Università degli Studi di Padova

Dipartimento di Ingegneria
dell'Informazione
Corso di Laurea in Ingegneria Informatica

PariConnectivity: Tunnel

Laureando: Nicola Paiero

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

Correlatore: Dott. Francesco Peruch

Anno accademico 2011/2012

Indice

Sommario

PariPari è un progetto software sviluppato in Java internamente al Dipartimento di Elettronica e Informatica dell'Università di Padova. Lo scopo di questo progetto è creare una rete peer-to-peer serverless che sia multifunzionale, anonima, che usi un sistema di crediti e che sia accessibile anche da reti esterne. La struttura di PariPari è completamente modulare, tra i vari plugin che la compongono c'è ConnectivityNIO il quale si occupa di fornire la connettività Internet a tutta la piattaforma.
Tuttavia, per un software che vive quasi interamente con lo scambio di informazioni via rete, è facile che il numero di socket, necessari ai vari servizi disponibili, cresca molto velocemente e per questo ha reso necessaria l'introduzione di un nuovo componente interno a ConnectivityNIO: il Tunnel. Il nuovo modulo permette di comunicare con host remoti attraverso un unico socket Java permettendo di limitare il numero di socket "reali" necessari al funzionamento corretto di PariPari.
Nel corso del Capitolo 1 il lettore verrà introdotto al progetto PariPari in modo da poter identificare efficacemente in quale contesto si va ad introdurre il Tunnel.
Il Capitolo 2 verte sull'analisi dei requisti e sulla specifica dei protocolli necessari al funzionamento del Tunnel, mentre la descrizione delle fasi di progetto e della realizzazione delle componenti software è affrontata nel Capitolo 3.
Nel Capitolo 4 vengono proposti alcuni possibili sviluppi sia sotto il profilo della stabilità del Tunnel sia come funzionalità aggiuntive non ancora presenti alla stesura di questo documento.

Capitolo 1

Introduzione

Il Capitolo 1 è di tipo introduttivo ed ha lo scopo di contestualizzare l'argomento trattato da questo documento: il Tunnel di ConnectivityNIO. Nella sezione 1.1 è proposta una panoramica sul progetto PariPari sia in termini di funzionamento delle sue componenti software principali sia in termini organizzativi e metodologici riguardanti il progetto.
Nella sezione 1.2 viene approfondita la trattazione del modulo ConnectivityNIO, modulo di appartenenza del Tunnel.

1.1 PariPari

1.1.1 Cos'è PariPari?

PariPari[8] è un progetto software sviluppato in Java internamente al Dipartimento di Elettronica e Informatica dell'Università di Padova. Lo scopo di questo progetto è creare una rete peer-to-peer [27] serverless che sia multifunzionale, anonima, che usi un sistema di crediti e che sia accessibile anche da reti esterne.
Nell'immaginario collettivo si identifica con rete peer-to-peer (p2p) un sistema di filesharing, ovvero una rete per lo scambio di file tra utenti. Tuttavia il concetto di rete peer-to-peer è più generale, in prima istanza, si può pensare semplicemente come una tipologia di rete antitetica a quelle di tipo client-

server. Nelle reti p2p infatti non esiste il concetto di gerarchia, tutti i nodi della rete hanno la stessa importanza (si parla di rete paritaria) e possono, in base alla circostanza, fungere da nodi client o nodi server comunicando direttamente tra loro.

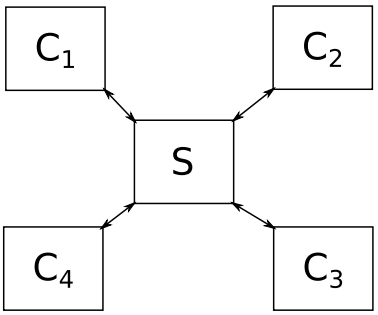

Figura 1.1: Sistema client-server. C_i sono client, S è il server.

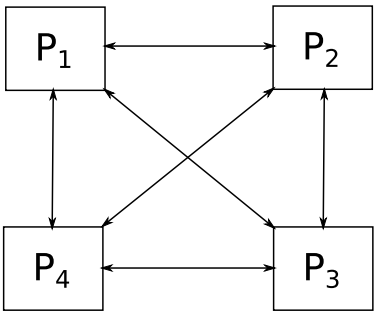

Figura 1.2: Sistema peer-to-peer. P_i sono dei peer.

PariPari fornisce i tipici servizi di filesharing come l'accesso alle reti BitTorrent [11] ed eDonkey [25], ma si prefigge l'obbietivo di portare nel dominio del peer-to-peer anche servizi che storicamente si basano su una struttura con server accentrato come: WebServer[28], MailServer [12], DnsServer [15], Irc [17] ed in generale qualsiasi servizio realizzabile con le API[26] messe a disposizione da PariPari; uno dei punti di forza del progetto PariPari è il fatto che, oltre a fornire dei servizi preconfezionati pronti all'uso, metta a disposizione un vero e proprio framework il quale permetterà a programmatori esterni al progetto di sviluppare le proprie applicazioni utilizzando la rete PariPari.

1.1.2 Plugins

PariPari ha una struttura interamente modulare; infatti ogni funzionalità fornita dal software è racchiusa dentro un contenitore detto plugin, questa architettura consente di attivare solamente i servizi realmente necessari a

perseguire lo scopo dell'utente finale.
Tuttavia è necessario fornire una gerarchia tra i plugin sviluppati all'interno del progetto PariPari, per questo sono suddivisi in cerchie: interna ed esterna.

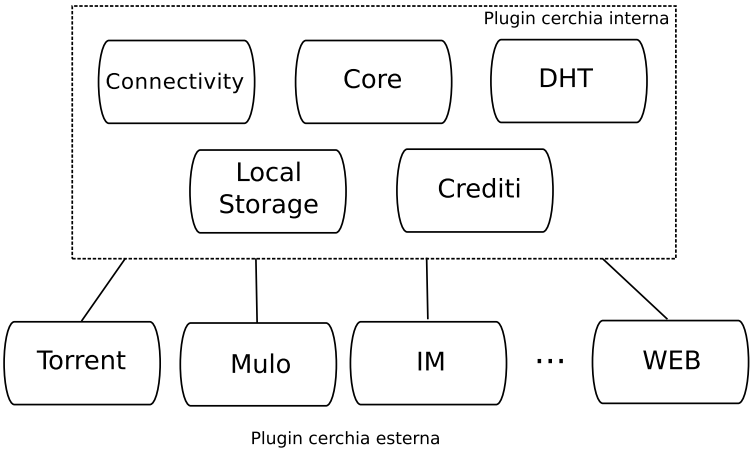

Figura 1.3: Suddivisone dei plugin di PariPari.

La cerchia interna forma un gruppo di moduli indispensabile a PariPari, i quali forniscono tutte le funzionalità di base da cui partire per lo sviluppo di nuovi plugin della cerchia esterna. I plugin interni sono i seguenti:

- Core: è il kernel di PariPari, si occupa di coordinare i plugin e della sicurezza.
- ConnectivityNIO: fornisce la connettività alla rete Internet.
- LocalStorage: fornisce l'accesso alla memoria secondaria (file, directory).
- PariDHT: mantiene funzionante la rete DHT [20] usata da PariPari.

Data la loro importanza i plugin della cerchia interna sono e saranno sviluppati solo da programmatori interni al progetto PariPari.
Tutti i plugin devono rispettare alcuni vincoli progettuali, i quali concretamente si realizzano mediante l'implementazione di interfacce comuni a tutti i

plugin. Queste interfacce sono fondamentali poichè formalizzano i metodi di base per l'uso dei plugin e si occupano di realizzare i meccanismi attraverso i quali i plugin forniscono e richiedono servizi.
Così come il plugin, anche i servizi da esso offerti devono rispettare dei vincoli, infatti devono essere offerti in forma di Application Programming Interface. Un API è una classe astratta usata come prototipo, in questa classe viene formalizzato il modo di utilizzare un dato servizio; ciò permette di scindere la definizione del servizio (API) dalla sua realizzazione. Ad esempio, una volta definita la classe astratta `ServizioAPI` questa potrà essere estesa e quindi realizzata da diversi plugin, le realizzazioni potranno essere molto diverse ma tutte forniranno il servizio specificato da `ServizioAPI`.

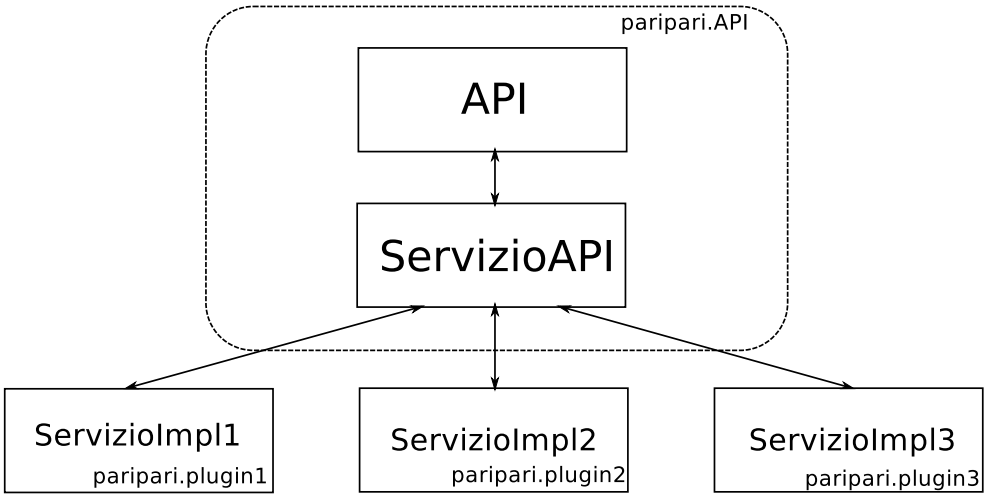

Figura 1.4: Esempio di gerarchia di un API. `ServizioAPI` dispone di diverse realizzazioni fornite da plugin differenti.

Ogni plugin possiede all'interno del proprio archivio Jar il file descriptor.xml dove sono elencati sotto forma di API i servizi forniti e quelli necessari al funzionamento del plugin stesso (dipendenze).

1.1.3 Sistema di crediti

PariPari possiede un sistema di crediti che regola lo scambio di risorse e servizi.
Il plugin Crediti entra in azione in due casi di scambio di servizi: tra peer della rete PariPari e tra i plugin di un peer. Il primo caso è comune alla gran parte dei sistemi di filesharing odierni, lo scopo principale è quello di garantire equità tra i nodi favorendo i peer votati allo scambio di risorse ed emarginando quelli i quali tendono a consumare risorse senza offrire nulla in cambio.
Anche lo scambio di servizi interno ad un istanza di PariPari è regolato dal sistema di crediti. Ogni servizio ha un costo ed un plugin che voglia usufruirne dovrà pagare un certo numero di crediti.
I crediti possono essere ottenuti in due modi. La prima modalità consiste nell'attendere un certo intervallo di tempo dopo il quale vengono distribuiti a tutti i plugin. Il secondo modo che un plugin ha per ottenere crediti è quello di offrire a sua volta dei servizi in cambio di crediti.
La distribuzione dei crediti ed il costo dei servizi sarà completamente configurabile dall'utente, permettendo di favorire i plugin ritenuti più utili e, nel caso un servizio sia offerto da diversi plugin, di scegliere quello più conveniente.

1.1.4 Il Core

Il Core [3] è il cuore di PariPari, l'importanza di questo plugin è cresciuta negli anni fino al rilascio della versione T.A.L.P.A. (The Acronym for Lightweight Plug-in Architecture).
Gli scopi del Core sono molteplici e saranno descritti in breve nelle prossime sezioni.

Caricamento PariPari

Il software PariPari non è pensato per essere avviato direttamente e sfrutta Java Web Start (JWS) [19]. JWS permette il download e l'avvio di software

scritto in Java direttamente da un browser mediante il Java Network Launching Protocol (JNLP).
La prima operazione svolta dal file JNLP è l'avvio del Core di PariPari, il quale provvederà a caricare tutti i plugin richiesti dall'utente, gestendo in automatico la risoluzione delle dipendenze dei plugin. Ciò permette all'utente di avviare PariPari con un solo click dal browser, con l'unico vincolo di avere una Java Virtual Machine installata sulla macchina.

Sicurezza

Data la natura di Java Web Start, la politica predefinita per la sicurezza prevede la conferma da parte dell'utente per ogni operazione potenzialmente rischiosa, come ad esempio l'accesso al disco (file, cartelle), l'accesso alla rete e la gestione dei thread attivi.
Ciò non è accettabile per un software come PariPari e per questo motivo il `SecurityManager` predefinito di Java è stato sostituito da `PariPari SecurityManager`. La scrittura del gestore della sicurezza ha comportato la risoluzione di due problemi. Il primo problema è relativo alla fornitura dei servizi di accesso alle risorse della macchina (memoria e rete) ed è stato risolto con la creazione dei plugin della cerchia interna. Connectivity e Storage sono gli unici plugin considerati sicuri e quindi hanno l'accesso alle risorse della macchina, per questo motivo lo sviluppo dei plugin della cerchia interna è e sarà sempre affidato a programmatori interni al progetto.
La seconda problematica, data dalla sostituzione del `SecurityManager` predefinito, è dovuta alla necessità di controllare chi e con quali modalità accede ai servizi forniti dai plugin della cerchia interna, per questo motivo il Core si pone come unico mezzo di comunicazione tra i plugin.

Comunicazione tra plugin

Ogni plugin di PariPari vive in un contesto limitato al proprio file Jar di appartenenza, infatti gli è impedito di accedere direttamente alle classi degli altri plugin.

Da qui nasce l'esigenza di un sistema per lo scambio di servizi all'interno di un'istanza di PariPari, questa funzione è svolta dal Core.
Grazie alla standardizzazione, tutti i plugin dispongono di strutture dati particolari che permettono lo scambio di richieste e risposte con il Core. I messaggi in uscita sono posti in un oggetto di tipo `IMoundPutter` mediante la chiamata al metodo `askTheCore` ereditato da `IPlugin`. Per quanto riguarda i messaggi in ingresso al plugin è necessaria una coppia di classi Java: un'istanza di `IPrivateMound` dove depositare i messaggi ed un oggetto di tipo `IListener`, il quale si occupa di controllare la presenza o meno di messaggi in ingresso.
Le richieste vengono gestite dal Core in maniera analoga: un oggetto di tipo `IKernel` serve le richieste accodate in un oggetto di tipo `IMound`. Un'istanza di tipo `IStoreKeeper` provvede ad inoltrare le richieste al rispettivo `IPrivateMound` di destinazione. Infine un oggetto di tipo `IWarehouse` mantiene una lista di tutte le richieste inoltrate e si occupa di risvegliare i plugin in attesa di una risposta.

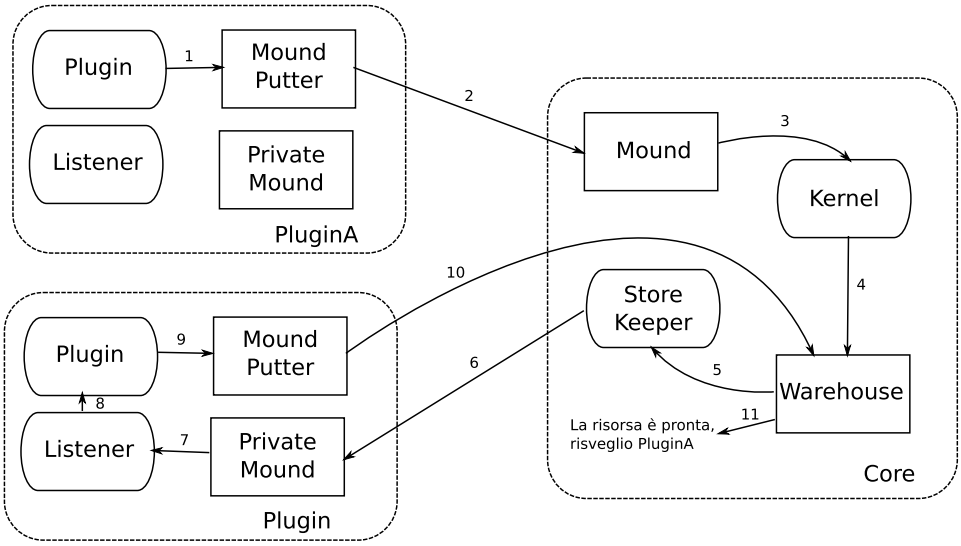

Figura 1.5: Esempio di routing di una richiesta. PluginA richiede una risorsa a PluginB mediante il Core.

1.1.5 Rete PariPari

La rete PariPari si basa su una tabella di hash distribuita (DHT) ed è realizzata dal plugin PariDHT. Le DHT forniscono dei servizi simili a quelli delle tabelle di hash tradizionali operando però in un contesto distribuito, in particolare presentano le seguenti proprietà:

- Scalabilità: la struttura della DHT permette un funzionamento efficiente anche quando si è in presenza di un numero molto elevato di nodi.
- Tolleranza ai guasti: la struttura della DHT mantiene il corretto funzionamento della rete indipendentemente dalle dinamiche con cui i nodi della rete si connettono (nuovi nodi, nodi con disconnessioni frequenti).
- Decentralizzazione: la DHT per operare non ha bisogno di nodi server, la struttura dati distribuita è mantenuta funzionante dai nodi che la compongono.

Il primo passo nella realizzazione di una DHT è l'identificazione di uno spazio di indirizzamento e di una funzione distanza tra gli indirizzi.
Nel caso di PariPari, il quale utilizza una struttura basata su Kademlia [14], come indirizzi usa stringhe di 256 bit e la distanza tra gli indirizzi è calcolata con la metrica XOR, cioè calcolando l'OR esclusivo tra i due indirizzi.
All'avvio, ad ogni nodo PariPari viene assegnato un indirizzo univoco appartenente allo spazio di indirizzamento detto ID.
Ora verranno analizzate più in dettaglio le due operazione fondamentali di una DHT: *store* e *search*.

Store

Si assuma che si voglia inserire nella tabella di hash distribuita una risorsa r identificata dalla chiave k (generalmente una stringa di testo). Con una funzione v di hash viene calcolato $k' = v(k)$ con k' appartenente allo spazio di indirizzamento.

Dopo questa operazione avviene l'inserimento della risorsa all'interno della tabella di hash, simile ad una chiamata di tipo $put(k', r)$. La risorsa r è inoltrata tra i nodi della rete fino a raggiungere un nodo $n_{k'}$ il cui indirizzo ha distanza minima da k'.

Search

Nel caso si voglia cercare nella tabella di hash distribuita una risorsa r associata alla chiave k, basterà calcolare $k' = v(k)$ ed effettuare una chiamata simile a $search(k')$.
La richiesta verrà inoltrata tra i nodi della rete fino a raggiungere il nodo $n_{k'}$ a cui è stata assegnata la risorsa r, il quale provvederà a rispondere al nodo il quale ha originato la richiesta.

Routing in DHT

Uno dei punti di forza di DHT è l'efficienza delle operazioni store e search, infatti ogni nodo della rete ne può contattare un altro in $O(logN)$ passaggi, dove N è il numero di nodi attivi presenti nella rete DHT.
Le buone prestazioni della tabella di hash distribuita sono ottenute usando un algoritmo greedy per instradare i messaggi tra i nodi; ad ogni passo dell'algoritmo, sfruttando la funzione distanza, il messaggio viene inoltrato al nodo nelle vicinanze con distanza minima, fino a raggiungere il nodo di destinazione.
Ciò comporta che ogni nodo della rete debba mantenere un insieme di contatti in modo da permettere l'instradamento usando l'algoritmo descritto in precedenza. Assumendo che gli indirizzi della rete siano di m bit (nel caso di PariPari $m = 256$), ogni nodo della rete deve mantenere una tabella di routing con m k-bucket, dove un k-bucket è una lista di k indirizzi (Indirizzo IP, numero di porta, ID DHT).
La tabella di routing è mantenuta seguendo delle regole ben precise; per decidere in quale k-bucket può essere inserito un nodo, viene confrontato l'indirizzo del nodo locale con quello del nodo candidato ad entrare nella ta-

bella di routing. Ad esempio un nodo può essere inserito nel k-bucket i-esimo se e solo condivide con l'indirizzo locale i primi $i-1$ bit e differisce per il bit i-esimo.
Si identificano quindi m distanze ed il numero dei nodi candidati ad entrare in un k-bucket diminuisce all'aumentare dell'indice della lista: in prima posizione potranno essere inseriti 1/2 dei nodi indirizzabili, in seconda posizione 1/4 dei nodi indirizzabili, in terza 1/8 e così via. Per cui quando un nodo deve instradare un messaggio nella rete DHT, preleva dalla tabella di routing l'indirizzo di un nodo con distanza minima. In linea teorica la rete manterrebbe le medesime prestazioni anche con un k-bucket con $k = 1$ tuttavia, data la natura volatile di alcuni nodi, si utilizza $5 \leq k \leq 20$ per garantire una maggiore robustezza della rete nel caso della disconnessione dei nodi. Nell'eventualità si abbia un numero superiore a 20 nodi candidati ad un k-bucket la scelta su quali nodi inserire è di tipo euristico, infatti generalmente i nodi con un uptime maggiore è più probabile che rimangano connessi in futuro e quindi sono dei candidati migliori all'inserimento nella tabella di routing.

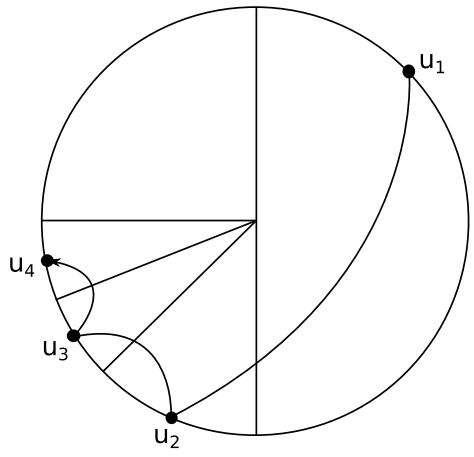

Figura 1.6: Esempio semplificato di routing su DHT. Sulla circonferenza sono posti gli indirizzi dei peer u_i, ad ogni salto la distanza tra l'indirizzo del nodo locale e l'indirizzo di destinazione diminuisce.

1.1.6 Metodologie e strumenti di sviluppo

PariPari è un progetto in continua espansione ed è per questo che lo sviluppo del software deve essere gestito al meglio sotto il profilo organizzativo, metodologico e tecnico.
Le metodologie adottate sono molto importanti sia per la qualità di PariPari sia per la formazione degli studenti partecipanti al progetto. Uno degli scopi di PariPari è proprio quello di introdurre gli studenti a strumenti e metodi usati in ambito professionale, i quali spesso sono totalmente ignorati dagli insegnamenti proposti dall'ateneo.

Suddivisione in team

I programmatori di PariPari sono studenti provenienti da differenti corsi di laurea dell'area dell'ingegneria dell'informazione; vengono arruolati nel progetto tramite delle lezioni speciali introduttive a PariPari ed in seguito vengono distribuiti in vari team in base alle capacità ed agli interessi personali. Un team è un gruppo di studenti, i quali lavorano ad un singolo modulo funzionale di PariPari, generalmente un plugin; ad uno dei membri del gruppo è affidato il titolo di Team Leader con funzioni organizzative sul gruppo stesso, dato che, entro certi limiti, lo sviluppo di una singola componente di PariPari è gestito in autonomia all'interno del team.
Periodicamente i team leader si ritrovano assieme ai responsabili di PariPari per discutere delle scelte tecnico/organizzative per gli sviluppi futuri del progetto.

Metodologie di sviluppo

A livello metodologico, il progetto PariPari segue i principi di Extreme Programming (XP) [2] per lo sviluppo del software. E' un tipo di metodologia agile di sviluppo del software e si contrappone a metodologie pesanti, come il "classico" metodo a cascata, sotto diversi punti di vista.
Nell'Extreme Programming uno degli aspetti importanti è la comunicazione

con il cliente, inteso come utilizzatore finale del software. Questo fatto porta ad analizzare a fondo i requisiti del software, posti in termini di funzionalità e modi d'uso.
L'importanza della fase di analisi dei requisiti è evidente in una delle cosiddette pratiche di XP: il Test Driven Development. Il Test Driven Development (TDD) è un processo di sviluppo del software, il quale pone al primo posto l'analisi dei requisiti e la verifica in corso d'opera dell'aderenza ai requisiti stessi.
Il concetto più importante di TDD, è quello di Unit Testing. La prima fase dello sviluppo del software è la formulazione dei requisiti in termini di test, ovvero la scrittura di una porzione di codice con il compito di collaudare una specifica parte del codice di produzione. Una volta scritto il test si può iniziare la scrittura del codice di produzione interessato dal test, verificando periodicamente il risultato del testing.

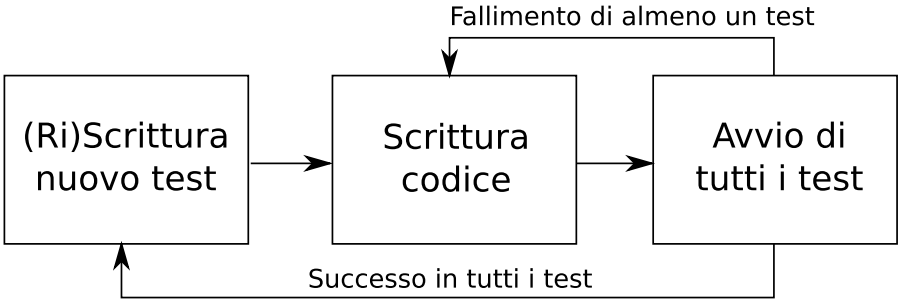

Figura 1.7: Esempio di Test Driven Development.

Il Test Driven Development porta numerosi vantaggi: entro certi limiti, non serve una fase di debugging e lo sviluppo procede in piccoli cicli di produzione ben definiti, in contrapposizione con il lungo e oneroso ciclo di produzione del metodo a cascata. La scrittura degli Unit Test prima del codice di produzione porta alla scrittura di un codice più modulare e più flessibile nel caso di modifiche strutturali al progetto.

Strumenti di sviluppo

Per sfruttare al meglio tutti gli accorgimenti metodologici e organizzativi sino a qui illustrati, è necessario un insieme di strumenti software.
Lo sviluppo del software viene svolto con Eclipse [7]. Eclipse è un Integrated Development Environment (IDE) open source prodotto da Eclipse Foundation. L'uso di un IDE per lo sviluppo del software porta numerosi benefici, poichè molti strumenti generalmente separati (editor, compilatore, debugger, suit di testing) sono racchiusi in un unico software aumentando la produttività. Ad esempio, la fase di testing illustrata nella sezione precedente, in Eclipse risulta completamente automatica e permette in pochi secondi di avere un quadro completo del successo o meno di tutti i test scritti.
Uno dei grossi problemi da affrontare nello sviluppo di software in squadra è la gestione del codice. Per questo è fondamentale l'utilizzo di un sistema di controllo della versione, il quale permette di memorizzare in remoto (un server) diverse versioni di un file e quindi ha anche la funzionalità di luogo di archiviazione del codice. In PariPari si è scelto di utilizzare Subversion (SVN) [6] per la sua larga diffusione e per la sua completa integrazione con Eclipse.
Recentemente si è aggiunto un nuovo strumento per lo sviluppo di PariPari: Redmine [1]. Redmine è uno strumento open source web-based per la gestione di progetti. Le sue funzionalità sono molteplici, le più importanti sono:

- scheduling del lavoro (release, feature).
- bug-tracker.
- collaborative editing (wiki).
- integrazione con SVN.

Tutte le funzionalità di Redmine possono essere integrate in Eclipse e ciò permette di avere un ambiente di sviluppo veramente integrato ed efficiente.

1.2 ConnectivityNIO

1.2.1 Cos'è ConnectivityNIO

ConnectivityNIO [21] è un plugin della cerchia interna, si occupa di fornire la connettività ad internet a tutti i moduli di PariPari. É l'unico plugin a cui il `PariPariSecurityManager` consente di accedere alla rete mediante le API di Java. Inoltre, l'accentramento dei servizi di connettività permette di gestire al meglio la "risorsa rete" sotto moltiplici aspetti quali l'allocazione della banda disponibile ed il numero di connessioni attive.
Sebbene le funzionalità ad alto livello fornite siano pressochè le medesime di quelle presenti nei primi rilasci del plugin, nell'ultimo periodo il codice ha subito una forte reingegnerizzazione per passare dall'uso delle liberie del package `java.net` al package `java.nio` [9] [18]. NIO sta per "New I/O", le librerie sono presenti in Java dalla versione 1.4 e sono pensate appositamente per l'input/output ad alte prestazioni, infatti permettono di accedere direttamente ad operazioni a basso livello fornite dal sistema operativo in uso.
Oltre al miglioramento prestazionale, Java NIO introduce in Connectivity la possibilità di avere scritture e letture di rete non bloccanti. Ciò comporta un vantaggio notevole, infatti è possibile delegare ad un solo thread la gestione delle operazioni di I/O di molti socket, riducendo al minimo le risorse sprecate dalla gestione di un elevato numero di thread, come nel caso si usi `java.net` dove generalmente si utilizza un thread per ogni socket attivo.
I dati da scrivere sulla rete vengono scritti momentaneamente in una struttura dati temporanea e viene segnalata la loro presenza ad un oggetto chiamato `Selector` con l'operazione `register`. A questo punto, non appena possibile, il `Selector` provvederà a svuotare il buffer, completando così la scrittura asincrona. In modo del tutto analogo avviene la lettura dalla rete.

API

Come tutti i plugin di PariPari anche ConnectivityNIO fornisce i propri servizi sotto forma di API. Gli oggetti restituiti sono molto simili ai socket di

Java e sono disponibili in versione bloccante e non bloccante.

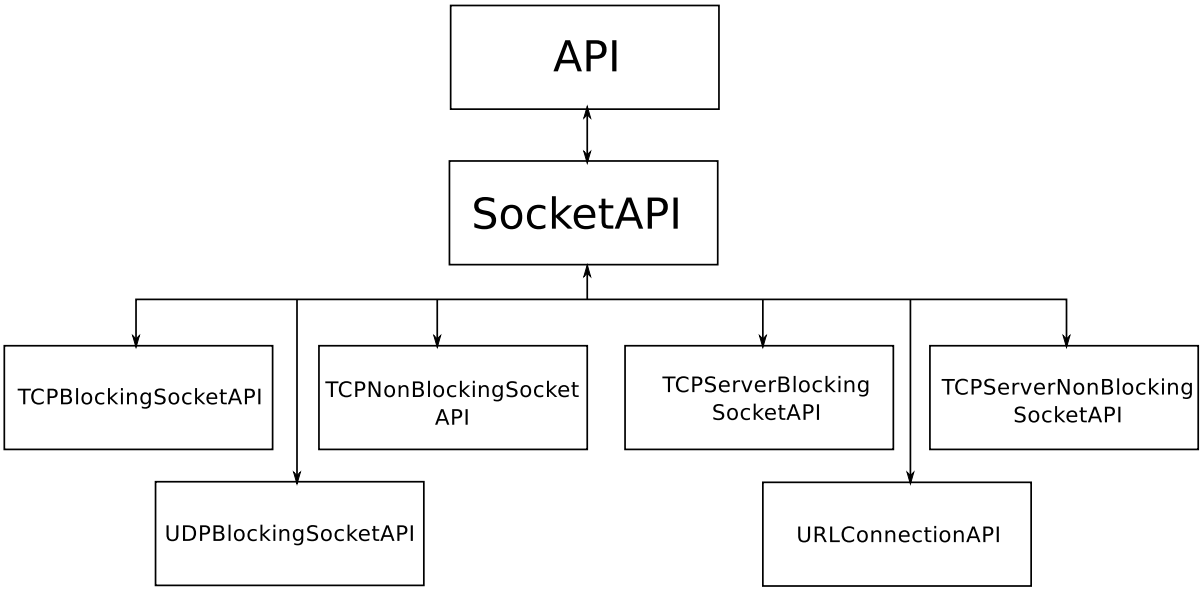

Figura 1.8: Gerarchia delle API fornite da ConnectivityNIO.

- `SocketAPI`: quest'interfaccia viene implementata da tutti i API di ConnectivityNIO, contiene metodi di utilità condivisi da tutti i tipi di socket oltre ai metodi ereditati da `API` che estende.

- `TCPBlockingSocketAPI`: definisce un socket TCP [23] di tipo bloccante, ovvero i metodi bloccano il thread che li ha chiamati fino a che l'operazione non è conclusa. Il socket permette la lettura e la scrittura tramite array di byte o via Stream.

- `TCPNonBlockingSocketAPI`: le funzionalità sono le medesime di `TCP BlockingSocketAPI`, la differenza è nel tipo di chiamate, in questo caso non bloccanti. Quando viene effettuata una chiamata è necessario specificare un'istanza di `PluginNotification`, il quale ha la duplice funzione di notificare il successo o meno dell'operazione di I/O oppure di eseguire direttamente del codice specificato dall'utente programmatore.

- `TCPServerBlockingSocketAPI`: definisce un server socket TCP, il funzionamento è simile a quello di `ServerSocket` di `java.net` e serve ad accettare le connessioni TCP in ingresso. Le chiamate sono di tipo non bloccante.

- `TCPServerNonBlockingSocketAPI`: le funzionalità sono le medesime di `TCPServerBlockingSocketAPI`, ma le chiamate funzionano in maniera non bloccante. Come in `TCPNonBlockingSocketAPI` è necessario l'uso di un oggetto `PluginNotification` per effettuare le chiamate non bloccanti.

- `UDPBlockingSocketAPI`: definisce un socket UDP [24], ed il suo uso è del tutto simile a quello di `DatagramSocket` di `java.net`. La modalità di funzionamento predefinita è di tipo non bloccante, ma può essere configurata runtime in non bloccante o in modalità ibrida (si blocca per un determinato intervallo di tempo).

- `URLConnectionAPI`: questa classe astratta definisce un oggetto per operare su un una risorsa specificata come Uniform Resource Locator (URL). Il suo uso è simile a quello di `URLConnection` di `java.net`.

1.2.2 NAT Traversal

NAT Traversal [21] è un modulo di ConnectivityNIO adibito all'attraversamento dei NAT [4], ha il compito di lavorare assieme al modulo tunneling per fornire una connettività quanto più completa senza la necessità di configurare i dispositivi NAT presenti sul percorso di rete.

Cos'è un NAT

NAT sta per Network Address Translation ed è una tecnica che permette di modificare i campi indirizzo del protocollo IP, nel caso si modifichi l'indirizzo sorgente si parla di Source-NAT, nel caso si modifichi l'indirizzo di destinazione si è di fronte a Destination-NAT. I NAT sono generalmente realizzati

nel firmware dei router e sono sempre più importanti poichè ormai sono presenti in gran parte delle reti domestiche, poichè permettono alle macchine di una rete locale di condividere un unico IP pubblico con cui accedere ad Internet.
In realtà in gran parte dei casi non viene effettuata la sola traduzione degli indirizzi IP ma vengono modificati anche i campi numero di porta a livello di trasporto nei protocolli TCP e UDP. In questo caso si parla di NAPT Network Address and Port Translation.

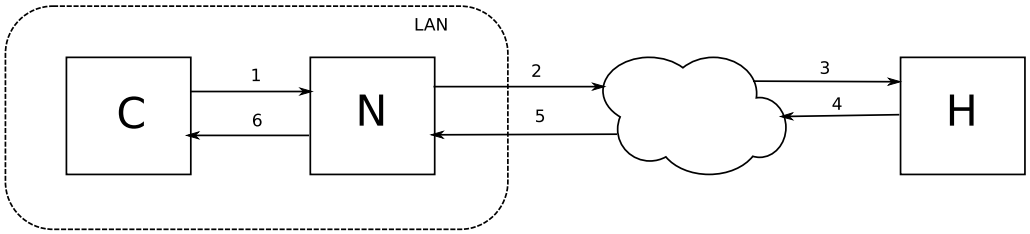

Figura 1.9: Esempio di rete con NAT. N è un NAT, C è un client connesso tramite NAT, H è un host qualsiasi. Per il dettaglio sulla traduzione degli indirizzi si rimanda alla Tabella 1.1

Tratta	Src	Dst
1	$addr_i : port_i$	$addr_h : port_h$
2/3	$addr_e : port_e$	$addr_h : port_h$
4/5	$addr_h : port_h$	$addr_e : port_e$
5	$addr_h : port_h$	$addr_i : port_i$

Tabella 1.1: Traduzione degli indirizzi con riferimento alla Figura 1.9. $addr_i : port_i$ sono indirizzo e numero di porta locali usati da C, $addr_e : port_e$ sono indirizzo e numero di porta pubblici tradotti dal NAT N, $addr_h : port_h$ sono indirizzo e numero di porta di un host esterno alla LAN H.

Tipi di NAT

La comprensione del modo in cui un NAT traduce i numeri di porta e gli indirizzi è fondamentale per la formulazione di una tecnica per l'attraversamento, per questo è possibile identificare dei tipi di NAT; siano $addr_i : port_i$

l'indirizzo ed il numero di porta di un host interno alla rete locale dietro NAT, siano $addr_e : port_e$ indirizzo e numero di porta con cui il NAT traduce le comunicazioni uscenti dalla LAN, siano $addr_h : port_h$ indirizzo e numero di porta di un host esterno con cui si voglia comunicare.

- Full-cone NAT: $addr_i : port_i$ vengono associati dal NAT ad $addr_e : port_e$ in maniera biunivoca, chiunque può contattare $addr_i : port_i$ contattando $addr_e : port_e$.

- Restricted cone NAT: $addr_i : port_i$ vengono associati dal NAT ad $addr_e : port_e$ in maniera biunivoca, qualsiasi host $addr_h : x$ può contattare $addr_i : port_i$ attraverso $addr_e : port_e$ se e solo se $addr_i : port_i$ ha precedentemente inviato un pacchetto a $addr_h$. x sta a significare che il numero di porta non è significativo.

- Port-restricted cone NAT: la traduzione è la stessa del tipo Restricted cone NAT, tuttavia viene aggiunta una restrizione sul numero di porta: $addr_h : port_h$.

- Symmetric NAT: la traduzione di indirizzi e numero di porta dipende dalla destinazione, cioè anche usando sempre $addr_i : port_i$ come sorgente, questo verrà tradotto in un $addr_e : port_e$ differente per ogni $addr_h : port_h$.

Questi sono i tipi di NAT "classici", purtroppo la realtà è ben diversa poichè le realizzazione dei NAT differisce da produttore a produttore e spesso i NAT commerciali presentano dei comportamenti ibridi tra i tipi illustrati, rendendo difficoltosa la caratterizzazione del comportamento dei NAT "reali".

STUN

Session Traversal Utilities for NAT (STUN) [16] è un protocollo che fornisce un insieme di strumenti utili per attraversare i NAT in combinazione con altre tecniche come UDP Hole Punching [5] e TURN [13].

L'aspetto più interessante di STUN è l'algoritmo che permette di individuare il tipo di NAT operante nella rete locale mediante lo scambio di determinati messaggi con un server STUN.

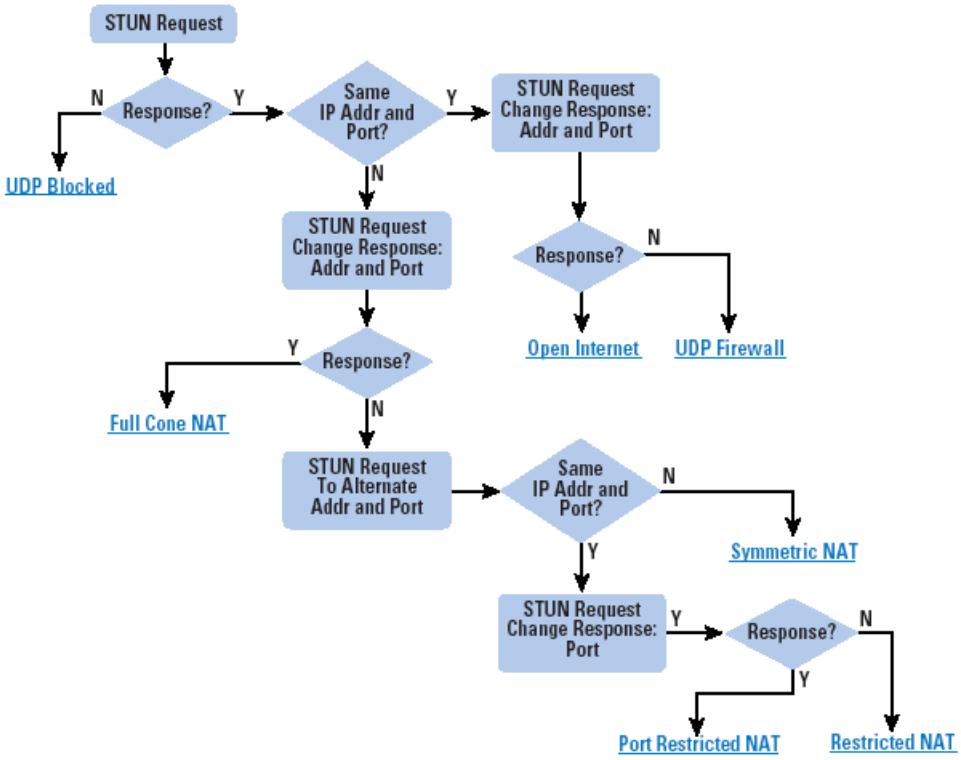

Figura 1.10: Diagramma di flusso dell'algoritmo STUN. Immagine tratta da [10].

Dato che le uniche tecniche veramente efficaci per il nat traversal valgono solo per il traffico UDP, anche STUN è trasportato da UDP e quindi rileva il tipo di NAT per quando riguarda la traduzione durante le comunicazioni UDP.

Nei casi Open Internet e Virtual Server la comunicazione UDP è possibile senza la necessità di tecniche aggiuntive.

Nei casi di NAT con restrizione la comunicazione in ingresso è possibile usando UDP Hole Punching.

In presenza di un NAT simmetrico la possibilità di comunicare via UDP è variabile, si può utilizzare UDP Hole Punching o TURN.
In caso di un firewall UDP l'unica soluzione è intervenire sul firewall stesso per permettere il traffico UDP.

Capitolo 2

Specifiche tunnel

Per un software che vive quasi interamente con lo scambio di informazioni via rete è facile che il numero di socket, necessari ai vari servizi disponibili, cresca molto velocemente. Se poi si considera che, ormai nella quasi totalità dei casi, le macchine su cui opera PariPari sono connesse ad Internet mediante un sistema di Network Address Translation la situazione si complica ulteriormente, poichè per molti utenti può essere difficoltoso configurare i dispositivi di rete per garantire la connettività in entrata ed uscita a tutti i servizi disponibili in PariPari. Proprio con questi due problemi in mente si è reso necessario pensare ad un sistema per limitare il numero totale di socket necessari al corretto funzionamento di PariPari e ad un sistema per oltrepassare i NAT garantendo, quando possibile, la connettività alla rete in entrata ed uscita senza la necessità di una configurazione manuale ad hoc dei dispositivi di rete quali router,natbox e access point.
Per garantire la connettività è stato aggiunto a ConnectivityNIO il package `nattraversal`, il quale usa strumenti e tecniche basati su tecnologie già rodate ed in uso come STUN, UDP Hole Punching e TURN.
A differenza di Nat Traversal, per cui si è potuto ricorrere a tecnologie già esistenti, per conseguire la riduzione del numero dei socket attivi è stato necessario realizzare una soluzione ad hoc per PariPari. L'idea di fondo è passare da una situazione in cui ogni servizio PariPari ha i propri socket ad

una situazione in cui tutto il traffico di rete passi attraverso un solo socket, un Tunnel con cui il plugin PariPari creda di comunicare direttamente con un altro nodo PariPari attraverso TCP o UDP utilizzando però un solo socket reale.

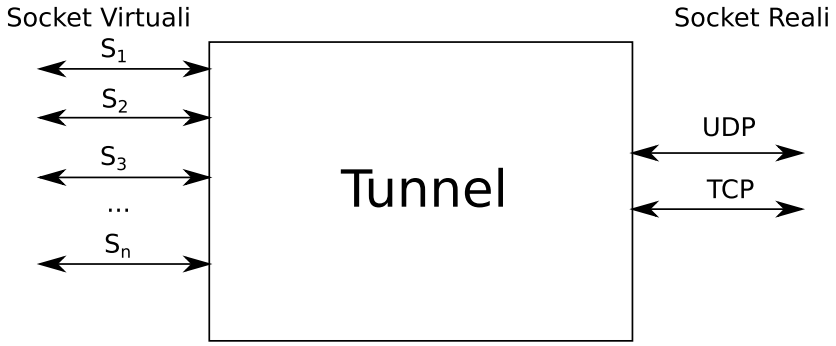

Figura 2.1: Schema ad alto livello del tunnel.

Nella sezione 2.1 verrà illustrato il processo di specifica e progetto dei protocolli di rete usati dal Tunnel, mentre nella sezione 2.2 verrà affrontato il problema di come indirizzare gli host della rete usando i nuovi protocolli.

2.1 Protocolli

Il primo passo nella realizzazione del tunnel per PariPari è identificare un metodo per effettuare il multiplexing/demultiplexing di una comunicazione che avviene attraverso i protocolli UDP e TCP. Quindi è necessario introdurre dei dati aggiuntivi a quelli della comunicazione vera e propria, i quali forniscano le informazioni minime per gestire il nuovo livello di astrazione aggiunto dal tunnel.
Gran parte delle informazioni per identificare una connessione sono già presenti a livello di trasporto (TCP e UDP) e nel protocollo IP, quindi per fare il multiplexing/demultiplexing basta aggiungere un nuovo livello di porte, dette porte virtuali. Per garantire la possibilità di sapere chi ha inviato tal comunicazione e a quale dei servizi PariPari è destinata, è sufficiente trasmettere

una coppia di porte virtuali, una per la sorgente ed una per la destinazione. L'indirizzamento dei nodi PariPari verrà trattato in dettaglio nella sezione 2.2.
Sebbene il tunnel possa operare sia su TCP sia su UDP, per motivi legati all'effettiva efficacia del modulo Nat Traversal la maggior parte del traffico di rete avverrà via UDP. E' per questo che è naturale scegliere come base di partenza per il protocollo PariPari per il tunnel il datagramma UDP. Infatti è sufficiente rimuovere dall'intestazione del datagramma UDP (Figura 2.2) i campi `Length` e `Checksum` per ottenere esattamente ciò di cui ha bisogno il tunnel per fare il multiplexing/demultiplexing delle comunicazioni. Una volta specificato il datagramma del protocollo PariPari, questo potrà essere incapsulato come payload dei due protocolli di trasporto utilizzati dal tunnel: TCP e UDP.

2.1.1 Protocollo PariPari

La base di partenza per l'intestazione del protocollo PariPari è l'header del datagramma UDP.

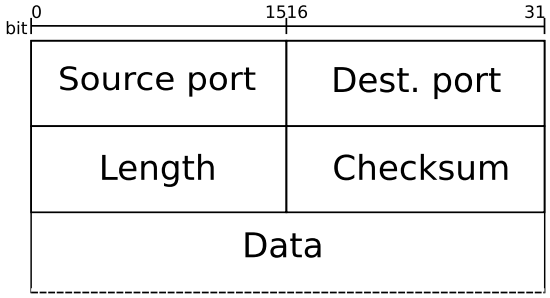

Figura 2.2: Struttura del datagramma UDP

Rimuovendo i campi `Length` e `Checksum` si ottiene un'intestazione minimale, ma sufficiente, per effettuare il multiplexing/demultiplexing delle comunicazioni.

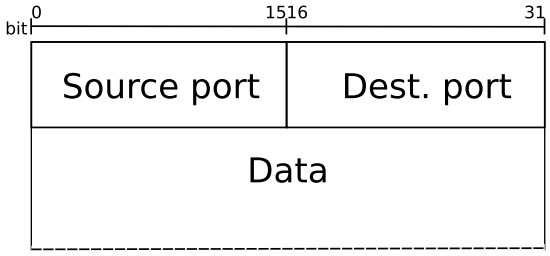

Figura 2.3: Struttura del datagramma PariPari.

Questo semplice protocollo ha un overhead molto limitato ed è intuitivo poichè ricalca UDP; tuttavia porta anche alcuni dei difetti di UDP, infatti non tiene traccia dello stato della connessione e non è affidabile poichè non garantisce nè la consegna nè la consegna ordinata dei pacchetti.

2.1.2 Protocollo PariPari garantito

Il protocollo PariPari sino a qui descritto copre molti scenari di utilizzo, infatti molti servizi di PariPari fanno affidamento su UDP per comunicare tra nodi nella rete, quindi l'introduzione del nuovo protocollo, in tutto simile ad UDP, non porta complicazioni nello sviluppo dei plugin. Ciò nonostante ci possono essere delle circostanze in cui non si può fare a meno di ricorrere ad un protocollo in cui ci sia la garanzia della ricezione del messaggio e che ci sia la certezza che i messaggi vengano ricevuti in ordine di invio dal destinatario. Per questo, accanto al protocollo PariPari, c'è il protocollo PariPari garantito. L'idea, almeno nella fase iniziale dello sviluppo, è quella di fornire un protocollo orientato al datagramma con garanzia di ricezione e consegna ordinata, però senza l'overhead del protocollo TCP dovuto a controllo di stato e di flusso.
Usando come base di partenza l'intestazione del protocollo PariPari è facile individuare le modifiche necessarie alla costruzione del protocollo garantito. Infatti, per quanto riguarda i datagrammi che effettivamente trasportano i dati, è sufficiente aggiungere un campo per l'acknowledge number, cioè un identificatore numerico che individua in maniera univoca un messaggio

trasmesso attraverso il protocollo PariPari garantito tra due host. Il primo messaggio ha sempre acknowledge number uguale a zero e l'assegnazione del valore di tale identificatore è gestita da una particolare struttura dati descritta in seguito nella sezione 3.3.1.

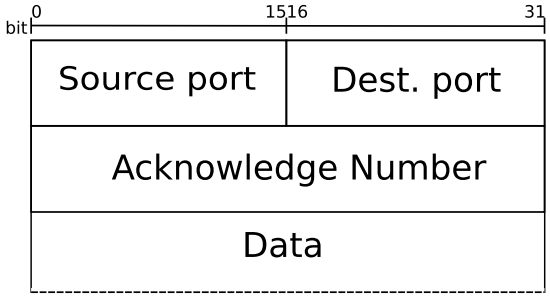

Figura 2.4: Struttura del datagramma PariPari garantito.

Ora è necessario definire un meccanismo per riferire al mittente l'avvenuta ricezione di un datagramma PariPari garantito. Questa operazione avviene attraverso un datagramma speciale di conferma.

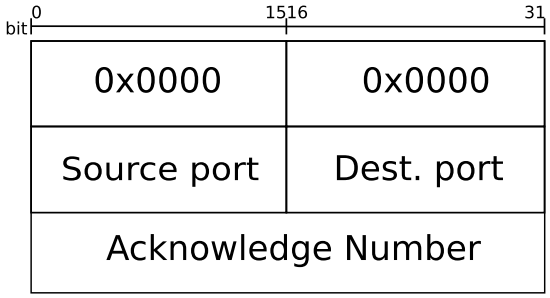

Figura 2.5: Struttura del datagramma di conferma ricezione del protocollo garantito.

Come si può notare nella Figura 2.5, l'intestazione del datagramma è formata da tre parti, i campi numero porta di destinazione e numero porta sorgente sono posti a zero per segnalare che il datagramma è di conferma di avvenuta ricezione. La seconda sezione consiste nei campi numero porta di destinazione e numero porta sorgente del datagramma che viene confermato,

tali valori sono necessari per semplificare l'operazione di identificazione della comunicazione a cui fa riferimento la ricevuta di consegna. Infine la terza, ed ultima, parte del datagramma consiste nell'acknowledge number riferito al datagramma di cui si dà conferma. Le dinamiche con cui vengono gestiti in ricezione ed invio i datagrammi del protocollo PariPari garantito verranno illustrate in seguito nel capitolo 3.

2.1.3 Framing su TCP

I due protocolli descritti, essendo orientati al datagramma, sono molto semplici da trasportare via UDP, infatti è sufficiente scrivere il datagramma PariPari all'interno del payload del pacchetto UDP. Tuttavia nel tunnel via TCP si presenta un problema dovuto alla natura stessa di TCP; infatti nel Transmission Control Protocol non esiste il concetto di datagramma e la comunicazione viene vista dagli host in gioco come un flusso di byte. Da qui nasce l'esigenza di estrapolare i datagrammi dei protocolli PariPari da una sequenza continua di byte.
L'approccio scelto per la soluzione del problema è quello del framing a sentinella con contatore. Prima di scrivere un datagramma PariPari nel canale TCP viene aggiunto un particolare header composto da due parti: la sentinella ed il contatore.
La sentinella è una specifica sequenza di byte che serve ad identificare l'inizio di un frame contenente un datagramma PariPari. Questa però non è sufficiente ad individuare con certezza un frame, infatti ci potrebbero essere dei casi in cui la sequenza identificata come sentinella, sia in realtà parte del messaggio. Per questo viene aggiunto anche il campo contatore il quale contiene la lunghezza in byte del datagramma PariPari (intestazione e dati). Se la lunghezza del datagramma estrapolato dal flusso TCP non coincide con quella del contatore, il datagramma viene scartato come non valido.

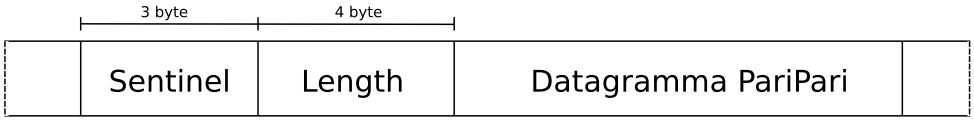

Figura 2.6: Struttura del frame su TCP.

L'algoritmo usato per estrarre i datagrammi PariPari dal flusso TCP verrà illustrato in seguito nella sezione 3.4.1.

2.1.4 Numerazione delle porte virtuali

I campi che contengono il numero delle porte virtuali sono di due byte, quindi teoricamente possono indirizzare 65535 porte. Ciò nonostante, per semplificare l'identificazione dei protocolli in fase di ricezione, lo spazio dei numeri di porta è stato diviso per protocollo di trasporto e per protocollo PariPari. I numeri di porta da 1 a 32767 sono riservati alle comunicazioni con protocollo PariPari, mentre i numeri da 32768 a 65535 sono riservati alle comunicazioni con protocollo PariPari garantito. A loro volta i due spazi sono divisi per protocollo di trasporto usato, come riportato in tabella 2.1.

	Via UDP	Via TCP
Protocollo PariPari	1-16383	16384-32767
Protocollo PariPari garantito	32768-49151	49152-65535

Tabella 2.1: Divisione dei numeri di porta virtuali.

L'utente finale ha a disposizione 16384 numeri di porta per ognuno dei due protocolli PariPari, la traduzione dei numeri di porta avviene in fase di invio e ricezione dei datagrammi.

2.1.5 PariPariDatagram e PariPariGrantedDatagram

`PariPariDatagram` e `PariPariGrantedDatagram` sono le due classi che realizzano i due protocolli descritti ed appartengono al package `paripari.con`

`nectivitynio.tunneling`.
La maggior parte dei costruttori e metodi ricalcano quelli di `DatagramPacket` di `java.net`, tuttavia ci sono delle differenze. La prima è che non viene utilizzata la classe `InetSocketAddress` come indirizzo per i datagrammi, ma si usa la classe `PPID` la quale verrà illustrata nella sezione 2.2. La seconda differenza è la presenza di metodi ausiliari che rispecchiano alcune caratteristiche specifiche del tunnel. Sono presenti dei metodi per specificare il tipo di tunnel da utilizzare: UDP o TCP. Inoltre c'è il metodo `getRawData` che restituisce la rappresentazione grezza in byte del datagramma PariPari, ottenuta aggiungendo le opportune intestazioni ai dati da trasmettere in base al protocollo di trasporto ed al protocollo PariPari.
La classe `PariPariGrantedDatagram` ha dei metodi ausiliari aggiuntivi, ci sono i metodi `setAckNum` e `getAckNum` i quali, rispettivamente, impostano e restituiscono l'acknowledge number come integer. Inoltre c'è il metodo `timeout` il quale restituisce un valore booleano che rappresenta la necessità o meno di un reinvio del datagramma, poichè è scaduto un timeout di attesa della ricevuta di consegna.
Non c'è una classe specifica per i datagrammi di conferma ricezione, vengono gestiti usando direttamente i valori contenuti nel pacchetto di trasporto.

2.2 Indirizzamento

A causa della presenza dei protocolli specifici per il tunnel, non è più sufficiente l'uso di `InetSocketAddress` (Indirizzo IP e numero di porta) per indirizzare i datagrammi attraverso il tunnel. Occorre un metodo di indirizzamento ad hoc, il quale deve rappresentare i seguenti fatti:

- la comunicazione attraverso il tunnel può avvenire via UDP e via TCP
- è necessario specificare un numero di porta virtuale per i protocolli PariPari

- sono necessarie delle informazioni aggiuntive, specifiche per ogni host, che indichino, se necessarie, le eventuali procedure per poter instaurare una connessione con l'host specificato ed attraversare i Nat presenti nel percorso

Avendo in mente queste specifiche, i valori necessari per indirizzare un host e quindi comunicare attraverso il tunnel sono elencati nella tabella 2.2.

Nome	Valore
Indirizzo IP	L'indirizzo IPv4 pubblico dell'host
Numero porta UDP	Il numero della porta, associata al protocollo UDP, in ascolto per la ricezione dei datagrammi dal tunnel.
Numero porta TCP	Il numero della porta, associata al protocollo TCP, in ascolto per la ricezione dei datagrammi dal tunnel.
Numero porta virtuale	Il numero della porta virtuale del protocollo PariPari con cui si vuole comunicare.
IP + numero porta ServerNat	L'indirizzo IP e il numero della porta del tunnel UDP del ServerNAT associato a tale host.
Numero porta virtuale ServerNat	Il numero della porta virtuale di ascolto del ServerNat associato a tale host.
Tipo di rete	Un codice di 1 byte il quale identifica il tipo di rete locale in cui si trova l'host, serve per decidere se sono necessarie delle operazioni di Nat Traversal.
Data	Un campo di 2 byte di dati aggiuntivi necessari per casi particolari di Nat Traversal.

Tabella 2.2: Tabella dei campi di un `PPID`.

I primi quattro campi assieme al campo "Tipo di rete" sono obbligatori in quanto specificano le informazioni basilari per comunicare attraverso il tunnel; invece i campi riferiti a ServerNat e Nat Traversal dipendono dal "Tipo

di rete", ci sono casi, come ad esempio `OPEN_INTERNET` o `VIRTUAL_SERVER`, in cui i campi non sono considerati.

2.2.1 La classe PPID

La classe `PPID` del package `paripari.connectivitynio.tunneling` realizza l'indirizzamento descritto in precedenza. La classe presenta metodi di tipo getter/setter per tutti i campi necessari, inoltre c'è un metodo che restituisce una rappresentazione in byte, di lunghezza fissa, del `PPID`; questo metodo è necessario poichè il `PPID` è pensato per essere pubblicato come valore nella rete DHT di PariPari. Infatti dato che questo tipo di indirizzamento specifica un "socket virtuale", il plugin che ne fa uso può pubblicare una coppia chiave/valore sulla rete DHT dove la chiave identifica il tipo di servizio offerto ed il valore è la rappresentazione in byte del `PPID`. Quindi qualsiasi nodo PariPari potrà ricercare servizi offerti attraverso il tunnel, semplicemente ricercando delle chiavi specifiche e costruendo il `PPID` a partire dai risultati della ricerca DHT.

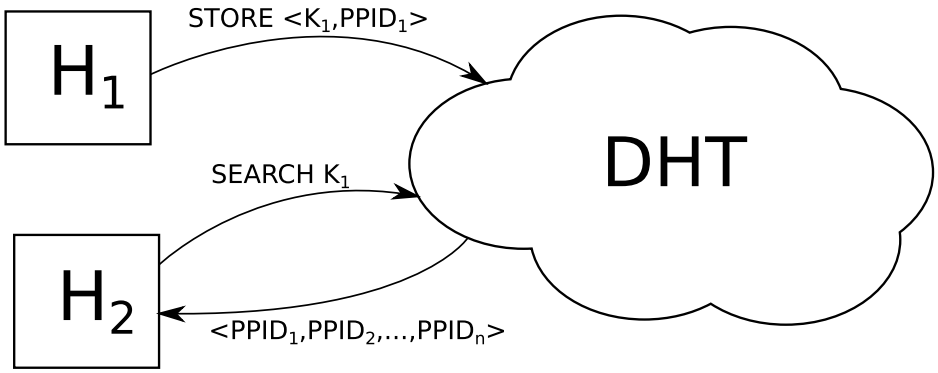

Figura 2.7: Schema di pubblicazione e ricerca sulla rete DHT di un servizio offerto via tunnel.

Capitolo 3

Funzionamento tunnel

Nel seguente capitolo è descritta la realizzazione del tunnel. É stato scelto un approccio graduale; il primo argomento trattato sarà l'interfaccia `ITunneling` (Sezione 3.1) la quale astrae l'accesso alle funzionalità del tunnel, in seguito verrà descritta la progettazione delle varie componenti: il deposito per i pacchetti ricevuti (Sezione 3.2), il funzionamento del tunnel via UDP (Sezione 3.3) e via TCP (Sezione 3.4). Infine si tratterà il funzionamento dei nuovi socket introdotti dal tunnel (Sezione 3.5) e di come `PPIDResolver` faccia cooperare il tunnel con Nat Traversal (Sezione 3.6).

3.1 Interfaccia ITunneling

Prima di illustrare il funzionamento dei vari meccanismi che permettono al tunnel di funzionare, è necessario introdurre un'interfaccia la quale coordini le varie componenti e permetta l'utilizzo del tunnel con chiamate a basso livello. L'interfaccia in questione è `ITunneling` del package `paripari.connectivity-nio.tunneling.interfaces` ed ha i seguenti metodi:

- `acquirePort` : permette di acquisire un numero di porta specificando il numero, il protocollo PariPari ed un riferimento al proprietario della porta.

- `releasePort` : permette di rilasciare tutte le risorse in uso legate al numero di porta e protocollo specificati.
- `sendDatagram`/`sendGrantedDatagram`: permettono di inviare rispettivamente un datagramma PariPari ed un datagramma PariPari garantito.
- `receiveDatagram`/`receiveGrantedDatagram`: permettono di ricevere, se disponibile, il datagramma successivo dal buffer associato al numero di porta specificato.

L'interfaccia è implementata dalla classe `Tunneling` del package `paripari.connectivitynio.tunneling`, la quale oltre a fornire le chiamate a basso livello per uso interno a ConnectivityNIO si occupa dell'avvio e del coordinamento di tutte le parti del tunnel come i moduli di invio e moduli di ricezione.
Di seguito verranno analizzati tutti i blocchi funzionali che compongono il tunnel.

3.2 PacketStore

`PacketStore` è una classe dedicata all'immagazzinamento dei datagrammi PariPari ricevuti attraverso il tunnel. Viene chiamata in causa da thread diversi e quindi è pensata per funzionare come un monitor.
I suoi metodi si dividono in due tipologie: quelli per depositare nuovi datagrammi e quelli per usare il magazzino. Prima di entrare nel dettaglio delle strutture dati usate per ognuno dei due protocolli PariPari, è utile descrivere il funzionamento della classe per la ricezione dei datagrammi, la procedura è concettualmente identica per i due protocolli usati dal tunnel.
Il primo metodo da chiamare è forse il più importante:

- `boolean acquirePort(int vPort, boolean granted,TunnelSocket owner);`

Prende come parametri il numero di porta desiderato, il tipo di protocollo PariPari da utilizzare ed un riferimento al proprietario della porta. Il metodo

si occupa di inizializzare le strutture dati relative al protocollo specificato, operazione fondamentale per operare correttamente con il tunnel. Infatti la politica predefinita è di scartare i datagrammi indirizzati ad un numero di porta non acquisito da qualcuno. Il metodo termina restituendo un valore booleano che rappresenta il successo o meno dell'operazione di acquisizione, generalmente un risultato `false` indica che il numero di porta è già stato acquisito.
Dopo l'acquisizione della risorsa numero di porta, si possono finalmente prelevare i datagrammi in arrivo dai tunnel UDP e TCP. Ciò avviene mediante i metodi:

- `PariPariDatagram getDatagram(int vPort);`
- `PariPariGrantedDatagram getGrantedDatagram(int vPort);`

Chiamando tali metodi si preleva il primo datagramma disponibile nei buffer associati alla porta specificata, se i buffer sono vuoti o il numero di porta non è stato precedentemente acquisito, viene restituito un riferimento `null`.
L'operazione per concludere l'uso di una porta virtuale è il rilascio delle risorse allocate, mediante il metodo:

- `boolean releasePort(int vPort, boolean granted);`

Il metodo si occupa di rilasciare tutte le risorse di memoria che sono state allocate per l'immagazzinamento dei datagrammi e quindi rende nuovamente disponibile il numero di porta specificato. Nel caso siano presenti dei datagrammi non ancora letti, questi verranno semplicemente persi.
E' ora necessario introdurre due metodi, i quali depositano i datagrammi ricevuti in ingresso dai canali UDP e TCP:

- `void store(PariPariDatagram pack);`
- `void storeGranted(PariPariGrantedDatagram pack);`

Sebbene ci siano due metodi diversi per ognuno dei due protocolli del tunnel, essi operano concettualmente allo stesso modo anche se la loro realizzazione

è diversa a causa delle strutture dati sottostanti ed alle peculiarità del protocollo a cui si riferiscono.
Per prima cosa viene verificato se la porta di destinazione è stata acquisita da qualcuno, se ciò non è avvenuto il datagramma viene scartato e quindi perso. Se la verifica ha successo, viene reperita la struttura dati associata al numero di porta e viene aggiunto al buffer il nuovo datagramma. Prima di terminare la procedura di `store`, viene effettuata una chiamata di tipo `notify` al riferimento del proprietario del numero di porta, specificato in fase di acquisizione; il motivo di questa chiamata verrà spiegato nella sezione 3.5.
A questo punto si possono introdurre le strutture dati usate da `PacketStore` per immagazzinare i datagrammi del protocollo PariPari e del protocollo PariPari garantito.

3.2.1 Strutture dati protocollo PariPari

Data la semplicità del protocollo PariPari non sono necessarie strutture dati complesse per immagazzinare i `PariPariDatagram`. La struttura principale è composta da una tabella di hash. Come chiave viene usato il numero di porta, come valore c'è un riferimento ad un vettore a lunghezza dinamica di `PariPariDatagram`.

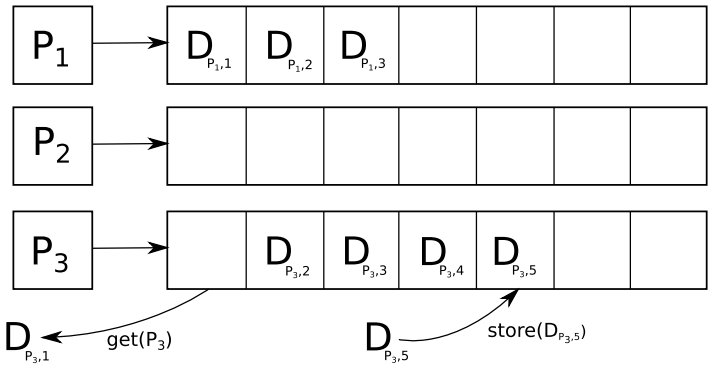

Figura 3.1: Schema del buffer del protocollo PariPari. P_i è un numero di porta, $D_{P_i,j}$ è il datagramma j-esimo destinato alla porta P_i.

In fase di acquisizione della risorsa porta viene semplicemente creato un nuovo vettore, il quale viene inserito nella tabella usando il numero di porta come chiave. L'operazione di inserimento e prelievo dei datagrammi dal buffer è semplice poichè avviene con un table lookup. L'accodamento di datagrammi provenienti da host diversi, ma indirizzati alla stessa porta virtuale, ricalca la maggior parte delle implementazioni dello stack UDP. I datagrammi vengono accodati nel vettore secondo l'ordine di arrivo, non distinguendo gli host sorgenti; l'operazione, se necessaria, di distinguere i datagrammi ricevuti per host sorgente è lasciata all'utilizzatore del tunnel.
Oltre alla tabella per l'immagazzinamento dei datagrammi viene mantenuta una tabella di look up anche per i riferimenti ai proprietari delle porte.

3.2.2 Strutture dati protocollo PariPari garantito

Le strutture dati necessarie ad immagazzinare i `PariPariGrantedDatagram` sono leggermente più complesse, a causa di alcune peculiarità del protocollo garantito; infatti è necessario avere buffer distinti per host sorgenti distinti. Bisogna tenere presenti questi fatti:

- ogni host sorgente ha la propria "sessione" di comunicazione.
- i `PariPariGrantedDatagram` vanno ordinati secondo l'acknowledge number.
- alcuni datagrammi possono essere accodati più volte in caso di ritrasmissione da parte dell'host sorgente.
- non si può prelevare dal buffer un datagramma se non è stato prelevato quello con acknowledge number inferiore.

Da qui, nasce l'esigenza di due classi contenitore: `BufferBucket` e `Granted Buffer`.
La classe `GrantedBuffer` rappresenta un contenitore di datagrammi garantiti di una specifica sessione. Si occupa di mantenere ordinati i `PariPariGranted`

`Datagram`, di scartare pacchetti duplicati e regolare il prelievo dei datagrammi rispettando i vincoli imposti dagli acknowledge numbers. I campi che permettono il funzionamento della classe sono due; il primo è una istanza di `PriorityQueue`, la quale contiene i datagrammi, li mantiene ordinati per acknowledge number e scarta i duplicati; il secondo campo è una variabile di stato che mantiene il valore del prossimo acknowledge number valido e serve per regolare il prelievo e l'inserimento dei datagrammi nella coda a priorità.

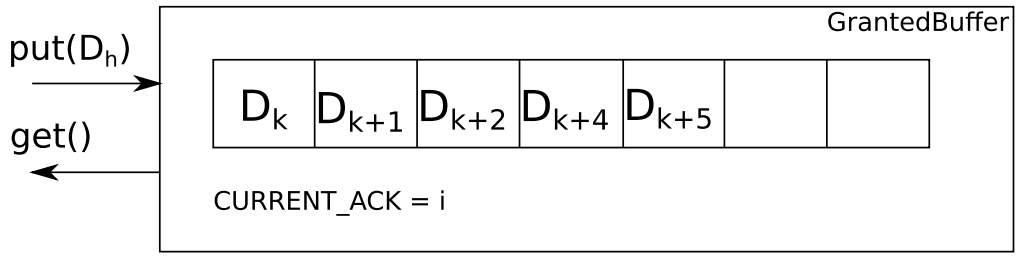

Figura 3.2: Schema di `GrantedBuffer`. D_k è il datagramma con acknowledge number k. L'operazione `get` restituisce il datagramma in testa al buffer solo quando $k = i$.

Su `GrantedBuffer` si opera con due metodi:

- `void put(PariPariGrantedDatagram pack)`: se il pacchetto specificato ha acknowledge number superiore a quello corrente viene inserito nella coda a priorità, altrimenti viene scartato dato che è un datagramma duplicato ed una copia è già stata ricevuta.
- `PariPariGrantedDatagram get()`: se ci sono datagrammi in coda e l'acknowledge number del pacchetto in testa alla coda è uguale a quello corrente, viene aggiornata la variabile di stato del numero di ack corrente e viene rimossa, e restituita, la testa della coda. Se il datagramma in testa ha un acknowledge number superiore a quello corrente o la coda è vuota viene restituito un riferimento `null`.

`GrantedBuffer` immagazzina i datagrammi provenienti da un solo host sorgente, è necessario introdurre la classe `BufferBucket`, la quale fa da contenitore a diversi `GrantedBuffer` associati allo stesso numero di porta di

destinazione. Al suo interno c'è un vettore di `GrantedBuffer` e presenta i seguenti metodi:

- `PariPariGrantedDatagram get()`: restituisce il prossimo datagramma dal primo `GrantedBuffer` non vuoto, altrimenti restituisce un riferimento `null`.
- `void put(PariPariGrantedDatagram pack)`: accoda il datagramma al proprio `GrantedBuffer` di riferimento, se questo non esiste (è il caso del primo datagramma ricevuto da un host sorgente) ne viene creato uno ex novo.

E' necessario definire una politica di prelievo dei datagrammi dai `Granted Buffer`. Ad ogni chiamata di `get`, il vettore dei `GrantedBuffer` viene riordinato; in testa ci sarà il buffer che non viene usato da più tempo mentre in coda ci sarà il buffer appena usato. Questo garantisce un minimo di equità nel prelievo dei datagrammi, tuttavia con la pratica si potrà affinare meglio la politica di gestione considerando altri fattori.

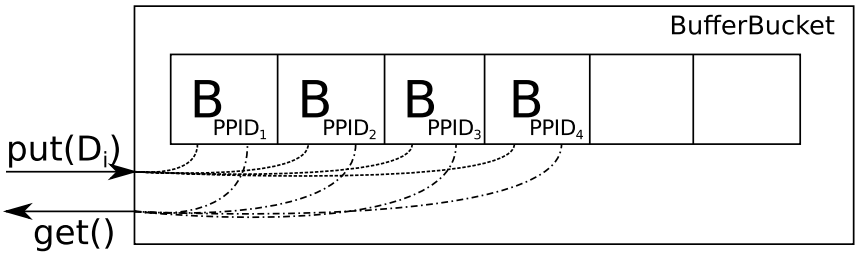

Figura 3.3: Schema di `BufferBucket`. B_{PPID_i} è un `GrantedBuffer` associato al `PPID` i-esimo, D_i è un datagramma proveniente da $PPID_i$

.

Dopo avere illustrato il funzionamento delle classi contenitore si può passare a come esse vengono utilizzate in `PacketStore`. La loro gestione è sostanzialmente la medesima del caso del protocollo non garantito; c'è una tabella di hash la quale ha il numero di porta come chiave e un `BufferBucket` come valore. L'inserimento ed il prelievo dei datagrammi avviene usando i

metodi del `BufferBucket` ottenuto con un look-up mediante il numero di porta. Anche nel protocollo garantito viene mantenuta una tabella dei proprietari dei numeri di porta. C'è un ultimo problema da risolvere; se un `GrantedBuffer` rimane vuoto per un certo lasso di tempo è da considerarsi esaurito. Per questo motivo, mediante un `PariPariTimer`, viene periodicamente scorsa la tabella dei `BufferBucket` sui quali viene chiamato il metodo `cleanDeadBuffers`. Tale metodo verifica se all'interno dell'istanza di `BufferBucket` sono presenti dei `GrantedBuffer` non più validi e li elimina dal vettore dei buffer.

3.3 Funzionamento via UDP

3.3.1 UdpSender

La classe `UdpSender` si occupa dell'invio dei datagrammi PariPari attraverso un socket UDP. Implementa l'interfaccia `PariPariRunnable` e quindi vive in un `PariPariThread` dedicato.
Il suo funzionamento è semplice poichè riflette il paradigma del produttore/consumatore.
All'interno della classe c'è un `BlockingQueue` di `PariPariDatagram`, i thread che desiderano inviare dei datagrammi li accodano in questa struttura dati attraverso i metodi:

- `void sendDatagram(PariPariDatagram pack);`
- `void sendGrantedDatagram(PariPariGrantedDatagram pack);`

Parallelamente, nel metodo `go` di `UdpSender` un ciclo si occupa di:

- prelevare il datagramma successivo dalla coda
- costruire un `DatagramPacket` usando le informazioni prelevate dal datagramma PariPari
- inviare nel canale UDP il `DatagramPacket` appena costruito

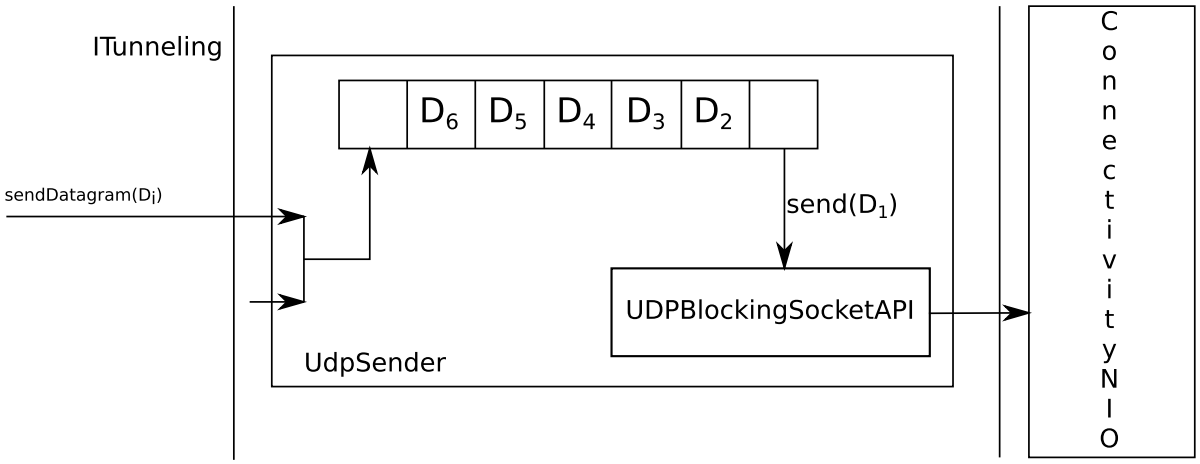

Figura 3.4: Schema `UdpSender`. D_i sono datagrammi.

UdpGrantedManager

La classe `UdpSender` invia senza distinzione `PariPariDatagram` e `PariPariGrantedDatagram` e ciò è sufficiente per quanto riguarda il protocollo non garantito. Tuttavia per il protocollo PariPari garantito è necessario un modulo che si occupi di:

- assegnare un acknowledge number ai nuovi datagrammi in base all'host di destinazione.
- tenere una copia dei datagrammi che non hanno ricevuto la conferma di recapito.
- reinviare i datagrammi che non ricevono la conferma di recapito entro un certo intervallo di tempo.

Ciò avviene grazie alla classe `UdpGrantedManager`, la quale fa da tramite tra chi vuole inviare datagrammi attraverso il protocollo garantito e la classe `UdpSender` che si occupa dell'effettivo invio dei datagrammi.
L'invio di un nuovo datagramma avviene tramite il metodo `send`, il quale esegue le seguenti operazioni:

- Assegnazione acknowledge number: preleva da una tabella di hash il successivo acknowledge number, con riferimento all'host di destinazione.
- Accodamento in volo: il datagramma viene accodato in una coda interna dedicata ai pacchetti in volo, ovvero i datagrammi che non hanno ancora ricevuto la conferma di recapito.
- Accodamento su `UdpSender`: il datagramma viene accodato su `UdpSender` per l'invio effettivo.

Successivamente, quando il modulo di ricezione riceverà la conferma di recapito, verrà chiamato il metodo `acknowledge`, il quale si occuperà di rimuovere dalla coda dei datagrammi in volo il `PariPariGrantedDatagram` trasmesso con successo.
Tuttavia può capitare che un datagramma non venga recapitato al primo invio, quindi è necessario un ulteriore invio. Periodicamente, grazie ad un `PariPariTimer`, viene controllata la coda dei pacchetti in volo, e se questi rimangono accodati per un intervallo di tempo superiore a `TIMEOUT` vengono inviati nuovamente ad `UdpSender` per un nuovo invio.
Le informazioni riguardanti gli acknowledge number da assegnare non sono contenute direttamente nella tabella di hash, ma in un oggetto contenitore chiamato `SenderSessionWrapper`. Periodicamente, attraverso un `PariPariTimer`, vengono controllati gli oggetti contenitore e se questi non sono stati utilizzati da un intervallo di tempo superiore a `TIME_TO_LIVE` vengono eliminati dalla tabella e la comunicazione garantita risulta conclusa.

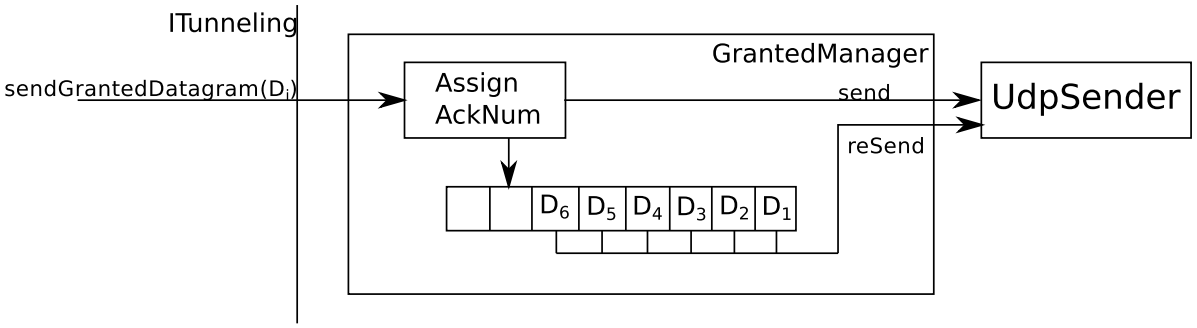

Figura 3.5: Schema `UdpGrantedManager`. D_i sono datagrammi.

3.3.2 UdpReceiver

La classe `UdpReceiver` si occupa di ricevere i datagrammi PariPari in ingresso dal canale UDP; implementa l'interfaccia `PariPariRunnable` e quindi vive in un thread dedicato. Nel metodo `go` c'è un ciclo il quale legge dal socket UDP un `DatagramPacket` alla volta e ne ispeziona il contenuto.
I casi che si possono verificare sono i seguenti:

- Datagramma non valido: il contenuto del datagramma non è coerente con le specifiche dei protocolli, viene scartato.

- Datagramma di acknowledge: il datagramma ricevuto è di conferma recapito, viene segnalata la ricezione alla classe `UdpGrantedManager` attraverso il metodo `acknowledge`, specificando l'id della comunicazione e l'acknowledge number, estraendoli dal datagramma ricevuto.

- Datagramma PariPari: partendo dal datagramma ricevuto viene costruito un `PariPariDatagram`, il quale viene inviato a `PacketStore` attraverso il metodo `store`.

- Datagramma PariPari garantito: viene costruito un `DatagramPacket` di conferma di ricezione seguendo le specifiche del protocollo e viene spedito immediatamente al mittente attraverso un socket UDP. Successivamente viene costruito un `PariPariGrantedDatagram` contenente il

messaggio ricevuto, il quale viene inviato a `PacketStore` attraverso il metodo `storeGranted`.

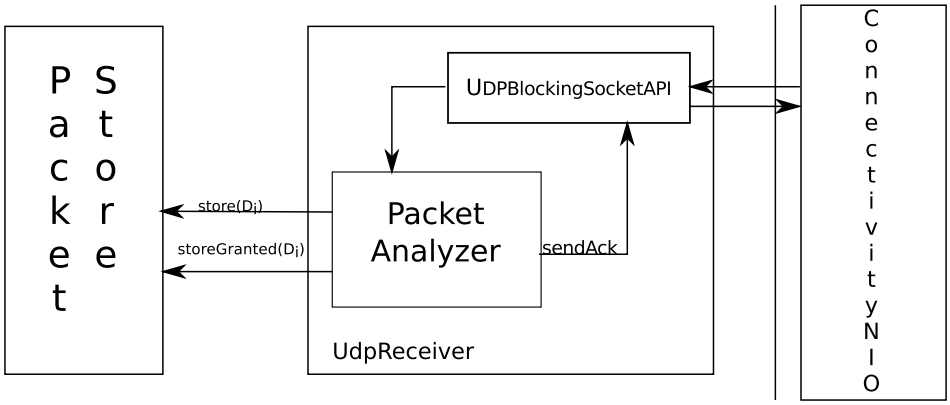

Figura 3.6: Schema `UdpReceiver`. D_i sono datagrammi.

3.3.3 Schema tunnel Udp

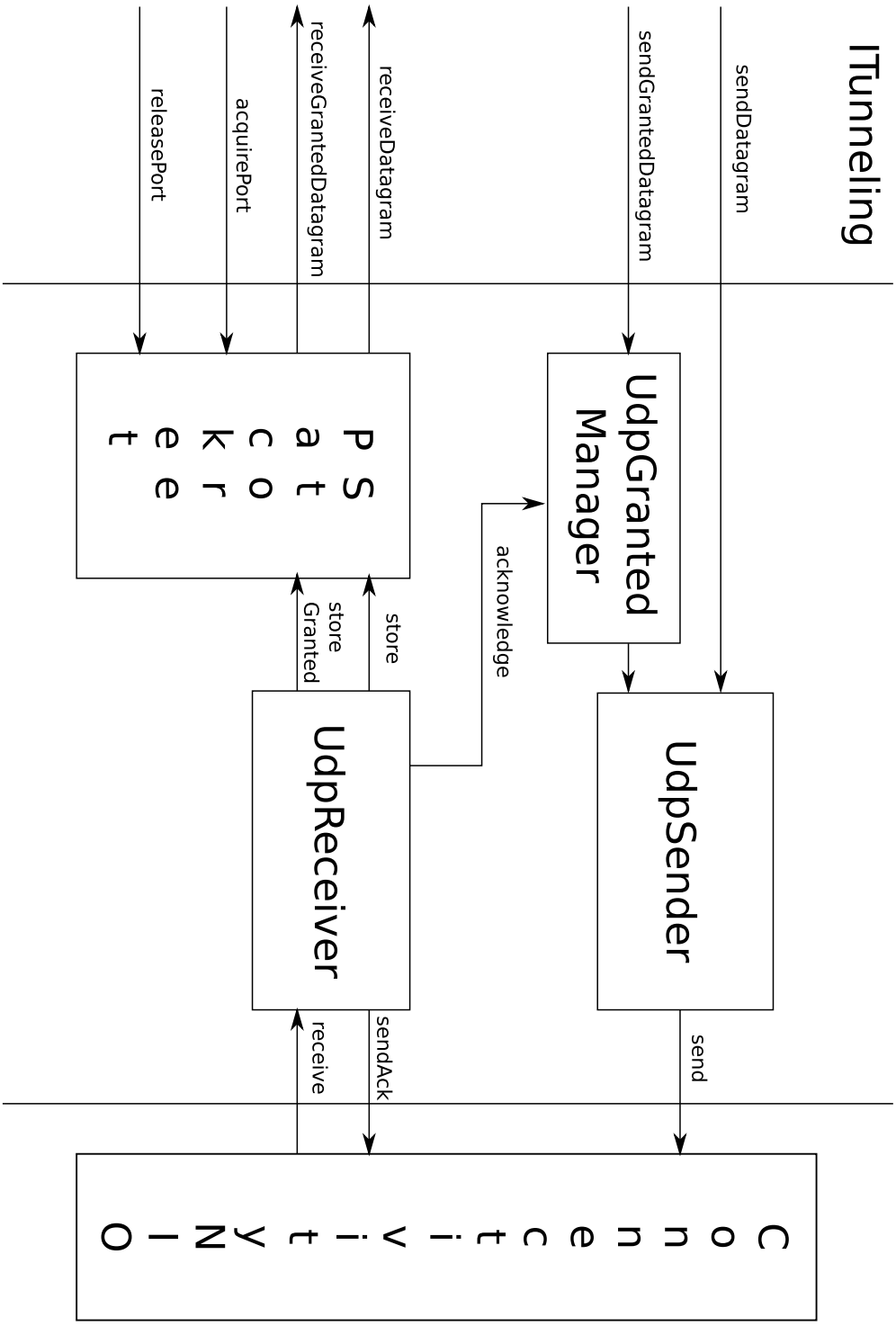

Figura 3.7: Schema tunnel via UDP.

3.4 Funzionamento via TCP

Nella realizzazione del tunnel TCP è difficile individuare dei blocchi con funzionalità specifiche come nel caso via UDP. Infatti ci sono delle complicazioni dovute alla natura stessa del protocollo TCP che impediscono una separazione netta tra modulo di invio e modulo di ricezione. La principale difficoltà sta nei socket TCP; a differenza dei socket UDP, i quali possono inviare e ricevere datagrammi da e verso qualsiasi host, ogni socket TCP crea un canale bidirezionale tra gli host che vogliono comunicare. In aggiunta c'è un ulteriore complicanza dovuta alla fase di inizializzazione della connessione.
Un altro fatto rilevante che differenzia la realizzazione del tunnel TCP dal tunnel UDP è la fase di ricezione, nel caso TCP questa è un'operazione costosa in termini computazionali, infatti c'è la necessità di estrare i datagrammi da un flusso di byte, come illustrato nella sezione 2.1.3. Da qui deriva la necessità di ricevere i dati in ingresso dai vari canali TCP in maniera parallela.

3.4.1 TcpStream

`TcpStream` è una classe con molteplici finalità e rappresenta una connessione TCP tra due host. Per prima cosa, la classe fa da contenitore al socket TCP di ConnectivityNIO che effettivamente invia e riceve dati dalla rete, quindi rappresenta il mezzo attraverso cui comunicare con l'host a cui è associata.
Il primo metodo da illustrare è quello che permette l'invio di dati:

- `void write(byte[] rawData);`

Il suo funzionamento è semplice, poichè il metodo `write` si limita a scrivere l'array di byte nel socket TCP.
Di maggiore interesse è il funzionamento della parte di ricezione. La classe `TcpStream` implementa l'interfaccia `PariPariRunnable` e quindi vive in un `PariPariThread` dedicato. Questo thread si occupa di ricevere dati dal canale TCP e di estrarre i datagrammi PariPari dai frame.
L'estrazione dei datagrammi è realizzata mediante un algoritmo a stati.

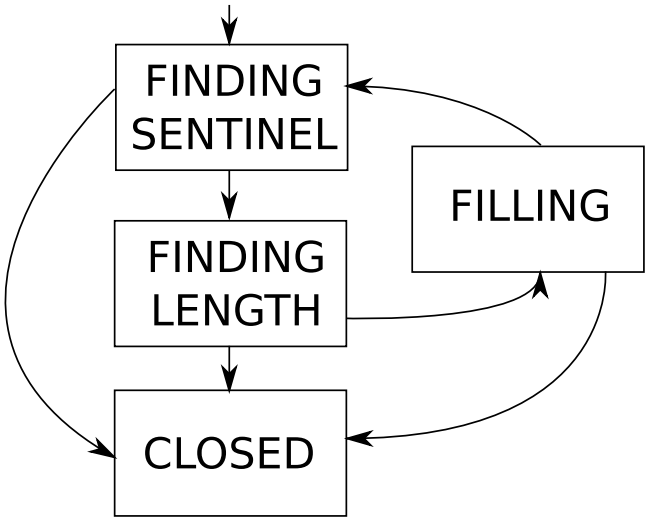

Figura 3.8: Schema degli stati di un `TcpStream`.

- `FINDING_SENTINEL`: indica che si sta cercando una sentinella per sincronizzarsi con i frame.
- `FINDING_LENGTH`: indica che si sta estraendo la lunghezza del frame.
- `FILLING`: si sta riempiendo un buffer temporaneo, il quale conterrà la rappresentazione in byte di un datagramma PariPari; il numero di byte dopo cui interrompere il riempimento è quello ricavato dal frame.
- `CLOSED`: il `TcpStream` è chiuso.

Il passaggio tra gli stati avviene ispezionando un buffer temporaneo in cui vengono scritti i byte ricevuti dal socket TCP.
Una volta estratta la rappresentazione in byte del datagramma PariPari, la procedura da seguire è sostanzialmente la stessa usata nel caso UDP. Si ispeziona il contenuto del buffer e si verifica uno dei seguenti casi:

- Datagramma non valido: il contenuto del datagramma non è coerente con le specifiche dei protocolli, viene scartato.

- Datagramma di acknowledge: il datagramma ricevuto è di conferma recapito, viene segnalata la ricezione alla classe `TcpGrantedManager` attraverso il metodo `acknowledge`, specificando l'id della comunicazione e l'acknowledge number estraendoli dal datagramma ricevuto.

- Datagramma PariPari: partendo dal datagramma ricevuto viene costruito un `PariPariDatagram`, il quale viene inviato a `PacketStore` attraverso il metodo `store`.

- Datagramma PariPari garantito: viene costruito un `DatagramPacket` di conferma di ricezione seguendo le specifiche del protocollo e viene spedito immediatamente al mittente attraverso il socket TCP. Successivamente viene costruito un `PariPariGrantedDatagram` contenente il messaggio ricevuto, il quale viene inviato a `PacketStore` attraverso il metodo `storeGranted`.

3.4.2 SocketStore

Le funzionalità, a basso livello, di invio e ricezione sono assolte dalla classe `TcpStream`; tuttavia c'è l'esigenza di avere un contenitore che tenga traccia di tutti i `TcpStream` attivi e ne permetta l'uso attraverso un'interfaccia unificata. Questa interfaccia è fornita dalla classe `SocketStore`; al suo interno c'è una lista di `TcpStream` attivi, la quale viene periodicamente controllata per eliminare eventuali `TcpStream` che non vengono utilizzati da un certo lasso di tempo e che quindi possono essere chiusi. Inoltre permette l'invio dei datagrammi PariPari, analogamente alla classe `UdpSender`, mediante i metodi:

- `void sendDatagram(PariPariDatagram pack);`

- `void sendGrantedDatagram(PariPariGrantedDatagram pack);`

La loro realizzazione è semplice poichè si limita ad inserire i datagrammi passati come parametro in una coda. Lo svuotamento di tale coda avviene

mediante il paradigma del produttore/consumatore.
`SocketStore` implementa l'interfaccia `PariPariRunnable` ed il metodo `go` esegue di continuo le seguenti operazioni:

- estrazione di un datagramma dalla coda.
- ricerca del `TcpStream` associato all'host di destinazione, se non presente ne viene creato uno ex novo.
- creazione della rappresentazione in byte del datagramma ed invio attraverso il `TcpStream`.

Nel caso dell'invio di `PariPariGrantedDatagram`, c'è anche nel tunnel via TCP l'intervento di una classe esterna per garantire la certezza del recapito: `TcpGrantedManager`. La classe `TcpGrantedManager` ha le stesse identiche funzionalità della classe `UdpGrantedManager`, anche la realizzazione è del tutto simile dato che usa strutture dati derivate o ispirate da quelle usate nel tunnel via UDP come spiegato nella sezione 3.3.1.
Sino a qui, l'unico punto del codice in cui avviene la generazione di nuovi `TcpStream` è nel codice di invio dei datagrammi, infatti se non viene trovato un `TcpStream` già attivo ne viene creato uno nuovo.
Per questo c'è la necessità di introdurre una nuova classe che si occupi di accettare le connessioni TCP in ingresso. Ciò avviene mediante una classe privata, interna a `SocketStore`, denominata `TcpAccepter`. Il suo funzionamento è semplice; la classe contiene al suo interno un socket server TCP. Nel metodo `go`, ereditato implementando `PariPariRunnable`, avviene la chiamata bloccante `accept` al server socket. Questo particolare metodo blocca il codice fino a che non riceve una connessione in ingresso, a quel punto restituisce un socket TCP con il quale viene creato un nuovo `TcpStream` da aggiungere alla lista dei `TcpStream` attivi di `SocketStore`.

3.4.3 Schema tunnel via TCP

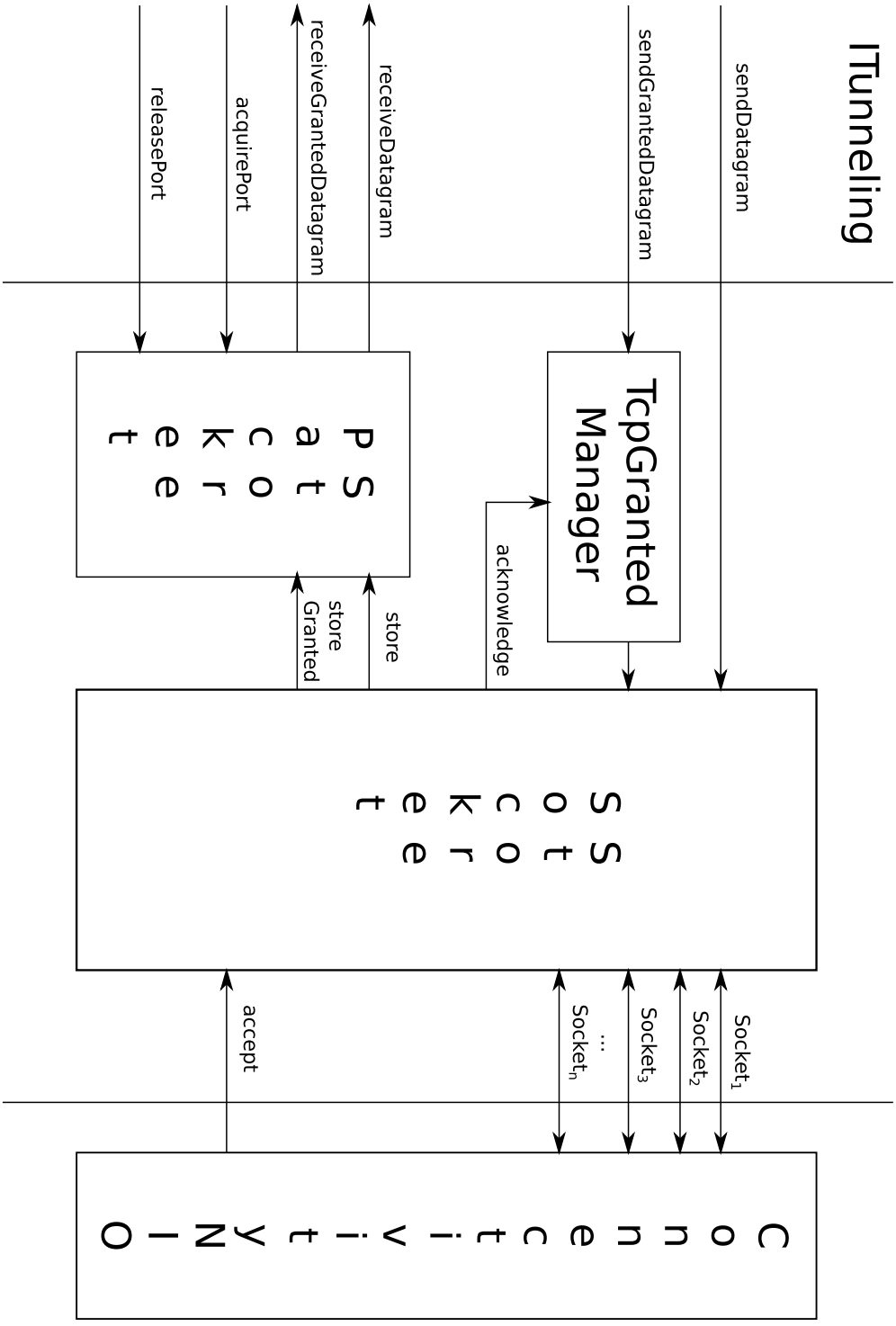

Figura 3.9: Schema tunnel via TCP.

3.5 Socket

L'unico modo sino a qui illustrato per eseguire comunicazioni attraverso il tunnel è l'uso di una realizzazione dell'interfaccia `ITunneling`: `Tunneling`. Tuttavia non è possibile consegnare un riferimento alla classe `Tunneling` all'utente programmatore, i motivi principali sono due:

- l'interfaccia contiene chiamate a basso livello, il cui uso può confondere il programmatore ignaro dei meccanismi di funzionamento di base del tunnel.
- sicurezza: attraverso l'interfaccia `ITunneling` chiunque può scrivere e leggere qualsiasi socket virtuale anche senza esserne il proprietario.

L'analisi dei due problemi porta alla stessa soluzione, cioè alla necessità di una classe con un'interfaccia d'uso semplice ed intuitiva, la quale nasconda e controlli le chiamate ad `ITunneling`, senza però diminuire le funzionalità concesse dal tunnel.
Il lavoro di sviluppo ha condotto alla creazione di due nuove classi: `PariPari DatagramSocket` e `PariPariGrantedSocket`. Entrambe funzionano allo stesso modo, si differenziano solo per il protocollo per cui fanno da tramite, e verranno fornite all'esterno di ConnectivityNIO come API. La loro struttura ricalca quella della classe `DatagramSocket` di `java.net` e quindi non dovrebbe spaventare i programmatori esterni che andranno ad utilizzare i socket virtuali.
La creazione delle istanze delle due classi può avvenire con il solo costruttore:

- `PariPariDatagramSocket(ITunneling tun,int port);`
- `PariPariGrantedSocket(ITunneling tun,int port);`

Ciò è possibile solo all'interno di ConnectivityNIO, l'unico plugin ad avere un riferimento ad una realizzazione di `ITunneling` valida. All'utente resta da specificare il numero di porta virtuale desiderato oppure il valore 0, il quale indica un qualsiasi numero di porta libero. Il costruttore provvederà alla

chiamata del metodo `acquirePort` specificando se stesso come proprietario del socket. Nel caso la creazione non vada a buon fine, come nel caso in cui il numero di porta risulti già riservato, viene lanciata un eccezione di tipo `BindException`.
La chiusura del socket, e quindi il rilascio delle risorse allocate mediante la chiamata di `releasePort` di `ITunneling`, è delegata al programmatore attraverso il metodo:

- `void close();`

L'invio dei datagrammi con `PariPariDatagramSocket` e `PariPariGrantedSocket` è possibile chiamando i metodi:

- `void send(PariPariDatagram pack);`
- `void send(PariPariGrantedDatagram pack);`

Il metodo `send` è non bloccante ed incapsula le chiamate ai metodi `send` di `ITunneling`.
In modo del tutto analogo all'operazione di invio, anche quella di ricezione dei datagrammi è semplice ed avviene con i metodi:

- `PariPariDatagram receive();`
- `PariPariGrantedDatagram receive();`

Il funzionamento è simile a quello degli altri metodi `PariPariDatagramSocket` e `PariPariGrantedSocket`, infatti si limita a racchiudere le chiamate ai metodi `receive` di `ITunneling`. Ciò nonostante, il metodo `receive` ha un comportamento completamente configurabile dall'utente. Operando sul metodo `setTimeout` si può variare il funzionamento di `receive`:

- `timeout = 0`: `receive` si blocca per un tempo indefinito in attesa di un datagramma.
- `timeout` $>$ `0`: `receive` si blocca per un tempo pari a `timeout` in attesa di un datagramma, se l'intervallo di attesa scade viene restituito un riferimento `null`.

- `timeout` $<$ `0`: `receive` ha un comportamento totalmente non bloccante. Se è disponibile restituisce un datagramma, altrimenti un riferimento `null`.

Le dinamiche bloccanti sono possibili poichè nella classe `PacketStore`, quando avviene l'inserimento di un nuovo datagramma, viene notificato l'avvenimento all'oggetto proprietario del numero di porta. Se il proprietario è in attesa viene risvegliato con la certezza di avere almeno un datagramma disponibile da leggere.

3.6 PPIDResolver

L'ultima componente mancante per assolvere tutte le funzionalità richieste dal tunnel è quella che permette al modulo tunneling ed al modulo nat traversal di cooperare. Ogni `PariPariDatagram` e `PariPariGrantedDatagram` inviato dall'utente o ricevuto dal tunnel passa attraverso un'istanza della classe `PPIDResolver`. Il passaggio dei datagrammi in `PPIDResolver` può essere schematizzato in due flussi, uno in uscita ed uno in ingresso.
Il flusso in uscita è quello cardine nell'integrazione tra tunnel e nat traversal; prima di consegnare i datagrammi ai moduli di invio, vengono passati nel metodo:

- `PariPariDatagram resolveOutgoing(PariPariDatagram pack);`

Il metodo ha due funzionalità, la prima è quella di tenere una cache di `PPID` completi, ovvero quelli reperiti tramite la rete DHT provvisti di tutti i campi necessari ad instaurare una connessione con l'host a cui si riferiscono. La seconda funzionalità è quella che chiama in gioco il modulo Nat Traversal; viene ispezionato il contenuto del `PPID` del datagramma, ed in base al tipo di rete in cui operano i due host (quello locale e quello remoto), avvia le procedure di attraversamento dei Nat.
Si possono verificare tre casi:

- Connessione diretta: i due host non necessitano di procedure per l'attraversamento dei Nat; il metodo `resolveOutgoing`, se necessario, aggiorna la cache dei `PPID` e restituisce il datagramma invariato.
- Connessione con Nat Traversal: la comunicazione tra gli host non è possibile, sono necessarie procedure di Nat Traversal come UDP Hole Punching o TURN. Viene avviata la procedura del caso e viene tenuta traccia del successo o meno nell'instaurare la connessione. A questo punto il metodo restituisce il datagramma in caso di successo od un eccezione in caso di insuccesso.
- Nat Traversal già avvenuto: è già avvenuta una procedura di attraversamento dei Nat tra i due host ed è ancora valida. Il caso viene gestito come una "Connessione diretta".

Nei casi in cui si necessitino dei servizi di attraversamento dei Nat, la trattazione è volutamente concisa ed ad alto livello per due ragioni. La prima è dovuta al fatto che la spiegazione del funzionamento del modulo Nat Traversal e delle sue casistiche esulano dall'argomento trattato da questo documento, perciò si rimanda a [21]. La seconda motivazione è causata dal fatto che la classe `PPIDResolver`, al momento della stesura di questo documento, non è ancora completa. Ciò è dovuto alla mancanza di sufficienti collaudi nell'attraversamento dei Nat, infatti benchè la teoria ed il codice di Nat Traversal siano sufficientemente completi, la pratica introduce difficoltà aggiuntive. La realizzazione dei dispositivi Nat è molto variabile da produttore a produttore, o addirittura da modello a modello, e porta alla luce casi comportamentali ibridi non contemplati dalla teoria individuabili solo mediante il collaudo di Nat Traversal su un buon numero di dispositivi.
La trattazione del funzionamento di `PPIDResolver` si conclude con la spiegazione di ciò che avviene nel flusso in entrata attraversando il metodo:

- `PariPariDatagram resolveIncoming(PariPariDatagram pack);`

Il datagramma viene fatto passare attraverso il metodo `resolveIncoming` giusto prima di essere restituito all'utente finale. L'unica operazione con-

dotta nel flusso in ingresso è quella di completare i `PPID` dei datagrammi. Infatti i `PPID` generati dai moduli di ricezione contengono un insieme di informazioni minime, le uniche a disposizione al momento della ricezione di un datagramma dal canale UDP/TCP: Indirizzo IP, numero di porta di trasporto (TCP o UDP), numero di porta virtuale. Viene ricercato un `PPID` completo nella cache all'interno di `PPIDResolver`, il quale viene sostituito nel `PariPariDatagram` a quello generato dai moduli di ricezione. Ciò è sempre possibile quando l'host locale ha iniziato la comunicazione con l'host remoto; al contrario, nel caso di ricezione di datagrammi da host sconosciuti, il `PPID` generato in ricezione rimane invariato. In teoria l'operazione di completamento dei `PPID` non sarebbe necessaria, le informazioni provenienti dai moduli di ricezione sono, nella quasi totalità dei casi, sufficienti per garantire la possibilità di rispondere all'host remoto; tuttavia avere `PPID` quanto più possibile completi, può essere di aiuto nell'identificazione degli host remoti e quindi nello smistamento dei datagrammi ricevuti da un socket.

Capitolo 4

Futuri sviluppi

Il primo passaggio fondamentale nel futuro del tunnel è un collaudo intensivo. L'utilità di questa fase è indubbia poichè permette di verificare il corretto comportamento del software sopratutto in casi limite, come quando si opera in una rete congestionata o dalle basse prestazioni in termini di banda e latenza. Ciò può portare ad un miglioramento di alcuni parametri sparsi tra le varie classi Java che formano il tunnel, come ad esempio tutti i parametri di timeout, i quali, al momento della stesura di questo documento, sono lasciati volutamente laschi per essere sicuri che eventuali malfunzionamenti del tunnel non siano dovuti a valori di timeout troppo vincolanti.
Ottenuta una buona stabilità del codice, uno degli sviluppi possibili e di maggiore interesse è senz'altro il miglioramento del protocollo PariPari garantito. Attualmente il protocollo è ridotto all'osso ed è funzionante, tuttavia delle migliorie nelle prestazioni e nel protocollo sono un obbiettivo da perseguire. Le prestazioni potrebbero essere migliorate realizzando, nei moduli di invio e ricezione, degli algoritmi di controllo del flusso e di congestione tipici del protocollo TCP.
Il protocollo andrebbe migliorato anche in robustezza, affiancando ai datagrammi di trasporto dei dati alcuni pacchetti per il controllo dello stato della connessione.
Un'altra possibile miglioria attuabile nel modulo tunnel è l'introduzione di

socket esclusivamente non bloccanti con la possibilità di specificare un oggetto simile a `PluginNotification` creando un socket dal funzionamento somigliante ai socket non bloccanti TCP di ConnectivityNIO.

Bibliografia

[1] Redmine. http://www.redmine.org/.

[2] Kent Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.

[3] Michele Bonazza. Paricore, 2009.

[4] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT), May 1994.

[5] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. *CoRR*, abs/cs/0603074, 2006.

[6] Apache Software Foundation. Subversion. http://subversion.apache.org/.

[7] Eclipse Foundation. Eclipse ide. http://www.eclipse.org/.

[8] PariPari Group. Paripari wiki. http://paripari.it/mediawiki/index.php/Main_Page.

[9] Ron Hitchens. *Java NIO*. O'Reilly & Associates, Inc., pub-ORA:adr, 2002.

[10] Geoff Huston. Anatomy: A look inside network address translators. http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_7-3/anatomy.html.

[11] BitTorrent Inc. Official site. `http://www.bittorrent.com/`.

[12] John C. Klensin. Simple mail transfer protocol. Internet RFC 5321, October 2008.

[13] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010.

[14] Maymounkov and Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric, 2002.

[15] P. Mockapetris. Domain names - implementation and specification, November 1987. RFC 1035.

[16] Network Working Group. RFC 5389 - Session Traversal Utilities for NAT (STUN). Technical report, IETF, October 2008.

[17] Jarkko Oikarinen and Darren P. Reed. Internet relay chat protocol. Internet RFC 1459, May 1993.

[18] Oracle. Java platform, standard edition 6 api specification. `http://docs.oracle.com/javase/6/docs/api/`.

[19] Oracle. Java web start. `http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/`.

[20] David A. Padua, editor. *Distributed Hash Table (DHT)*. Springer, 2011.

[21] Francesco Peruch. Paripari: Connectivity optimization, 2010-2011.

[22] Larry L. Peterson and Bruce S. Davie. *Computer networks - a systems approach (3. ed.)*. Morgan Kaufmann, 2003.

[23] J. Postel. Transmission control protocol. Technical Report RFC 793, DARPA, September 1980.

[24] Jon B. Postel. User datagram protocol. Internet RFC 768, August 1980.

[25] Emule project. Official site. `http://www.emule-project.net`.

[26] Wikipedia. Application programming interface. `http://en.wikipedia.org/wiki/Api`.

[27] Wikipedia. Peer-to-peer. `http://en.wikipedia.org/wiki/Peer-to-peer`.

[28] Wikipedia. Web server. `http://en.wikipedia.org/wiki/Web_server`.

Elenco delle figure

Elenco delle tabelle