tra loro attraverso Internet non hanno bisogno di un indirizzo univoco. Al contrario, e' possibile assegnare ad un host un "*indirizzo privato*" che non sia necessariamente unico globalmente ma che sia unico in un ambito più ristretto, ad esempio all'interno della rete aziendale in cui si trova. Per realizzare ciò utilizza un meccanismo che traduce gli indirizzi privati, univoci localmente, in "*indirizzi pubblici*", univoci globalmente. Il vantaggio di questa tecnica risale al fatto che normalmente questo dispositivo riesce a svolgere il proprio compito avendo a disposizione un numero di indirizzi pubblici molto minore del numero di indirizzi che sarebbero necessari se ciascun host dell'azienda avesse un indirizzo globalmente unico, facendo così risparmiare identificatori.
Per una rete P2P, come PariPari, questo meccanismo di traduzione può far sorgere delle complicazioni di comunicazione tra nodi interni e nodi esterni alla rete privata. Alcune di queste nascono dal fatto che i router domestici, i quali implementano il NAT, non consentono connessioni in ingresso, verso la rete privata, provenienti da host sconosciuti della rete pubblica trovandosi così "tagliati fuori" dalla rete P2P.
Obbiettivo di questa tesi è quello di sviluppare una tecnica di NAT Traversal che aggiri automaticamente questo problema permettendo a qualsiasi host mascherato da NAT di raggiungere ed essere raggiunto da qualsiasi nodo della rete PariPari.
Le metodologie utilizzate consistono nell'individuazione del proprio indirizzo IP pubblico, utilizzando la tecnica di IP Discover, la verifica della presenza di virtual server, attraverso il protocollo STUN (Session Traversal Utilities for Network Address Translator) e sfruttando la tecnica UDP Hole Punching, permettere lo scambio di pacchetti tra nodi interni e nodi esterni utilizzando un terzo nodo si supporto ("Server NAT"). Nei prossimi capitoli verrà presentato in maggior dettaglio il lavoro svolto; nel primo capitolo sarà esposta una panoramica sulla struttura attuale della rete PariPari, sull'organizzazione del lavoro e sulle decisioni prese per rendere il tutto più facile e più efficiente. Il capitolo successivo sarà dedicato alla presentazione del plugin Connectivity NIO e ai suoi sotto-plugin, soffermandosi principalmente sulle caratteristiche che lo differenziano dalla versione precedente e sui servizi che esso offre, tra i quali il NAT Traversal. Nell'ultimo capitolo si approfondirà in dettaglio il protocollo NAT Traversal, gli usi, le soluzioni implementate, i problemi riscontrati ed i servizi che attualmente offre agli utenti della rete PariPari. Per finire, nelle conclusioni, verrà esposto lo stato attuale del lavoro e i possibili sviluppi futuri dell'applicazione.

[1] http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

Capitolo 1

PariPari

1.1 Il Progetto

Con il termine **peer-to-peer** (o **P2P**) s'intende una rete di computer o qualsiasi rete informatica formata da un numero di nodi equivalenti (*peer*), cioè da nodi che fungono sia da cliente (client), richiedendo servizi, che da servente (server), offrendo servizi agli altri peer.
I vari nodi che compongono la rete possono differenziarsi nella configurazione locale, nella velocità di elaborazione dati, nell'ampiezza di banda disponibile e nella quantità di spazio di memorizzazione utilizzabile. Inizialmente le reti P2P erano reti ibride cioè si basavano su uno o più server centrali con il compito di mettere in contatto tra loro i vari nodi senza però essere utilizzati per il trasferimento dati che avveniva tra peer. Un esempio di quest'architettura si ha con la rete Napster, famosa per lo scambio di file musicali, che utilizzava un server centrale per mettere in collegamento due nodi qualsiasi che volevano scambiarsi informazioni. Ovviamente s'intuisce subito la fragilità di tale struttura, se il server dovesse guastarsi o bloccarsi tutta la rete sarebbe compromessa.

PariPari è un progetto software del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova sviluppato da studenti, principalmente laureandi triennali e magistrali. Esso mira a creare una rete P2P serverless, semplice, efficiente, espandibile, multifunzionale che garantisca l'anonimato dei nodi.

Essendo una rete serverless, cioè non basata su un server centrale, risulta essere molto robusta dal momento che la caduta, o semplicemente la disconnessione di alcuni peer non influenzano le prestazioni dell'intera rete. Lo scopo di tale progetto è di inglobare e distribuire tutti i più comuni servizi disponibili su Internet, quali File sharing, telefonia, istant messaging e renderne il facile utilizzo a tutti gli utenti tramite, per esempio, l'autogestione della banda e delle risorse disponibili. Inoltre permette ad altre persone di realizzare dei propri servizi ad hoc integrandoli nella rete senza nessuna difficoltà rendendoli così disponibili a tutti ed ampliando le potenzialità di PariPari.

1.1.1 Architettura

Per raggiungere gli obiettivi prefissati e per semplificare il lavoro di sviluppo, PariPari utilizza un'architettura modulare a plugin. Al centro di questa architettura troviamo il *Core*, il motore di PariPari, con il compito di fare da tramite tra i vari plugin della rete. Attorno al Core, in quella che noi chiamiamo *cerchia interna*, troviamo quei plugin che hanno il compito di gestire le risorse come la banda, le connessioni, lo spazio di memorizzazione e gli identificatori dei vari nodi e file. Esternamente, in quella che noi banalmente chiamiamo *cerchia esterna,* troviamo tutti gli altri plugin che, utilizzando le risorse rese disponibili dalla cerchia interna, erogano servizi specifici a tutti i plugin e utenti della rete. In seguito descriveremo alcuni di questi plugin e i servizi che essi offrono.

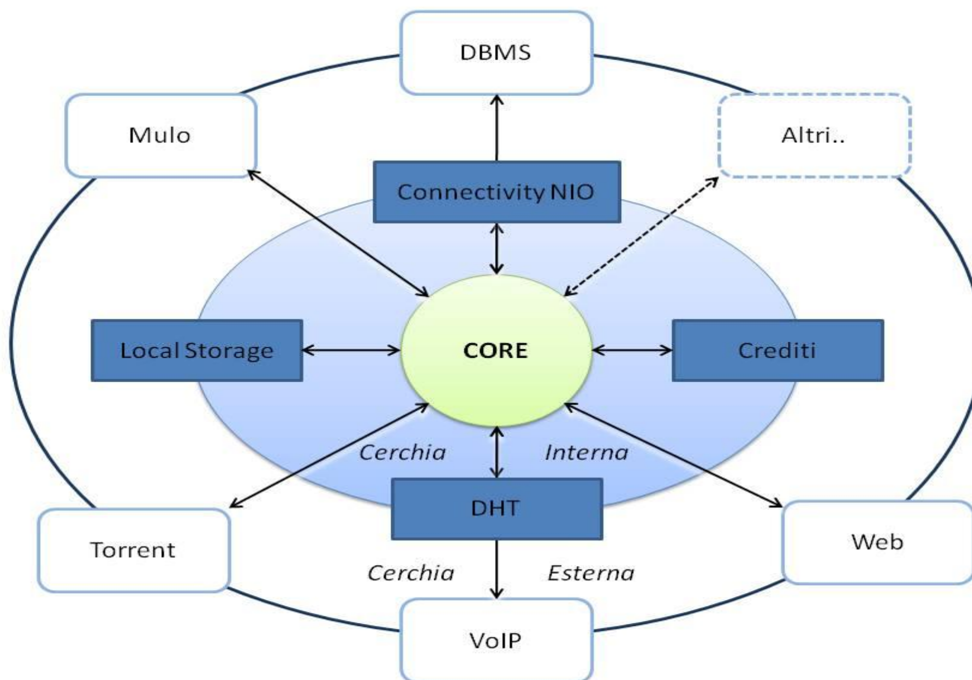

Figura 1.1: Architettura modulare di PariPari

Riferendoci alla figura sovrastante notiamo che tutti i moduli sono incentrati attorno al Core: esso, come già detto, si occupa di coordinare le richieste di risorse tra i vari plugin della cerchia esterna e quelli della cerchia interna sincronizzando, dove necessario, le varie applicazioni. Nella cerchia interna troviamo:

- **Connectivity NIO**: il suo compito è di amministrare la banda disponibile in ingresso e uscita, suddividendola in base alle varie esigenze dei plugin. In aggiunta tramite i suoi sotto-plugin offre servizi di multicast, anonimato e NAT Traversal.

- **DHT:** è il plugin responsabile della memorizzazione degli indirizzi di risorse e di utenti. Per tale scopo utilizza tabelle distribuite.

- **Local Storage**: ha il compito di memorizzare informazioni riguardanti impostazioni e dati temporanei. Tali informazioni sono memorizzate localmente in cartelle separate per ogni plugin.

- **Crediti**: regola l'uso delle risorse da parte dei vari plugin con un sistema di monetizzazione bastato su crediti. Ogni risorsa ha un costo, in funzione della quantità richiesta e del tempo di utilizzo. Solo coloro che hanno crediti a sufficienza possono usufruirne.

Nella cerchia esterna troviamo quei plugin che offrono diversi servizi oggigiorno molto utilizzati. Alcuni di questi sono:

- **Mulo e Torrent**: due applicazioni molto famose e largamente utilizzate che offrono un servizio di file sharing. Esse si appoggiano a reti diverse eDonkey, Kad e Gnutella.

- **Distributed Storage:** offre un servizio che consente di memorizzare i file in modo distribuito e ridondante nella rete PariPari in modo che siano sempre accessibili a tutti.

- **WEB**: mette a disposizione un WEB server distribuito.

- **IM e IRC**: due plugin che offrono servizi di messaggistica istantanea e chat.

- **VoIP:** un’altra applicazione ormai largamente utilizzata che mette a disposizione un servizio di comunicazione vocale sfruttando il protocollo IP.

Questa struttura modulare rende PariPari una piattaforma sicura da eventuali intrusioni e facilmente espandibile, dal momento che un qualsiasi programmatore potrà, in un futuro, sviluppare in modo semplice e veloce un proprio plugin che fornirà nuovi servizi all’intera rete.

1.1.2 Implementazione

Per la realizzazione del progetto PariPari si è scelto di usare il linguaggio di programmazione Java. Le principali motivazioni che hanno portato a tale scelta sono, innanzitutto, la propensione di questo linguaggio a una programmazione fortemente modulare, la semplice portabilità tra i vari sistemi operativi, l'integrazione dell'applicativo con il browser web grazie a Java web start e inoltre al fatto che sostanzialmente è uno dei pochi linguaggi visti e studiati presso questo dipartimento. Una conseguenza negativa è che Java risente di alcuni limiti non trascurabili, come ad esempio l'impossibilità di gestire direttamente locazioni di memoria, il supporto non nativo della rappresentazione numerica priva di segno, le scarse prestazioni in determinate operazioni e la pesantezza, in termini di spazio occupato, del software. Per quanto riguarda la stesura del codice è stato deciso di usare un modello di programmazione simile al XP[1] (Extreme Programming) che si basa essenzialmente sulla stesura di codice di test per verificare l'effettivo funzionamento del codice sorgente evitando così un debuggin[2] futuro. Inoltre viene usato un sistema di controllo versione (SVN[3]) per la gestione e condivisione del codice sorgente tra i vari gruppi di lavoro e un sistema centralizzato di bug – tracking (Bugzilla[4]) per la segnalazione e gestione di bug che, dopo un'accurata analisi, " dovrebbero" essere risolti.

1.2 Il Gruppo

In questo momento il gruppo di lavoro che sviluppa il progetto conta quasi un centinaio di persone, tra cui la maggior parte laureandi della triennale e della magistrale, da che si può capire la difficoltà di organizzazione e gestione dell'intero gruppo visto che tale numero, con gli anni, e' destinato ad aumentare notevolmente. Per affrontare questo problema, oltre alle scelte descritte nel paragrafo precedente, si è deciso di suddividere il lavoro in gruppi e sottogruppi dove ognuno si occupa di implementare e testare un determinato modulo del progetto.
Ogni gruppo ha un suo capo gruppo, di solito un tesista della magistrale, che dirige e coordina il lavoro del proprio team e fa da tramite sia con gli altri gruppi che con il capo progetto. All'interno del gruppo o sottogruppo si tenderebbe a fare un'ulteriore suddivisione tra sviluppatori e tester, quest'ultimi con il compito di testare il codice scritto dagli sviluppatori controllandone la correttezza.

[1] http://www.extremeprogramming.org/
[2] http://it.wikipedia.org/wiki/Debugging
[3] http://svnbook.red-bean.com/
[4] http://www.bugzilla.org/

Capitolo 2

Connectivity NIO

2.1 Il Plugin

Come detto in precedenza, il plugin Connectivity NIO, fa parte della così detta cerchia interna cioè fa parte di quel gruppo ristretto di plugin che gestiscono le risorse, quindi fondamentali per il corretto funzionamento dell'intera rete. Esso si occupa della gestione della banda disponibile in ingresso e uscita, dell'instaurazione e gestione delle connessioni tra nodi, offrendo anche un servizio di NAT Traversal a quei peer che non riescono a comunicare direttamente tra loro perché nascosti da NAT. Si rimanda al capitolo tre per maggiori approfondimenti su quest'argomento. Inoltre, tramite i suoi sotto-plugin, offre servizi di anonimato e multicast. Tali sotto-plugin sono:

- **Anonimato**[1]: questo sotto-plugin si occupa di garantire l'anonimato dei nodi che instaurano una comunicazione tra loro. Per raggiungere tale obbiettivo utilizza una tecnica di Onion Routing per garantire l'anonimato del mittente e una tecnica di Fake Recipient per garantire l'anonimato del destinatario. Inoltre si occupa di oscurare l'identità dei peer che effettuano pubblicazioni o ricerche in DHT.

- **Multicast**[2]: offre un servizio di connessione multicast per applicazioni di telefonia, VoIP[3] (Voice over IP) e di chat, IRC[4] (Internet Relay Chat). Per far ciò utilizza tecniche di Simple Conference ed Advance Conference, la prima basata sulla centralizzazione dello smistamento dei messaggi, la seconda, più attinente all'ideologia P2P, basata sulla distribuzione del lavoro tra i vari nodi della rete.

---

[1] http://www.paripari.it/mediawiki/index.php/Anonimato
[2] http://it.wikipedia.org/wiki/Multicast
[3] http://www.paripari.it/mediawiki/index.php/VoIP_en
[4] http://www.paripari.it/mediawiki/index.php/IRC

La banda disponibile di connessione è, come ovvio, una risorsa limitata ed usata da tutti per trasmettere informazioni, richieste, risposte alle richieste e quant' altro. Come ogni risorsa limitata ed usata da più di un utilizzatore essa e' soggetta a contesa tra i vari plugin e utenti i quali, se non controllati, possono abusarne impedendone l'utilizzabilità' ad altri. In ambito informatico questo problema prende il nome di *starvation*.
Per evitare ciò e garantire una corretta e più efficiente gestione della banda, ConnectivityNIO implementa una strategia che utilizza l'algoritmo *Token Bucket* integrato con un sistema di monetizzazione gestito dal modulo Crediti che, come detto nel capitolo precedente, imposta un costo in termini di crediti per ogni risorsa utilizzata, in questo caso la banda. Questo costo può variare a seconda di quanta banda si vuole utilizzare, per quanto tempo, quante connessioni si vogliono instaurare, etc. Solo coloro che hanno crediti a sufficienza possono servirsi di ConnectivityNIO richiedendo i servizi che esso offre.
In seguito si spiegheranno le differenze di questa nuova versione da quella precedente.

2.2 La versione NIO

In sostanza Connectivity NIO non è altro che un'evoluzione del vecchio plugin Connectivity[1] che per comunicare utilizzava i socket[2] impedendo trasmissioni non bloccanti. Questa nuova versione, come si può intuire dal suffisso del nome, fa uso del package java.nio[3], il quale definisce principalmente tre nuovi tipi fondamentali di oggetti: buffers, channels e selectors.

- **Buffers:** non sono altro che semplici contenitori dati nei quali inserire ed estrarre informazioni.

- **Channels** (canali): sono astrazioni di connessioni tra soggetti che vogliono comunicare tra loro.

- **Selectors** (selettori): non sono altro che oggetti che si occupano di multiplexare e demultiplexare i canali.

Le differenze attualmente implementate riguardano principalmente la modalità di assegnazione dei NAP (Network Access Point) e la modalità di gestione dei vari processi di lettura/scrittura che, attraverso gli oggetti resi disponibili da java.nio, saranno resi non bloccanti.

[1] http://www.paripari.it/mediawiki/index.php/Connectivity_en
[2] http://it.wikipedia.org/wiki/Socket_(reti)
[3] http://java.sun.com/j2se/1.4.2/docs/api/java/nio/package-summary.html

Per il resto funziona tutto come la versione precedente a parte il fatto che attualmente non si fa uso della libreria PluginSender[3].
Nelle prossime sezioni queste differenze verranno spiegate più nel dettaglio.

2.2.1 Assegnazione NAP

I **NAP** non sono altro che punti d’accesso alla rete attraverso i quali trasmettere e ricevere informazioni. Possiamo immaginarli come punti d’ingresso ed uscita ad un canale di comunicazione nel quale passa un flusso di dati. Essi possono essere concessi sia in modalità bloccante sia in modalità non bloccante, in seguito verrà spiegata la differenza, inoltre tali NAP differiscono a seconda del protocollo di trasporto che si vuole utilizzare per la trasmissione. I NAP implementati fin ora sono:

- PPTcpNap: punto d’accesso per connessioni TCP in uscita.
- PPUdpNap: punto d’accesso per connessioni UDP sia in uscita che in entrata.
- PPTcpServer: punto d’accesso in attesa di connessioni TCP in entrata.

Attualmente, alla richiesta di uno o più NAP da parte di un plugin, il modulo ConnectivityNIO esegue una determinata procedura di registrazione prima di assegnarglielo. Più nello specifico:

1. Registra il plugin richiedente, es. “Mulo”, all’interno del “database” di ConnectivityNIO impostando la banda limite in ingresso (*bandin*) ed in uscita (*bandout*) che tale plugin vuole utilizzare per la trasmissione. Per la precisione non è proprio un database. Ogni plugin ha un oggetto del tipo DataPluginRecord (*record*) che contiene un Hashtable[1] con l’elenco dei wrapper associati al plugin, ma il concetto è proprio quello di un database.

```
Data_PluginDataBase  pluginData = new Data_PluginDataBase ( ILogger myLog );
pluginData.insertPlugin("Mulo", bandin,bandout );
IData_PluginRecord record  =  pluginData.getPluginRecord("Mulo");
```

2. Registra un oggetto del tipo Data_NAPWrapper (*napwrapper*) in una Hashtable contenuta nel record del plugin richiedente, come detto in precedenza, passandogli: il tipo di NAP che si vuole utilizzare, es. “UdpNap” per una trasmissione UDP, il tipo di modalità richiesta per il NAP (1 bloccante, 2 non-bloccante), il record del plugin richiedente ed altri

[1] http://www.paripari.it/mediawiki/index.php/PluginSender_en
[2] http://it.wikipedia.org/wiki/Hash_table

parametri quali un selettore (*nppDTManager*) e un file di Log (*myLog*) la cui funzionalità verrà spiegata nel prossimo sottocapitolo.

```
PPUdpNapAPIImpl UdpNap = new PPUdpNapAPIImpl (“ ”, bandin, bandout);
Data_NapWrapper napwrapper = new Data_NapWrapper(UdpNap, 2, nppDTManager, record,myLog);
```

3. Associa al napwrapper il NAP richiesto.

A questo punto il NAP è pronto per essere utilizzato per trasmettere o ricevere dati. Se in un secondo momento si richiede un nuovo NAP, ConnectivityNIO controlla se il plugin richiedente e'già' stato registrato nel proprio “database”. Se non lo è fa la procedura vista sopra, altrimenti aggiunge semplicemente, nella Hashtable contenuta nel record del plugin richiedente, un nuovo napwrapper con le informazioni richieste ed associa il nuovo NAP che si desidera utilizzare.

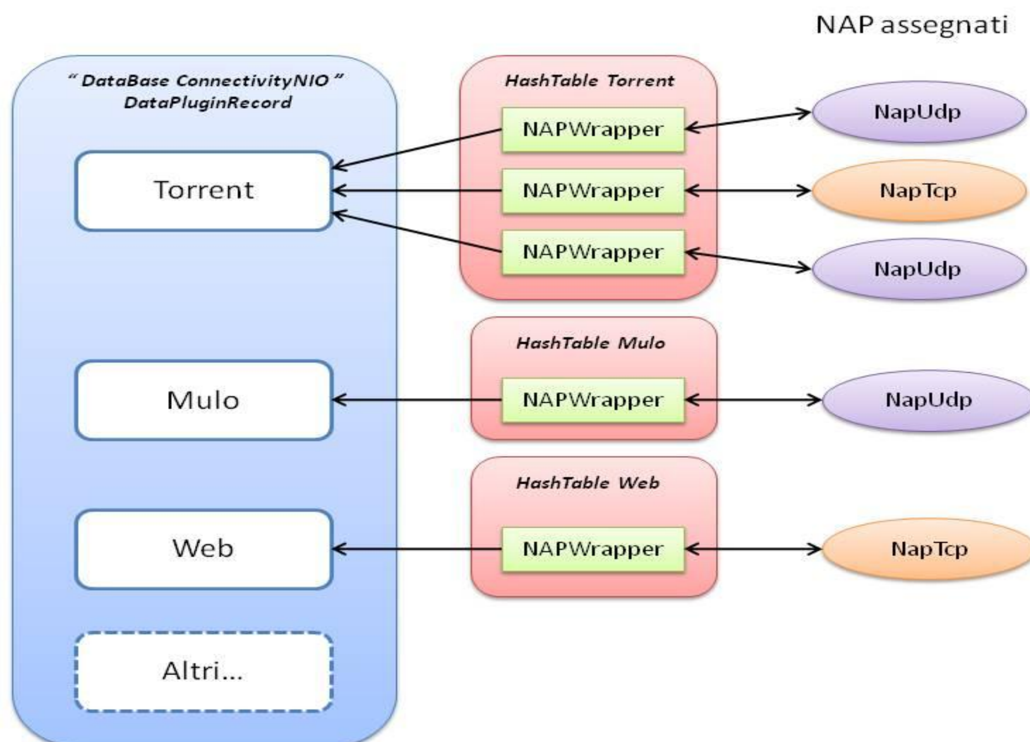

Figura 2.1: Assegnazione NAP

2.2.2 Gestione flussi di dati

Una volta ottenuto il NAP, il plugin richiedente è pronto ad inviare/ricevere dati. Questi dati sono immagazzinati in buffer o in code a seconda del protocollo di trasporto utilizzato. TCP immagazzina i dati in buffer mentre UDP in code[2]. Questa differenziazione è dovuta principalmente al fatto che una trasmissione TCP la si può rappresentare come un flusso continuo di byte mentre una trasmissione UDP la si può immaginare come una concatenazione di datagrammi che viaggiano uno dopo l'altro. Per questo abbiamo la necessità di due tipi di contenitori che ci rendano semplice e veloce ogni processo di inserimento ed estrazione di byte e datagrammi. Come visto nel paragrafo precedente, ad ogni napwrapper viene associato un selettore. Questo selettore è privato di ConnectivityNIO (Connectivity Java Nio Selector in seguito chiamato "selettore NIO"). Attualmente viene utilizzato solo un selettore privato, inizializzato dal thread[1] *tranferManager_NPP_Selector*, per tutti i NAP attivi. I buffer e le code menzionate precedentemente, attraverso il selettore NIO, vengono continuamente riempiti e svuotati.
Immaginiamo che un processo lettore, che svuota un buffer, sia più lento di un processo scrittore che lo riempie. Com'è facilmente intuibile, dal fatto che un contenitore non può immagazzinare infinite informazioni, prima o poi il buffer si riempirà. Il selettore NIO, una volta resosi conto che un buffer è pieno addormenta la relativa associazione NAP - selettore impedendo il passaggio di nuovi dati, quindi evitando che il NAP trasmetta dati a un buffer che non può contenerli perché pieno. In questo caso la trasmissione si bloccherà finche il buffer avrà spazio per contenere nuove informazioni. Se in un secondo momento, il selettore NIO si accorge che il buffer comincia ad essere svuotato risveglia l'associazione precedentemente addormentata, acconsentendo il passaggio di nuovi dati che andranno a riempire nuovamente il buffer. Qualora un NAP venisse richiesto in modalità non-bloccante, il plugin richiedente può anche richiedere un selettore (*PPSelector*) che diventerà di sua proprietà. Attualmente viene concesso un solo selettore privato per ogni plugin (varie richieste restituiranno lo stesso oggetto). Esso ha il compito di notificare quali buffer, tra quelli letti dal plugin, contengono almeno un byte di informazione, cioè quali buffer sono pronti per essere svuotati. Una volta che il processo lettore svuota un buffer, questo non viene più notificato dal selettore privato del plugin finche non verrà nuovamente riempito con almeno un byte di informazione. Questo garantisce che il processo lettore non si blocchi a causa della ricerca dei buffer da svuotare.

---

[1] http://it.wikipedia.org/wiki/Thread_(informatica)
[2] http://it.wikipedia.org/wiki/Coda_(informatica)
[3] http://it.wikipedia.org/wiki/Overflow

Per capire meglio il meccanismo fin qui descritto si rimanda alla figura seguente dove il plugin “Torrent” richiede NAP in modalità non-bloccante, con annesso un selettore privato, mentre il plugin “Mulo” li richiede in modalità bloccante.

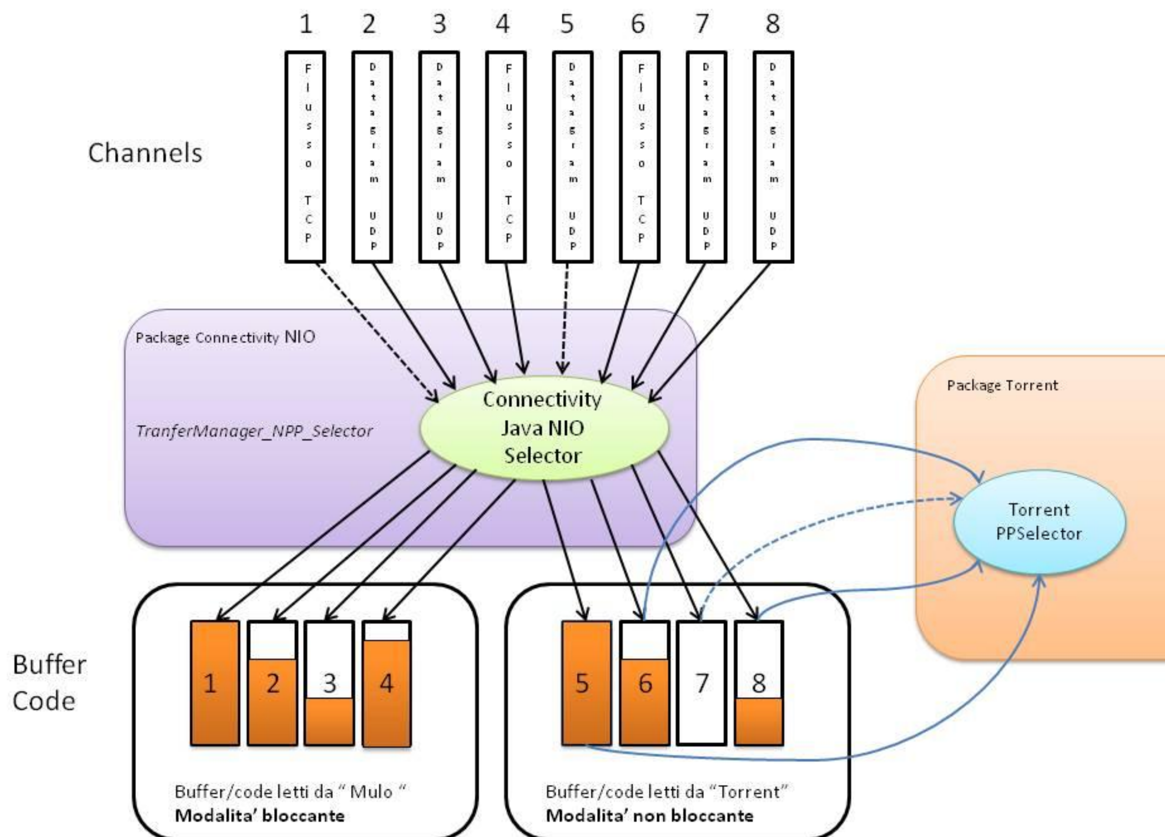

Figura 2.2: Gestione buffer/code in modalità bloccante e non-bloccante

Si può notare come le associazioni 1 e 5 vengano addormentate perché i relativi buffer sono pieni mentre l’associazione 7, nel selettore privato di Torrent, viene addormentata perché il relativo buffer è stato svuotato.

Capitolo 3

NAT Traversal

In questo capitolo verrà presentata l'implementazione della tecnica NAT Traversal per la rete PariPari. Essa, come già accennato, ha l'obbiettivo di garantire la comunicazione tra nodi aggirando NAT e firewall.
Nell' introduzione si è già parlato di firewall e virtual server senza però spiegarne il relativo funzionamento. I **firewall** sono dei dispositivi che, opportunamente configurati, filtrano il traffico in uscita e in ingresso dalla Internet secondo regole applicate agli indirizzi IP, alle porte e ai protocolli utilizzati. Il **virtual server** è una tecnica utilizzata nei router domestici che ci permette di impostare nel NAT alcune porte, con il relativo protocollo, per contattare un determinato host della rete privata. In egual modo possiamo dire che un router, con virtual server attivo, si comporta come un server con IP pubblico che rimane in ascolto su porte prestabilite accettando qualsiasi comunicazione entrante in tali porte.

Il funzionamento del NAT Traversal è abbastanza semplice; in fase di avvio, ConnectivityNIO si connette a DHT ricercando un nodo che offre sevizi di NAT Traversal (Server NAT). In caso la ricerca non dia nessun risultato si userà un Server NAT statico precedentemente allocato.
Ogni volta che un peer caricherà ConnectivityNIO controllerà se può fare o non può fare da Server NAT. In caso positivo e in caso si acconsenta a offrire servizi di NAT Traversal verrà pubblicato in DHT come Server NAT. Al contrario, se un nodo non può fare da Server NAT, dovrà loggarsi a uno di essi altrimenti non potrà essere raggiunto da nessun peer della rete.
Quando un peer vorrà trasmettere delle informazioni ad un altro, che non riesce a raggiungere direttamente perché mascherato da NAT, tramite la tecnica UDP Hole Punching potrà spedirgliele senza nessun problema. In seguito verranno spiegate in dettaglio le varie tecniche e protocolli. Attualmente l'implementazione del NAT Traversal utilizza il protocollo di trasporto UDP (utilizza datagrammi UDP) sia per un fatto di semplicità realizzativa ed implementativa sia per rispettare certi protocolli già esistenti. In futuro si potrà implementare il tutto utilizzando TCP, cosa che in campo ingegneristico non ha riscontrato sempre successo.

Questi datagrammi utilizzati sono nella forma generale: intestazione UDP (Header UDP) seguita dal payload, nel quale viene inserita in testa una pseudo-intestazione NAT Traversal di 34 byte (Header NAT Traversal) contenente: il servizio richiesto (NAT Traversal per l'appunto), il tipo di richiesta/risposta (richiesta/risposta IP Discover, richiesta/risposta STUN, etc..) e il numero di byte successivi utili per soddisfare la richiesta e/o attraverso i quali spedire la risposta.

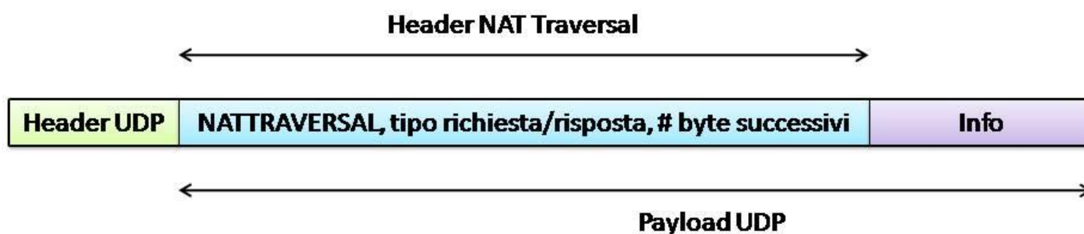

Figura 3.1: Pacchetto di richiesta/risposta generica

In seguito verrà presentata in dettaglio l'implementazione della tecnica IP Discover, del protocollo STUN e della tecnica UDP Hole Punching con le relative richieste e risposte. Per far ciò si utilizzeranno delle assunzioni di uso generale:

- Gli indirizzi IP, sia pubblici che privati, sono da ritenersi in versione IPv4, quindi formati da 4 byte.

- I termini Server o Server NAT si riferiscono ad dei semplici nodi PariPari che offrono servizi di NAT Traversal.

- Quando un host della rete interna ne contatta uno della rete esterna attraverso un NAT, una porta di quest'ultimo, quella utilizzata per il contatto, viene aperta fintanto che non scade un timeout. In altre parole possiamo dire che, finché non scade il timeout, l'host contattato può a sua volta contattare quello della rete interna senza nessun problema. Allo scadere del timeout viene cancellata l'associazione nel NAT fra la porta e l'indirizzo dell'host esterno che non potrà più contattare quello interno. Questo, come ovvio, è il normale meccanismo di funzionamento di un NAT che detiene le associazioni all'interno di una propria tabella.

3.1 IP Discover

IP Discover è un protocollo leggero utilizzato per rilevare il proprio indirizzo IP pubblico che sarà utilizzato, come vedremo in seguito, per scoprire se si è mascherati da NAT e/o protetti da firewall.
Il suo funzionamento è molto semplice ed intuibile; il nodo, che vuole conoscere il proprio indirizzo pubblico, invia una richiesta IP Discover ad un nodo che funge da Server NAT. Tale server verrà cercato in DHT attraverso chiavi di ricerca che vedremo in seguito. Trovato un Server ed inviatagli la richiesta esso risponderà al richiedente inviandogli una risposta IP Discover nella quale inserirà l'indirizzo del mittente estratto dalla richiesta, ovvero l'indirizzo pubblico utilizzato dal peer per navigare in Internet.

Richiesta IP Discover:

Risposta IP Discover:

Figura 3.2: Richiesta/Risposta IP Discover.

Una volta ricevuta la risposta il richiedente estrae l'indirizzo contenuto nel pacchetto e lo confronta con l'indirizzo della propria scheda di rete (indirizzo privato). Le situazioni che si possono presentare sono:

- L'indirizzo pubblico estratto è uguale al proprio indirizzo privato e il risultato non varia cambiando la porta di comunicazione. Questa situazione si ha quando il nodo non è mascherato da NAT né protetto da firewall.

- L'indirizzo pubblico estratto è uguale al proprio indirizzo privato ma cambiando la porta usata per inviare la richiesta non si riceve nessuna risposta. Questo è il caso in cui il nodo non è mascherato da NAT ma molto probabilmente è protetto da firewall che gli permette di usare solo alcune porte.

- L'indirizzo pubblico estratto è diverso dal proprio indirizzo privato e il servizio di IP Discover funziona su tutte le porte. Questo è il caso in cui il nodo è mascherato da NAT ma non è protetto da firewall.

- L'indirizzo pubblico estratto è diverso dal proprio indirizzo privato e il servizio funziona solo su alcune porte. Caso in cui il nodo è mascherato da NAT e molto probabilmente protetto da firewall.

- Il caso peggiore si ha quando non si riceve nessuna risposta utilizzando qualsiasi porta. In questo caso si può provare a contattare un altro Server NAT.

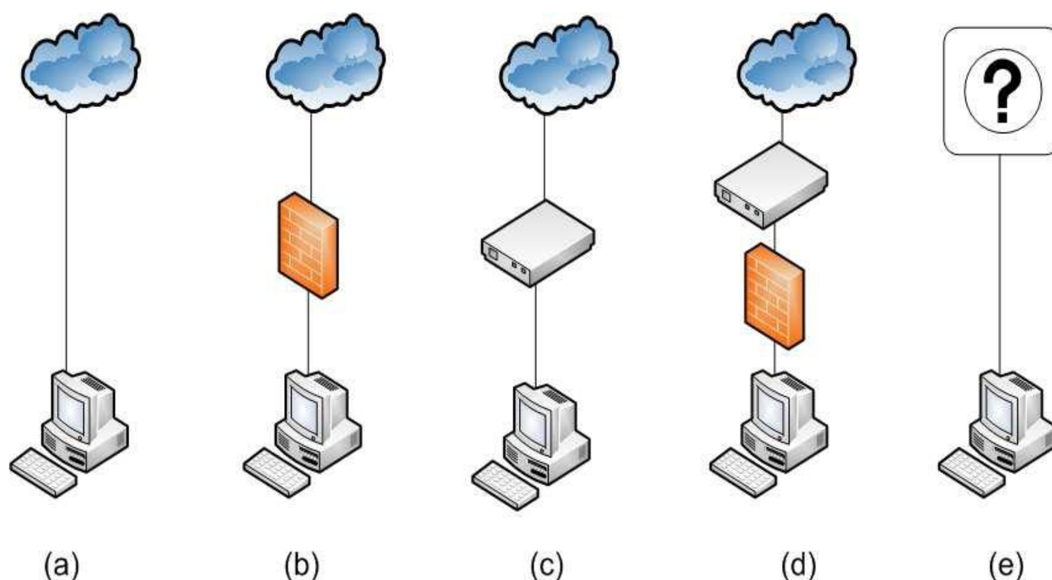

Figura 3.3: Possibili mascheramenti: (a) comunicazione libera, (b) presenza di un firewall,(c) presenza di un NAT, (d) presenza di NAT + firewall, (e) caso peggiore non si riceve nulla.

Se ci troviamo nei primi due casi non c'è bisogno di utilizzare NAT Traversal per essere raggiunti dagli altri nodi della rete. Anzi, si possono offrire i servizi di NAT Traversal proponendosi come Server NAT.
Nel terzo e quarto caso bisogna utilizzare il protocollo STUN per verificare se nel NAT è attivo un virtual server. Nell'ultimo caso, se non si riceve risposta da nessun Server NAT, molto probabilmente si è protetti da un firewall che impedisce tutto il traffico entrante. Questa situazione è molto critica perché il nostro nodo risulta irraggiungibile dal resto della rete PariPari.

3.2 STUN

Questo protocollo, oltre ad offrire un servizio simile a quello di IP Discover, mette a disposizione una tecnica per capire se un NAT è configurato con la tecnica virtual server cioè se nel NAT ci sono porte che permettono traffico entrante da peer a lui sconosciuti. Il suo funzionamento è molto simile al protocollo visto precedentemente. Il nodo invia una richiesta STUN ad un Server NAT, ricercato come prima attraverso chiavi di ricerca, il quale gli risponderà con due pacchetti contenenti l'indirizzo pubblico e la porta del mittente estratti dalla richiesta.
La prima risposta verrà inviata attraverso la porta sulla quale il richiedente ha inviato la richiesta, la seconda attraverso una porta diversa scelta arbitrariamente.

Richiesta STUN:

Risposta STUN:

Figura 3.4: Richiesta/Risposta STUN

Le situazioni che si possono presentare sono:

- Il richiedente riceve entrambe le risposte. È attivo il virtual server e il nodo, attraverso la porta utilizzata, può contattare chiunque ed essere a sua volta contattato da tutti i nodi della rete.

- Il richiedente riceve solo la prima risposta, quella inviatagli usando la porta sulla quale precedentemente era stata inviata la richiesta. Questa situazione si presenta quando il nodo è libero di comunicare con chiunque su quella porta ma non può ricevere connessioni da peer sconosciuti. In questo caso non è impostato nessun virtual server.

- Come nel protocollo precedente, il caso peggiore si ha quando il nodo non riceve nessuna risposta. In questo caso probabilmente le connessioni in ingresso o in uscita sono bloccate sulla porta utilizzata e quindi si dovrà ritentare su una porta diversa.

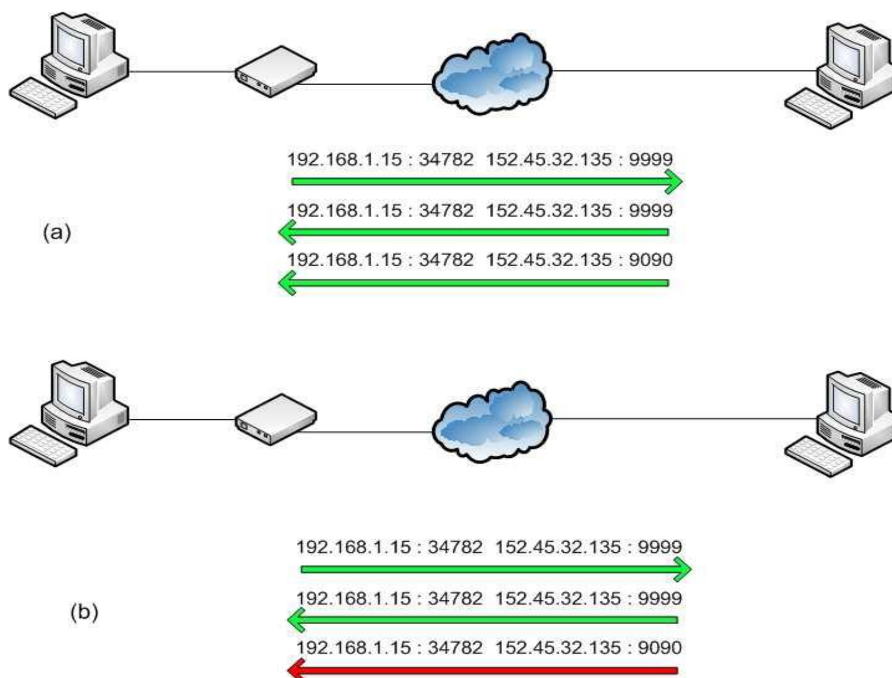

Figura 3.5: Casi STUN: (a) Virtual Server attivo, (b) normale mascheramento NAT

Il primo caso è quello più favorevole perché ci permette, anche mascherati da NAT, di contattare ed essere contattati da tutti. Anche in questo caso è possibile offrire servizi di NAT Traversal proponendosi come Server NAT.
Il secondo caso, quello più comune, si risolve usando l'UDP Hole Punching tecnica che vedremo in seguito.
Nel caso peggiore che tutte le porte risultino bloccate il peer sarà irraggiungibile dal resto della rete perché probabilmente protetto da un firewall che blocca tutto il traffico entrante.

Come abbiamo visto, grazie alla tecnica IP Discover e al protocollo STUN qualsiasi nodo nella rete può capire se è in grado o meno di fornire servizi di NAT Traversal cioè se può essere eletto Server NAT. I dettagli di tale nodo verranno presentati nella prossima sezione.

3.3 Server NAT

Un Server NAT, come già accennato, ha il compito di rispondere alle varie richieste di IP Discover, STUN e coordinare i peer durante l'UDP Hole Punching che vedremo nella prossima sezione. Si ricorda che un nodo non può scegliere liberamente se fornire i servizi di NAT Traversal, ma deve prima capire se è in grado di poterli erogare. Come visto precedentemente i casi in cui è possibile offrire tali servizi sono:

- Dopo aver scoperto, tramite IP Discover, che non si è mascherati da NAT e non si è protetti da firewall che blocca tutto il traffico entrante.

- Dopo aver scoperto, tramite protocollo STUN, che nonostante si sia nascosti da NAT in tale dispositivo è attivo il virtual server.

Un nodo che soddisfa una di queste condizioni e vuole offrire i servizi di NAT Traversal deve, come prima cosa, pubblicare in DHT le informazioni con le quali essere contattato e ricercato. Più nello specifico queste sono:

- Il proprio indirizzo IP privato se non si sia mascherati da NAT, altrimenti bisognerà pubblicare il proprio indirizzo IP pubblico rilevandolo tramite tecnica IP Discover.

- La porta sulla quale si aspetteranno le richieste dai vari peer: essa può essere una scelta arbitraria se il nodo non è nascosto da NAT né protetto da firewall (nel codice è stata scelta la 9999), altrimenti dovrà essere una porta che permette traffico entrante da peer sconosciuti, tale porta la si ricava con il protocollo STUN.

- Le chiavi di ricerca, menzionate nei sottocapitoli precedenti, con le quali essere ricercato in DHT. Attualmente queste chiavi sono: “Server”, “ServerNat”, “NatTraversal”.

Dopodiché deve avviare un thread che rimane in ascolto sulla porta pubblicata aspettando richieste da soddisfare.

Le richieste, viste precedentemente, alle quali il Server NAT risponde sono:

- Richieste di IP Discover. Il Server risponde inviando una risposta IP Discover al mittente.

- Richieste STUN. Il Server invia due risposte STUN al mittente, nella modalità vista precedentemente.

Inoltre risponde a:

- Richieste UDP Hole Punching. Il Server inoltra un'ulteriore richiesta UDP Hole Punching al nodo che si desidera contattare.

- Risposte UDP Hole Punching da parte di un nodo precedentemente contattato. Il Server inoltra, al peer che vuole contattare questo nodo, una risposta contenente le informazioni per raggiungerlo.

- Richieste di Login. Il Server ricerca nella propria cache se il richiedente è già stato registrato. In caso affermativo rinnova il lifetime, tempo di vita scaduto il quale la registrazione sarà cancellata, altrimenti viene registrato come spiegato successivamente.

L'utilità di quest' ultime richieste e risposte saranno più chiare in seguito quando presenteremo la tecnica UDP Hole Punching.
La cache, appena menzionata, contiene una tabella nella quale il Server inserisce le informazioni dei peer a lui loggati. In particolare, per ogni peer tiene in memoria:

IP privato	IP pubblico	Porta inoltro	Lifetime
...	...	...	...

Ad ogni richiesta di login, il Server estrae l'indirizzo pubblico del mittente e controlla nella propria cache se tale indirizzo compare. In caso affermativo aggiorna il relativo lifetime, scaduto il quale tutte le informazioni riguardanti tale peer saranno cancellate perché probabilmente esso è diventato inattivo. Nel caso che l'indirizzo pubblico estratto non fosse presente nella tabella verranno inserite le informazioni, sopracitate, riguardanti tale nodo. Queste informazioni, come vedremo in seguito, saranno utili per aggirare i NAT ed in particolare per il funzionamento della tecnica UDP Hole Punching. Nella sezione successiva vedremo più nel dettaglio la funzione utilizzata per loggarsi ad un Server NAT.
Nel caso il Server NAT riceva una richiesta/risposta contenente informazioni non pertinenti o corrotte risponde semplicemente al mittente inviando una risposta denominata "BadRequest" con la quale gli comunica che la richiesta/risposta da lui ricevuta non era corretta.

3.3.1 Login

Come detto in precedenza un nodo non può offrire servizi di NAT Traversal se è mascherato da NAT sul quale non è attivo nessun virtual server. In questo caso esso dovrà loggarsi ad un Server NAT per essere raggiunto dagli altri peer.
Questa fase di login viene eseguita da un nodo ogni volta che ne ha la possibilità per evitare sgradite sorprese. In particolare può succedere che:

- Un nodo sia loggato a un Server NAT inattivo o non raggiungibile per svariati motivi. Questa situazione porterebbe alla non raggiungibilità del nodo stesso da parte di tutti i peer della rete. Ad ogni login, effettuato con successo, si sarà sicuri di essere loggati ad un Server attivo.

- Il Server ha cancellato, dalla propria tabella, le informazioni per raggiungere il nodo perché è scaduto il relativo lifetime. Ad ogni distinto login il lifetime verrà aggiornato.

- Il NAT che maschera un nodo ha cancellato dalla propria tabella l'associazione Porta / IP Server attraverso la quale fa passare tutte le richieste provenienti dal Server. In questo caso quando il Server vorrà comunicare con il nodo non potrà più farlo perché il NAT lo considererà come un peer sconosciuto. Anche in questo caso, tale associazione verrà rinnovata ad ogni login.

Si ricorda che un Server NAT tiene nella propria cache una tabella nella quale, per ogni nodo a lui loggato, tiene le informazioni riguardanti: l'indirizzo privato del nodo, l'indirizzo pubblico e la porta sulla quale il nodo attende eventuali richieste da parte del Server. Per riempire questa tabella l'unica informazione che il Server non sa ricavare automaticamente è l'indirizzo privato del peer che si vuole loggare. Quindi tale informazione viene inserita nella richiesta di Login.

Figura 3.6 : Richiesta di Login

Una volta che il peer si è loggato avvia un thread che rimane in ascolto sulla porta usata per inviare la richiesta di login, aspettando pacchetti da parte del Server che, come vedremo nella prossima tecnica, saranno soprattutto inoltri UDP Hole Punching.

3.4 UDP Hole Punching

Con " UDP Hole Punching " si intende una tecnica che consente di stabilire connessioni UDP bidirezionali tra due computer appartenenti a reti private distinte. Questa tecnica è molto utilizzata da Skype per aggirare i firewall e dispositivi NAT. La situazione più comune nella quale ci possiamo trovare è che entrambi i peer siano mascherati da NAT, come in figura.

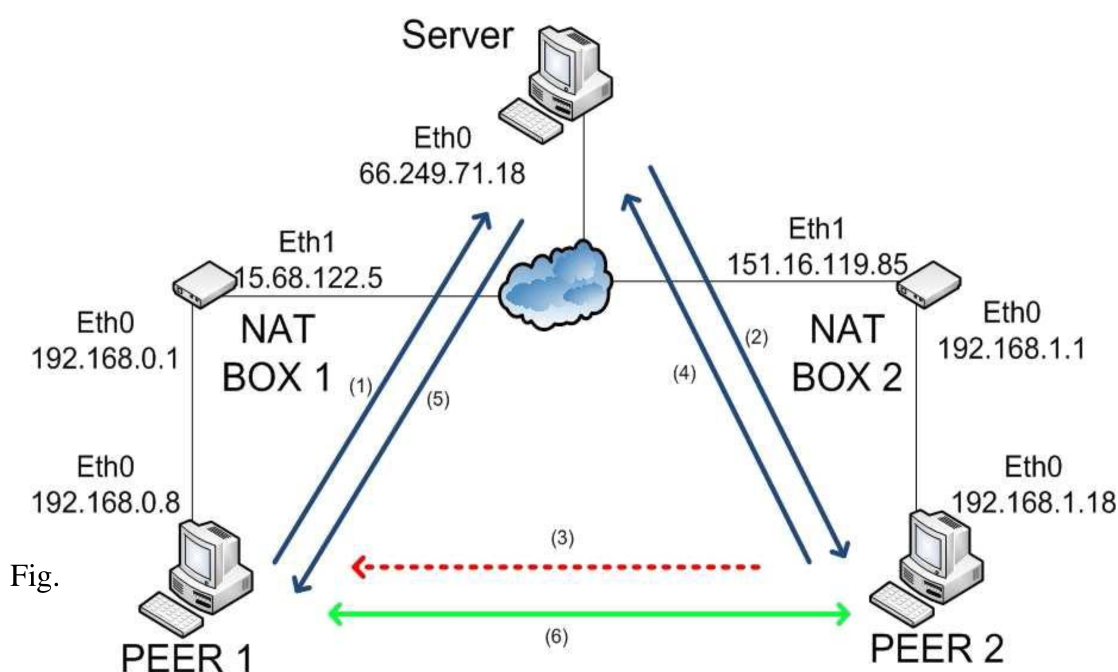

Fig.

Figura 3.7: Sincronizzazione UDP Hole Punching

Per spiegare il funzionamento di questa tecnica faremo riferimento alla figura sovrastante dove PEER 1 vuole contattare PEER 2 ed entrambi sono mascherati da NAT. Per la precisione, la condizione che PEER 1 sia offuscato da NAT non è necessaria per l'uso dell'UDP Hole Punching. Il caso in l'uso di questa tecnica è veramente fondamentale è quando il nodo che si desidera contattare è mascherato da NAT e in tale dispositivo non ci sono porte aperte che lasciano passare trasmissioni da peer sconosciuti. Inoltre assumeremo che:

- PEER 2 si sia precedentemente loggato al Server.
- Sia i PEER che il Server quando ricevono un pacchetto sappiano estrarre le informazioni del mittente (IP/Porta).

L'obbiettivo che cercheremo di raggiungere sarà quello di permettere a PEER 1 di inviare datagrammi UDP su una porta aperta precedentemente da PEER 2.
Per capire meglio tale obbiettivo supponiamo che PEER 1 voglia mandare direttamente un pacchetto a PEER 2 senza usare l'UDP Hole Punching.
Esso lo dovrebbe inviare su una "porta X" di PEER 2 che precedentemente dovrebbe creare, nel proprio NAT, un'associazione "porta X" / IP PEER 1 che permetterebbe al pacchetto di passare. Per fare ciò PEER 2 dovrebbe conoscere a priori l'indirizzo pubblico di PEER 1 che userà nella trasmissione e la "porta X" sulla quale esso gli invierà le informazioni. Visto il fatto che PEER 2 non è un veggente né un mago si intuisce subito che questa situazione è molto improbabile. Per questo motivo abbiamo bisogno di una sincronizzazione iniziale tra i due peer che permetta principalmente la creazione di quest'associazione nel NAT di PEER 2.

Di seguito verranno presentati i passi della tecnica UDP Hole Punching riferendoci alla figura 3.7:

1. IL PEER 1 cerca in DHT il Server NAT, responsabile del nodo che vuole contattare (PEER 2), recuperandone le informazioni per contattarlo. Usando queste gli invia al Server una richiesta UDP Hole Punching. (*UDPHPRequestClient*) contenente l'indirizzo IP privato di PEER 2.

Figura 3.8: Pacchetto spedito da PEER 1 al Server NAT.

2. Il Server contattato cerca nella propria cache le informazioni con le quali contattare PEER 2 (IP pubblico, porta di inoltro) e gli inoltra una richiesta (*UDPHPRequestServer*) che contiene l'IP pubblico di PEER 1 e il numero di porta attraverso la quale PEER 1 ha inviato la richiesta, la stessa sulla quale aspetterà la risposta.

Figura 3.9: Pacchetto spedito dal Server NAT a PEER 2

3. PEER 2, ricevuta la richiesta da parte del Server, manda un messaggio vuoto, denominato “FakeRecipient”, a PEER 1 attraverso una porta scelta arbitrariamente. Questo pacchetto servirà per creare l’associazione, nel NAT di PEER 2, porta arbitraria / IP PEER 1 attraverso la quale PEER 1 potrà contattare PEER 2 non risultando più un nodo sconosciuto.

Figura 3.10: Pacchetto “FakeRecipient” spedito da PEER 2 a PEER 1.

4. PEER 2 invia una risposta (*UDPHPResponseClient*) al Server NAT a cui è loggato contenente l’IP pubblico di PEER 1, la porta sulla quale PEER 1 sta aspettando la risposta e la porta arbitraria con la quale si è creata l’associazione precedente.

Figura 3.11: Pacchetto spedito da PEER 2 a Server NAT.

5. Il Server inoltrerà la risposta (*UDPHPResponseServer*) al PEER 1 contenente l’IP pubblico del PEER 2 e il numero di porta che esso ha aperto.

Figura 3.12: Pacchetto spedito da Server NAT a PEER 1.

6. Alla fine della sincronizzazione, se tutto è andato per il verso giusto, PEER 1 riceverà una risposta contenente le informazioni (IP + Port PEER 2) grazie alle quali potrà raggiungere PEER 2 non risultando come nodo sconosciuto.

Bisogna fare attenzione che questa tecnica funziona con molti tipi di NAT fatta eccezione però per i NAT simmetrici (chiamati a volte NAT bi-direzzionali) che tendono ad essere installati nelle grandi reti aziendali. Questi dispositivi hanno una configurazione tale che ad ogni richiesta da uno specifico indirizzo e una specifica porta verso una destinazione, identificata a sua volta da indirizzo e porta, viene associata una connessione dotata di un proprio IP pubblico. In altre parole quando un host interno alla rete privata decide di inviare richieste ad host esterni distinti, o anche su diverse porte dello stesso host esterno, lo fa utilizzando indirizzi IP pubblici distinti.
In questo caso, riferendoci sempre alla figura 3.7 e ipotizzando che il NAT che maschera PEER 1 sia un NAT simmetrico, abbiamo che l'indirizzo IP pubblico del Server è diverso dall'indirizzo IP pubblico del PEER 2 quindi, come appena spiegato, l'indirizzo pubblico utilizzato da PEER 1 per contattare il Server e con il quale PEER 2 fa l'associazione nel proprio NAT, sarà diverso dall'indirizzo pubblico con il quale PEER 1 cercherà di contattare PEER 2 facendolo così risultare un peer sconosciuto.
La tecnica TURN[1] (Traversal Using Relays around NAT) aggira questo problema. Esso utilizza il Server per l'inoltro vero e proprio dei pacchetti che non vengono più scambiati direttamente tra i peer. In altre parole possiamo dire che quando un nodo vuole mandare un pacchetto ad un altro delega il Server a svolgere questo compito. Si capisce subito che la banda disponibile del Server, che ricordo essere un normale nodo PariPari, sarà utilizzata principalmente per svolgere del lavoro da parte di altri peer non lasciandone per le proprie esigenze. Per questo motivo tale tecnica non è stata implementata perché ritenuta troppo pesante e poco efficiente per un normale nodo di PariPari anche se si avrebbe la certezza di raggiungere la destinazione perfino se fossimo mascherati da un NAT simmetrico.

[1] http://en.wikipedia.org/wiki/Traversal_Using_Relay_NAT

3.5 Package

In questo sottocapitolo verrà presentata una panoramica sulle classi e i metodi da me implementati con lo scopo di rendere più comprensibile il funzionamento del codice a tutti i componenti attuali e futuri del gruppo PariPari, che lo vedranno per la prima volta. Bisogna fare attenzione che in certi metodi si userà l'ID di un nodo (identificatore univoco del nodo pubblicato in DHT) al posto del solito indirizzo IP. Questo per rispettare le direttive concordate con il plugin DHT.
Tutte le classi e i relativi metodi qui in seguito esposti si trovano nel package: " **paripari.connectivityNIO.NatTraversal** ". Esso contiene:

- **IPDiscover.java** : implementa le richieste e risposte IP Discover. Contiene:
  - *requestIPDiscover ()* : richiesta di servizio IP Discover. Restituisce il proprio IP pubblico.
  - *responseIPDiscover (DatagramPacket packet)* : risposta IP Discover alla richiesta precedente.
- **NatManagerDHT.java** : contiene i metodi per pubblicare in DHT i peer che si propongono come Server NAT e per cercare tali nodi. Al suo interno troviamo:
  - *storeServiceOnDHT (String service, String[] keys, byte[] info)*: fa lo Store in DHT del nodo che si propone Server NAT pubblicando il servizio offerto, le chiavi di ricerca con le quali essere scovato e le informazioni per raggiungerlo. Ritorna True se tutto è andato a buon fine, False altrimenti.
  - *searchOnDHT (String key, String ID)* : restituisce le informazioni per raggiungere il Server NAT responsabile del nodo identificato da ID. Tale ricerca verrà fatta utilizzando una chiave di ricerca "key", come descritto nei capitoli precedenti.
- **NatTraversal.java** : classe principale che contiene una funzione con la quale inviare dati a un nodo nascosto da NAT. Tale metodo è:
  - *send (DatagramPacket packet, String ID)* : spedisce un pacchetto ad un nodo identificato dall'ID. Ritorna True se tutto è andato al meglio, False altrimenti.
- **ServerAccept.java** : classe per verificare se un nodo può offrire servizi di NAT Traversal. In caso affermativo viene fatto lo store in DHT, al contrario viene fatto il login ad un Server NAT. Contiene:
  - *Login ()* : fa il login di un nodo ad un Server NAT. Ritorna True se il login è stato effettuato con successo, False altrimenti.

- **ServerNAT.java** : classe che implementa il Server NAT. Utilizza i metodi sottolineati fin qui ed in seguito illustrati. Inoltre contiene:
    - *badRequest(DatagramPacket packet)*: risposta ad una richiesta non pertinente.
    - *requestUDPHoleServer(DatagramPacket p):* inoltra una richiesta al peer che il mittente del pacchetto "p" vuole contattare.
      [Punto 2 tecnica UDP Hole Punching, pag. 23]
    - *responseUDPHoleServer(DatagramPacket p)*: inoltra una risposta contenete le informazioni per raggiungere il nodo desiderato.
      [Punto 5 tecnica UDP Hole Punching, pag. 24]

- **STUN.java** : classe che implementa le richieste e risposte STUN. I suoi metodi sono:
    - *requestSTUN ()* : richiesta di servizio STUN. Ritorna il numero di porta sulla quale è attivo un virtual server, altrimenti ritorna -1.
    - *responseSTUN (DatagramPacket p)* : risposta alla richiesta precedente.

- **UDPHolePunching.java**: classe che implementa le richieste e risposte, lato client, della tecnica UDP Hole Punching. I metodi qui contenuti sono :
    - *requestUDPHolePunching (InetAddress IPDest)*: fa una richiesta di UDP Hole Punching richiedendo le informazioni per contattare il peer identificato dall'indirizzo IPDest. Ritorna tali informazioni.
      [Punto 1 tecnica UDP Hole Punching, pag. 23]
    - *responseUDPHolePunching (DatagramPacket p)*: metodo che implementa sia la risposta del destinatario verso il Server sia la creazione dell'associazione nel proprio NAT.
      [Punti 3 e 4 tecnica UDP Hole Punching, pag. 24]

Conclusioni

Oggigiorno i NAT sono dispositivi largamente utilizzati che risolvono in gran parte la necessità di risparmiare indirizzi IP. Di conseguenza tecniche come il NAT Traversal sono diventate fondamentali per il funzionamento di qualsiasi rete peer-to-peer. Con l'introduzione dell'IPv6 tali tecniche diventeranno obsolete ma come la storia ci insegna tutte le tecniche di risparmio, che siano metodologie per risparmiare memoria, clock della CPU (Central Processing Unit) e quant' altro, prima o poi diventeranno di nuovo utili, forse tra un milione di anni ma non si sa mai. Credo che coloro che hanno progettato l'IPv4 non si aspettassero di certo un uso cosi eccessivo di tali indirizzi.
Attualmente il NAT Traversal funziona abbastanza bene aggirando i normali firewall e router domestici. In futuro si dovrà interfacciare con DHT in base ai cambiamenti concordati. In particolare le modifiche che verranno attuate saranno: all'avvio di DHT in un nodo, questo avrà uno stato interno NAT impostato su Forse. Tale stato identifica la presenza o meno di un mascheramento NAT e potrà assumere globalmente tre valori: True, False, Forse. Il nodo rimarrà in stato Forse finché connectivityNIO, attraverso l'IP Discover, non gli comunicherà l'effettivo valore di tale variabile. Un nodo potrà inoltre cambiare da Forse a False il proprio stato nel caso riceva un pacchetto nella InPort di DHT, porta di ingresso di DHT (attualmente di default è la 5335), questo per il fatto che non sarà mascherato da NAT avendo ricevuto un pacchetto da un peer sconosciuto. Nel caso di passaggio di stato interno da Forse a False, il nodo dovrà rieffettuare nuovamente la pubblicazione in DHT. Inoltre nella comunicazione con gli altri nodi, verrà scambiato, oltre alla solita tripletta (ID, IP, Port) pure un campo di un bit (NAT) contenente lo stato attuale del nodo. Esso sarà impostato a True, se lo stato interno del nodo sarà True o Forse, a False in caso contrario. Forniti questi cambiamenti, all'avvio di ConnectivityNIO in un nodo, questo dovrà connettersi a DHT, ricercare dei server NAT e identificare il proprio stato NAT. Nel caso questo stato risulti False e l'utente abbia dato il consenso nella configurazione, verrà fatto lo store su DHT del servizio NAT Traversal. Bisognerà trovare un sistema per premiare i nodi che vorranno essere eletti Server NAT, probabilmente dandogli dei privilegi o un tot di crediti. Nel caso lo stato NAT risulti True verrà fatto lo store su DHT di una stringa con chiave ID del nodo e, memorizzati nei 12 byte allegati, le informazioni per raggiungerlo. Inoltre verrà fatto un Login ad un Server NAT.

In futuro i plugin e i peer useranno l'ID come parametro di comunicazione non più l'IP. Per questo quando invocheranno una richiesta send (pacchetto, ID) connectivityNIO, tramite la tecnica NAT Traversal se necessario, farà una ricerca di tale ID (come stringa) su DHT, otterrà le informazioni per contattare tale nodo e poi effettuerà il contatto vero e proprio.
Apportate queste modifiche si dovrà finire di testare il tutto, magari utilizzando o modificando i test da me fatti per la versione attuale.
Come l'esperienza ci insegna solo un uso prolungato di tale tecnica farà insorgere eventuali bug non tenuti in considerazione.
Lavorare nel progetto PariPari è stata un'ottima esperienza sia per capire ed imparare come funziona un progetto importante sviluppato da molte persone, per acquisire nuove tecniche e strategie di programmazione, sia per far pratica nel campo della progettazione e sviluppo dei software, un aspetto secondo me fondamentale, che questo corso di laurea, essendo molto teorico e poco pratico, non tiene molto in evidenza. Come ultimo parere personale tengo a dire che ci vorrebbero più progetti di questo tipo, anche in corsi di laurea diversi sopprimendo magari certi corsi puramente teorici i quali forniscono conoscenze che, la maggior parte dei casi, non verranno mai utilizzate perché oggigiorno in disuso. L'esperienza insegna più di quanto farebbe un libro o un corso tenuto male.

Bibliografia

[1] Paolo Bertasi. *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*. Dipartimento di Ingegneria dell'Informazione, Università di Padova, 2004.

[2] Larry L. Peterson, Bruce S. Davie. *Reti di Calcolatori – Terza Edizione*. Apogeo, Italia, Milano, 2004.

[3] A. S. Tanenbaum. *Reti di Calcolatori - Quarta Edizione*. Pearson Education Italia, Milano, 2003.

[4] Giacomo Portolan. *PARIPARI – NAT TRAVERSAL*. Dipartimento di Ingegneria dell'Informazione, Università di Padova, A.A. 2008/2009.

[5] PariPari mediawiki. *Connectivity NIO: http://www.paripari.it/mediawiki/index.php/ConnectivityNIO_en*

[6] SUN Javadoc. *package java.nio*: *http://java.sun.com/j2se/1.4.2/docs/api/java/nio/package-summary.html*

[7] J. Rosenberg, R. Mahy, P. Matthews, D. Wing. *RFC 5389: Session Traversal Utilities for NAT (STUN)*, Ottobre 2008: *http://datatracker.ietf.org/doc/rfc5389*

[8] Wikipedia. *UDP Hole Punching*: *http://en.wikipedia.org/wiki/UDP_hole_punching*

Elenco figure

1.1 Architettura modulare di PariPari 4

2.1 Assegnazione NAP 10

2.2 Gestione buffer/code in modalità bloccante e non-bloccante 12

3.1 Pacchetto di richiesta/risposta generica 14

3.2 Richiesta/Risposta IP Discover 15

3.3 Possibili mascheramenti: (a) comunicazione libera, (b) presenza di un firewall,(c) presenza di un NAT, (d) presenza di NAT + firewall, (e) caso peggiore non si riceve nulla 16

3.4 Richiesta/Risposta STUN 17

3.5 Casi STUN: (a) Virtual Server attivo, (b) normale mascheramento NAT ... 18

3.6 Richiesta di Login 21

3.7 Sincronizzazione UDP Hole Punching 22

3.8 Pacchetto spedito da PEER 1 a Server NAT 23

3.9 Pacchetto spedito da Server NAT a PEER 2 23

3.10 Pacchetto “FakeRecipient” spedito da PEER 2 a PEER 1 24

3.11 Pacchetto spedito da PEER 2 a Server NAT 24

3.12 Pacchetto spedito da Server NAT a PEER 1 24