PARITORRENT: PERFORMANCE REFACTORING

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Andrea Aldegheri

Corso di laurea in Ingegneria Informatica

Padova, 20 Aprile 2010
A.A. 2009-2010

UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

PARITORRENT: PERFORMANCE REFACTORING

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

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Andrea Aldegheri

Padova, 20 Aprile 2010
A.A. 2009-2010

Alla mia famiglia
per il loro supporto

Ai miei amici
per il loro sostegno

Indice

Introduzione

Questa tesi presenta il lavoro di refactoring fatto al plugin Torrent del progetto PariPari.

L'intervento è stato programmato al fine di limitare le risorse usate dal plugin nella sua esecuzione e semplificare la mantenibilità del codice. Una cosa essenziale per questo lavoro è stata l'adozione della libreria java.nio che ha permesso una gestione migliore delle connessioni.

Il primo capitolo va a fornire un'ottica di base sul progetto PariPari trattandone le caratteristiche peculiari, la struttura nonché lo stile di programmazione adottato.

Il capitolo successivo ha il compito di fornire una trattazione teorica del funzionamento del protocollo Torrent e delle estensioni esistenti che il nostro plugin supporta.

Il terzo capitolo presenta le funzionalità che la libreria java.nio mette a disposizione e i vantaggi rispetto alla libreria java.io.

Successivamente si va a discutere il refactoring spiegandone l'importanza e dando una traccia sui lavori principali che questa operazione comporta.

Il capitolo cinque va a illustrare il funzionamento del plugin Torrent precedente al lavoro fatto e ne illustra alcuni problemi che vi erano.

Gli ultimi due capitoli spiegano il nuovo funzionamento che si è ottenuto tramite il refactoring del codice, riportano alcune scelte di sviluppo fatte e presentano un confrontano di performance rilevate nel download di uno stesso file torrent.

Capitolo 1

Il progetto PariPari

PariPari nasce dall'idea di avere una piattaforma unica per fornire servizi disponibili in Internet. Questo per cercare di ottenere un sistema unico in grado di gestire al meglio le risorse disponibili. Il progetto di questo applicativo è stato presentato da Paolo Bertasi in [1] dove viene descritto come una rete serverless di tipo DHT[1] basata su Kademlia [2] in grado di garantire l'anonimato dei nodi e un sistema di crediti multifunzionale.

Ad oggi PariPari è un progetto che viene sviluppato da più di 60 studenti dell'Università degli Studi di Padova e punta a formare una folta comunità di sviluppo al suo rilascio ufficiale. PariPari, infatti, è anche una piattaforma per sviluppatori in quanto mette a disposizione una serie di API[3] necessarie all'implementazione di nuovi plugin atti a fornire servizi. L'attuale gruppo di lavoro si suddivide in gruppi, uno per plugin. Ogni gruppo ha un coordinatore, il *team leader*[4] e l'intero progetto è guidato da un *project manager*[5], l'architetto Paolo Bertasi.

Lo sviluppo di questo programma è fatto in linguaggio Java per favorirne la diffusione dato la mancanza della necessità di ricompilare il codice. Proprio questo aspetto che favorisce la compatibilità del programma, comporta il problema dell'impossibilità di andare a gestire alcuni aspetti a basso livello del dispositivo su cui si esegue e una leggera riduzione in termini di prestazioni. PariPari presenta un'architettura a plugin con un core centrale che ne permette l'esecuzione e gestisce le comunicazioni. Una simile architettura rappresenta un punto di forza

[1] Tabelle di Hash Distribuite.
[2] Kademlia, protocollo peer-to-peer.
[3] Application Programming Interface, insieme di procedure disponibili al programmatore.
[4] figura che guida, istruisce, direziona e comanda un gruppo di individui di un team.
[5] responsabile unico della valutazione, pianificazione, realizzazione e controllo di un progetto.

dato che rende possibile l'aggiunta di nuovi plugin senza andare a toccare il resto del programma.

1.1 Le Peculiarità

Abbiamo accennato alle caratteristiche di PariPari nella predente introduzione, ora vediamole in dettaglio.

- PariPari è una rete serveless: questa prima caratteristica denota la mancanza di nodi privilegiati o centrali. Tale caratteristica rende PariPari robusto a eventuali attacchi *DoS*[6] dato che questi si basano sull'inibire l'accesso ai nodi fornitori di servizi della rete.

- PariPari ha una struttura modulare: attorno al nucleo centrale (il Core) sono presenti una serie di plugin atti a svolgere e fornire determinati servizi. Si possono suddividere tali plugin in due categorie: i plugin della cerchia interna che permettono di gestire le risorse a disposizione quali, per esempio, la banda a disposizione o lo spazio su disco e i plugin esterni che usano le risorse tramite i plugin del primo tipo e si interfacciano con l'utente finale. Per meglio far capire la struttura è riportata in Figura 1.1. Cosa da sottolineare è il fatto che i vari plugin possono comunicare tra di loro esclusivamente tramite il core ossia indirettamente.

- Sistema di crediti: PariPari si basa su un sistema di crediti che regola le comunicazioni fra nodi della rete e fra plugin nello stesso host. Sostanzialmente si ha una specie di sistema economico dove un plugin "paga" per avere un servizio e "viene pagato" qualora svolga servizi per altri. Allo stato attuale esiste la distinzione tra i crediti esterni, ossia tra nodi PariPari, e i crediti interni al nodo.

- Single-Click-Launch: PariPari è stata sviluppata secondo la tecnologia *Java Web Start(JWS)*, la quale permette di lanciare il nostro applicativo direttamente da un Browser Web con un semplicissimo click. Questo è possibile grazie al fatto che non è necessario installare PariPari prima di poterlo usare.

[6] attacco con il quale porta il funzionamento di un sistema informatico che fornisce un servizio al limite delle prestazioni fino a renderlo non più in grado di erogare il servizio.

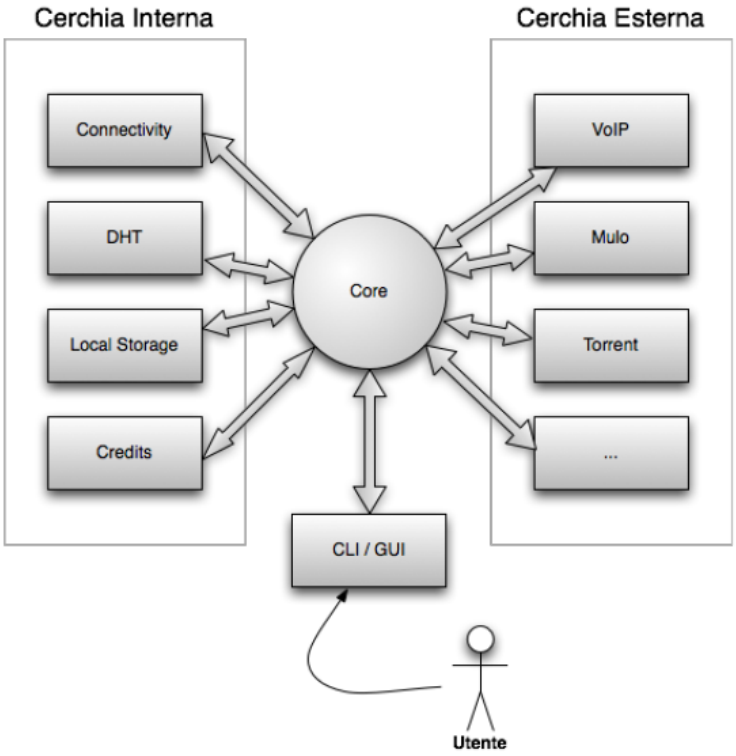

Figura 1.1: La struttura a plugin

1.2 L'Architettura

Scendiamo ora in dettaglio nel vedere i componenti base dell'architettura a plugin e spiegarne velocemente le interazioni fra questi.

1.2.1 Il Core

Innanzitutto presentiamo il nucleo di PariPari, il Core. Questo era inizialmente pensato come un componente che si occupasse di caricare i plugin e gestire i messaggi. Tuttavia, con lo svilupparsi del progetto, si è giunti a trasformarlo in un componente molto più complesso che, oltre a smistare i messaggi, si occupa di avviare, fermare o riavviare in modo dinamico i vari plugin, di gestire gli aggiornamenti disponibili ai plugin, di fornire un'interfaccia grafica, di definire un gestore della sicurezza, di controllare i vari thread e molto altro.

La parte di scambio messaggi è grossomodo la stessa che è stata implementata all'inizio del progetto data la sua semplicità, efficenza e robustezza. Uno schema

di come avviene l'instradamento dei messaggi è riportato in figura 1.2 dove si possono identificare i due plugin A e B e uno schema degli oggetti del core che vengono presi in causa. Nell'esempio si presenta il caso in cui il plugin A mandi un messaggio a B.

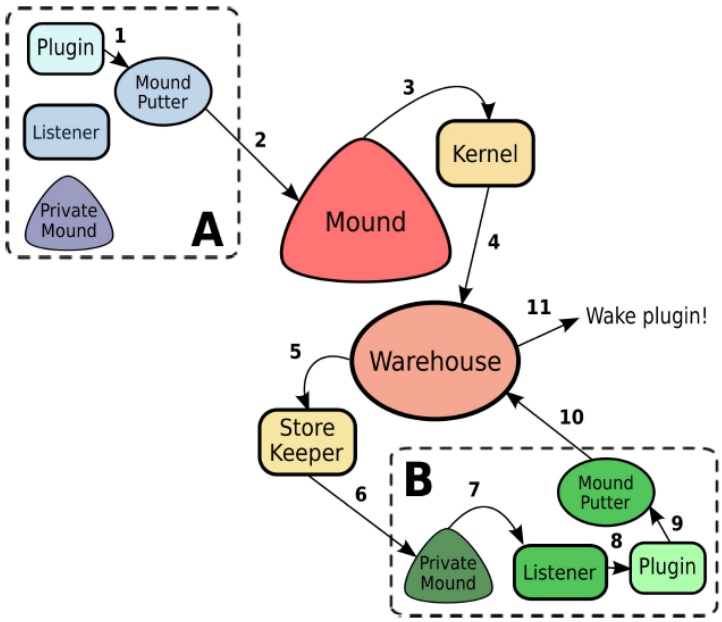

Figura 1.2: Gestione Messaggi

Brevemente, il plugin A crea un thread che si occupa di mandare il messaggio all'oggetto `MoundPutter` il quale, a sua volta, lo inserisce nella coda globale e mette in "pausa" il thread . A questo punto il `Kernel` del core serve le richieste una ad una ordinandole sulla base di ciò che sancisce il sistema di crediti. Durante la fase di completamento delle richieste, queste sono piazzate nell'oggetto Warehouse il quale tiene traccia dell'avanzamento e delle risorse già acquisite dalla richiesta. La richiesta, una volta in fase di evasione, viene passata all'oggetto Store Keeper il quale si occupa di reindirizzarla al plugin che la può soddisfare. La struttura interna per la ricezione dei messaggi consiste sempre di una coda e un oggetto che la gestisce. I messaggi di risposta ad una richiesta possiedono informazioni sulle risorse utilizzate o prodotte, sulla correttezza della richiesta e molto altro. Questi messaggi vengono mandati all'oggetto Warehouse il quale, nel momento in cui rileva che le risorse per una determinata richiesta sono tutte disponibili, risveglia il thread che ha fatto la richiesta. Questo, dopo aver controllato la risposta, comunica al sistema di crediti come si è comportato il plu-

gin che ha risposto alla richiesta. Il sistema di crediti, infatti, si basa sui feedback.

Il fatto che non sono possibili le comunicazioni dirette fra plugin ha due sostanziali vantaggi:

- Viene impedito a plugin malevoli di recuperare informazioni sulla gestione dei crediti tramite richieste fasulle;
- Permette di gestire meglio le risorse, aumentando o diminuendo l'attesa delle richieste.

Secondo compito essenziale del core è il caricamento dei plugin il cui processo è molto semplice e riassumibile in poche fasi. In primo luogo viene istanziato un ClassLoader a cui sono passate tutte le classi del plugin, successivamente si creano gli oggetti per la comunicazione con il Core e viene fatto partire il thread principale del plugin dopo averne inizializzato la classe con gli oggetti per la comunicazione appena creati.

Da notare che al caricamento di un plugin il Core controlla se in questo sono state dichiarate dipendenze da altri plugin e, in tal caso, il Core stesso si occupa di rendere disponibili questi plugin prima che lo stesso ne richieda i servizi. Questo aspetto viene garantito tramite un grafo delle dipendenze che ogni plugin contiene nei file di configurazione.

1.2.2 I Plugin

Viste come sono svolte le principali funzioni del Core, andiamo a descrivere velocemente la struttura dei plugin. Come illustrato da Bonazza in [2], lo sviluppo di una struttura è stata un'operazione molto complessa e problematica. Per semplificarne lo sviluppo, esiste il package paripari.plugin che contiene alcune classi astratte con i metodi necessari per mandare e ricevere messaggi fra plugin. Questo pacchetto contiene le classi:

- Plugin, rappresenta la classe principale e al caricamento del plugin viene chiamato il metodo `init()`;
- Pluginthread è il thread che fa richieste al core;
- Listener è il thread che sta in ascolto per messaggi in arrivo dal core;

- ConstructoParameters che rappresenta i parametri che devono essere passati ai costruttori degli oggetti che si richiedono ad altri plugin.

Altra cosa da riportare è che, per poter caricare un plugin, non è sufficiente produrre e compilare il codice. Tutte le classi e le risorse necessarie devono essere inserite in un archivio jar con un descrittore *XML*[7] del plugin.

Questo file di descrizione, oltre a contenere informazioni come il nome del plugin, la versione e le eventuali dipendenze, specifica anche di quali API necessita e quali implementa e altre infromazioni necessarie al sistema di crediti.

Ultimo passo suggerito e vivamente consigliato consiste nel firmare l'archivio jar creato. Tale operazione, infatti, permette di impedirne la modifica, di avere una cartella per il plugin nella directory principale e di rendere disponibile il controllo di aggiornamenti del plugin stesso sul server degli aggiornamenti PariPari. Per il processo di firma al momento abbiamo un nostro KeyStore che mantiene i certificati sui plugin di PariPari.

1.2.3 La Rete

PariPari si basa su una rete DHT che abbiamo chiamato PariDHT. Nonostante ci siano molti modi di implementare una DHT, essenzialmente si ha una struttura base di fondo. In particolare ogni nodo in una rete ha assegnato un indirizzo casuale in uno spazio di indirizzamento a b-bit (solitamente b è 160 o più per evitare collisioni).

In questo tipo di rete si ha una partizione dello spazio degli indirizzi per cui un nodo riesce a sapere la distanza da un altro in base al suo indirizzo. In particolare, se noi mappiamo i nodi della rete ordinatamente su una circonferenza, avremo che un nodo saprà se il nodo che deve raggiungere è nell'altra metà della rete oppure nella sua metà. Nella seconda ipotesi saprà se è nel suo quarto o meno e così via, dividendo sempre di un fattore due.

Per un aspetto di robustezza della rete, ogni nodo rimane in contatto con un gruppo di nodi della rete.

[7] eXtensible Markup Language, metalinguaggio di markup.

Ogni risorsa presente in queste reti viene mappata nella rete ad un indirizzo tramite un *hash*[8] delle parole chiave che vengono usate per localizzarla. Ovviamente le informazioni che vengono salvate si riferiscono a come recuparare la risorsa e, per non perdere il collegamento alla risorsa, queste informazioni vengono replicate anche sui nodi vicini.

A questo punto un nodo che ricerca una risorsa non fa null'altro se non calcolare l'hash delle parole chiave usate nella ricerca e contattare il nodo che dovrebbe avere le informazioni.

1.3 L'eXtreme Programming

L'*eXtreme Programming* è una metodologia di programmazione agile che permette di organizzare le persone per incrementarne sia la produttività delle stesse che la qualità del codice sviluppato. Tale metodologia fu formulata da Kent Beck, Ward Cunningham e Ron Jeffries[10] ed era volta a sovvertire il normale procedimento di produzione del codice in ambito aziendale, considerato troppo rigido e incapace di produrre in larga scala codice di qualità.

Questa metodologia prevede, infatti, la verifica continua del codice prodotto attraverso l'utilizzo di test, la programmazione in coppia e la frequente reingegnerizzazione del software prodotto senza rispettare particolari cicli di sviluppo. Pur non essendo un ambito aziendale questo tipo di modello è particolarmente utile allo sviluppo di PariPari in quanto permette di ottenere un prodotto funzionante in tempi relativamente brevi e di continuare a migliorarlo senza dover rispettare i cicli standard.

Altro aspetto rilevante è che, mentre nei sistemi di sviluppo tradizionali si fissano i requisiti all'inizio del lavoro e eventuali cambiamenti postumi sono costosi, con XP si hanno tanti cicli di sviluppo brevi che riducono i costi di successive modifiche. In questa dottrina, infatti, i cambiamenti sono una cosa normale e devono essere pianificati invece di cercare di definire tutti i requisiti dall'inizio.

Nell'approccio del lavoro del nostro gruppo, abbiamo previsto che ogni individuo allo stesso tempo debba scrivere la parte di codice relativa alle funzionalità da implementare assegnategli e i test sulle funzioni che un altro membro del suo gruppo deve implementare (questa tecnica è nota come Pair Programming).

[8]funzione non iniettiva che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita.

Lo sviluppo di PariPari, in particolare, segue una delle tecniche suggerite dall'XP chiamata Test-Driven Development nella quale il testing ricopre un ruolo principale.

Lo sviluppo basato sui test permette di aumentare la qualità del progetto in quanto i test definiscono il comportamento atteso del software e possono venire interpretati come specifiche. Questo approccio, inoltre, permette di facilitare le operazioni di *refactoring* del codice prevenendo l'introduzione di comportamenti errati.

Va tenuto presente che, per la produzione di software con questa dottrina di sviluppo, bisogna produrre classi con alcune proprietà specifiche per favorire lo sviluppo di test semplici. In particolare bisogna cercare di strutturare il software con classi che sono vere e proprie librerie con metodi indipendenti che eseguono le varie operazioni e altre classi più semplici che si occupano di chiamare i metodi dalle librerie e di fare poche operazioni elementari.

Capitolo 2

Il protocollo Torrent

La nascita del protocollo BitTorrent è avvenuta nella primavera del 2001 ad opera di Bram Cohen, il quale ha sviluppato un client funzionante già pochi mesi più tardi. La più grande innovazione che ha portato nei sistemi di file sharing è data dalla mancanza di un'architettura basata sui server (cosa che invece si ha, ad esempio, in napster e ed2k).

Uno degli obiettivi che si prefissa tale protocollo è la condivisione dei file di grandi dimensioni i quali, nelle reti basate su server, andavano a caricare di lavoro i server.

In poche parole tale protocollo vuole fornire un meccanismo per coordinare in automatico il lavoro di più computer per ottenere il massimo beneficio possibile per i partecipanti.

2.1 Il protocollo BitTorrent

Un peer che voglia scaricare tramite questo protocollo, oltre ad avere il software, deve seguire pochi passi:

- Ottenere un file .torrent che si riferisce al contenuto desiderato dall'autore dello stesso oppure tramite motori di ricerca sul web o ancora tramite dei BitTorrent index;
- Aprire il file con un client BitTorrent per connettersi ai tracker collegati;
- Procedere con il download, collegandosi ad altri peer o seeder;
- A download completato rimanere nella rete come seeder (comportamento opzionale ma di buona norma per la salute della rete).

In un rapido confronto con altre reti p2p si può notare come l'utente abbia più lavoro da fare anche se questo viene ripagato da delle performance superiori in fase di download.
Andiamo ora a illustrare in cosa consistono i file torrent e i tracker.

2.1.1 I file .torrent

In questo sistema tale file ha la funzione di indice e contiene la descrizione di tutti i pacchetti in cui è suddivisa la risorsa desiderata, comprendendo le chiavi di hash che vengono usate per verificare l'integrità dei pezzi che si vanno a scaricare.
Possiamo dunque affermare che il file .torrent è una risorsa fondamentale per la condivisione tramite questo sistema peer-to-peer.

I dati all'interno di questo file sono mantenuti organizzati tramite la codifica Bencode. Questa codifica supporta solamente quattro tipi di dato, ossia gli interi, le stringhe di byte, le liste e i dizionari. Una descrizione più accurata con esempi si può trovare in [3]. Una caratteristica fondamentale di questa codifica è la possibilità di annidare all'interno di liste e dizionari altre strutture. Questo permette, di conseguenza, di introdurre in una lista un numero indefinito di dizionari e liste senza avere alcuna restrinzione.

Tornando alla struttura del file, questo al suo interno può avere le seguenti informazioni:

- announce (*stringa*): indirizzo URL del tracker codificato come stringa ASCII;
- announce-list (*lista*): estensione del protocollo verso il Multitracker. Il dizionario announce è comunque richiesto per retrocompatibilità;
- creation date (*intero*): data di creazione codificata come timestamp Unix;
- comment (*stringa*): un commento al Torrent codificato come stringa ASCII;
- created by (*stringa*): programma che ha generato il Torrent;
- info (*dizionario*): dizionario principale che descrive il contenuto del Torrent.

Poi contiene informazioni sulle risorse a cui si riferisce e nel caso sia un singolo file si hanno i campi:

- length (*intero*): dimensione del file in byte;
- md5sum (*stringa*): impronta MD5 del file per migliore identificazione;

- name (*stringa*): il nome del file in formato ASCII (i nomi UTF-8 e Unicode generano problemi di codifica);

Al contrario, se il file .torrent si riferisce a più file, si ha che nella codifica bencode è presente una lista di dizionari, dove ogni dizionario rappresenta un file e contiene le seguenti informazioni:

- length (*intero*): dimensione del file in byte;
- md5sum (*stringa*): impronta MD5 del file per migliore identificazione;
- path (*lista*): lista di stringhe che permette di ricostruire il percorso del file prendendo gli elementi del loro ordine; l'ultimo elemento di questa lista rappresenta nome e estensione del file (esempio: "Dir1", "Dir2", "File.ext" rappresenta Dir1/Dir2/File.ext);
- piece length (*integer*): lunghezza in byte di ogni parte in cui è suddiviso il(i) file;
- pieces (*stringa*): stringa che concatena le fingerprint SHA1 del/i file in formato ASCII. Se in fase di creazione la stringa non ha lunghezza multipla di 20, essa viene completata con zeri.

Si noti come non tutte queste informazioni siano obbligatorie, bensì solo announce e info debbano essere presenti e, per la parte riguardante il file, la stringa md5 è l'unica cosa opzionale.

2.1.2 I Tracker

Essenzialmente il tracker non è altro che un normale webserver HTTP o HTTPS che risponde a richieste GET codificate tramite i metodi *CGI*[1], ossia aggiungendo dopo l'URL *(Uniform Resource Locator)* un ? e la sequenza di coppie di parametri e valori nella forma p̈arametro=valoreës̈eparate da&. Un esempio è:
http://pippo.franco.gigio:39604/announce?info_hash=%B0%5D%93%1DD%E1
%C8wV%94%40%DEq%CCaW%D7%2E%87%A4&peer_id=-PP0001-%EE%87
%9C%9A%95D%AC%CE%81%AF%ADu&port=7881&downloaded=0
&uploaded=0&left=47252336&numwant=50&compact=1&event=started

[1]Common GatewayInterface, tecnologia usata dai web server per interfacciarsi con applicazioni esterne.

Questo server web risponde a queste richieste con un file di testo contenente le informazioni codificate tramite Bencode.

Il client nel richiedere deve fornire una serie di parametri al tracker, come al solito alcuni obbligatori e altri facoltativi:

- info_hash (obbligatorio): hash SHA1 del dizionario info codificato in Bencode, in formato stringa codificata secondo le convenzioni URL;
- peer_id (obbligatorio): una stringa di 20 caratteri che permette di identificare in maniera quasi-univoca l'utente sul tracker;
- port (obbligatorio): numero di porta sulla quale il client è in ascolto. Le porte tipiche sono nel range 6881-6900;
- uploaded (obbligatorio): byte inviati agli altri client dall'inizio della sessione, codificati in ASCII base-10;
- downloaded (obbligatorio): byte scaricati dagli altri client dall'inizio della sessione, codificati in ASCII base-10;
- left (obbligatorio): byte rimanenti al completamento del file, codificati in ASCII base-10. Il valore 0 indica un seeder;
- compact: indica al tracker di utilizzare il Compact Announce, per l'elenco dei peer;
- event (obbligatorio): può assumere i tre seguenti valori: started, stopped, completed. Il primo viene inviato a inizio sessione e indica al tracker che si sta iniziando una nuova sessione. Il secondo chiude la connessione con il tracker e chiede di essere rimosso dalla lista dei peer. L'ultimo comunica al tracker il completamento del download e il passaggio allo status di seeder;
- ip: indirizzo IP da comunicare agli altri peer. Solitamente utilizzato se si è dietro NAT/router;
- numwant: numero di fonti massimo che il tracker deve comunicare;
- key: stringa randomizzata per una migliore identificazione univoca del client;
- trackerid: se il tracker ha comunicato in precedenza un tracker id, esso va inviato qui.

A questo punto, il tracker elabora la richiesta e struttura il documento con la codifica Bencode inserendo i seguenti campi:

- failure_reason (*stringa*): se presente, rappresenta la condizione di errore in formato human-readable. In presenza di failure reason non devono essere presenti ulteriori elementi;

- warning_message (*stringa*): rappresenta una condizione di allerta in formato human-readable. L'elaborazione della risposta di Announce non viene interrotta e all'utente viene mostrato un messaggio descrittivo;

- min_interval (*intero*): tempo minimo in secondi che deve trascorrere tra due richieste Announce, pena il rifiuto da parte del server;

- interval (*intero*): tempo in secondi da raccomandare al client per gli intervalli tra gli aggiornamenti Announce, al fine di non sovraccaricare il server. Questo valore deve essere compreso tra il valore di min interval e il time-out per la sconnessione forzata dei peer morti;

- tracker_id (*stringa*): una stringa che identifichi univocamente il tracker;

- complete (*intero*): seeder attualmente connessi;

- incomplete (*intero*): leecher attualmente connessi;

Come altro campo sono presenti le informazioni sui peer. Nel caso il client che ha fatto la richiesta supporta la forma "Compact Announce" allora sarà presente una successione di stringhe rappresentanti ognuna un peer. Tali stringhe sono lunghe 6 byte e i primi 4 rappresentano l'indirizzo IP in formato numerico mentre i rimanenti due la porta.

Nel caso in cui questo formato non sia supportato, si trova una lista di dizionari in cui ogni dizionario contiente i dati necessari ad identificare e contattare un peer, ossia:

- peer_id (*stringa*): il valore che il peer ha fornito in fase di connessione;

- ip (*stringa*): l'indirizzo IP in formato IPv4, IPv6 o DNS;

- port (*intero*): numero di porta usato dal peer.

2.1.3 Peer Protocol

Dopo la prima parte di comunicazione con il tracker, il client può cominciare a contattare gli altri peer e, ovviamente, essere contattato da altri peer. La specifica ufficiale prevede di identificare due stati possibili per un peer: lo stato di *choked*, in questo caso il peer non invierà dati fino a quando non passerà in modalità *unchoked*, e lo stato di *interested* per indicare se il peer è interessato a qualche pezzo in nostro possesso. Ovviamente, il client, per ogni peer che contatterà salverà anche il proprio stato nei confronti del peer in questione, ossia se si stanno bloccando le comunicazioni con quel peer e l'eventuale interessamento a qualche pezzo che ha disponibile.

Le comunicazioni fra i peer avvengono tramite protocollo TCP e il primo messaggio che si scambiano i due peer è un *handshake*. Questo messaggio ha una struttura fissata in questo modo: `[pstrlen] [pstr] [reserved] [info_hash] [peer_id]` dove `pstrlen` indica la lunghezza in byte di `pstr` (solitamente 19 byte), il cui campo contiene una stringa che identifica il protocollo. Il campo `reserved` identifica le estensioni usate e contiene altri byte per implementazioni future; il campo `info_hash` è un hash *SHA-1*[2] del dizionario contenuto nel file .torrent ed è indispensabile per identificare univocamente il torrent a cui ci si riferisce e, per finire, il campo `peer_id` è una stringa da 20 byte che identifica il peer.

Terminata la fase iniziale di handshake, i successivi messaggi hanno la forma seguente: `[lunghezza] [ID_messaggio] [payload]`. In questo caso il campo `lunghezza` è di 4 byte e indica il numero di byte totali dei due campi successivi. Il campo `ID_messaggio` è un singolo byte (non presente nel caso di messaggio di keepalive) che indica il tipo di messaggio inviato e, come ultimo campo, è presente il `payload` che ha significati diversi in base al tipo di messaggio.

2.2 Le Estensioni

Oltre al protocollo originale di BitTorrent, si possono trovare alcune estensioni per migliorare la funzionalità di questo. Nella fattispecie sono presenti le librerie *libtorrent*[5] che sono state sviluppate da Arvid Norberg nel 2005 che, nate originariamente come implementazione in C++ del protocollo, forniscono funzionalità aggiuntive tra le quali la più importante è l'*extension protocol*[5] e il

[2]Secure Hash Algorithm, funzione crittografica di hash.

peer exchange[5]. La prima ha lo scopo di fornire uno strato di supporto per creare estensioni al protocollo base senza entrare in conflitto con questo e, di conseguenza, rimanere compatibili con client che non le supportano. La seconda funzionalità, abbreviata con *PEX*, è stata introdotta per facilitare la scoperta dei nuovi peer. Questa estensione, infatti, permette ai vari peer che sono in contatto di scambiarsi i peer a cui sono connessi e ha un effetto positivo anche sui tracker dato che ne allegerisce il carico di lavoro.

Altra invenzione che si è diffusa molto velocemente è la *PE* (Protocol Encryption) la quale ha la finalità di rendere il traffico degli handshake illegibile agli *ISP*[3]. Questa funzionalità venne introdotta inizialmente da BitComet nel 2005 ed era chiamata Protocol Header Encryption. La crittografia in questione sfrutta uno scambio chiavi combinato con il campo info_hash del torrent per creare una chiave *RC4*[4] condivisa e, successivamente, si può avere solamente lo scambio di handshake criptato oppure tutto lo scambio dati fra i due peer.

[3]Internet Service Provider, fornitore di servizi Internet.
[4]algoritmo di cifratura a flusso a chiave simmetrica

Capitolo 3

Java NIO

Sin dalla distribuzione 1.4 di *J2SE* (Java 2 Platform, Standard Edition) è presente nella libreria il package `java.nio`. Questo package, new I/O o non-blocking I/O che dir si voglia, è una collezione di API che offrono caratteristiche avanzate per l'input/output.

La "vecchia" libreria `java.io` rappresenta un insieme di classi molto potenti e flessibili ma che, avendo come base il concetto di Stream, non fornisce un completo supporto alle operazioni di I/O e non è più sufficente alle esigenze di sviluppo attuali.

Al giorno d'oggi, infatti, le prestazioni dei programmi sono un aspetto critico. I vari sistemi operativi e l'hardware sottostante continuano a evolversi per migliorare sempre più le prestazioni. Come già accennato, la JVM offre un ambiente di lavoro uniforme che evita al programmatore l'analisi delle differenze presenti fra sistemi operativi diversi. Questo semplifica ovviamente lo sviluppo ma nasconde alcune funzionalità che il sistema operativo mette a disposizione.

Ma allora cosa fare? Chiaramente uno sviluppatore potrebbe sfruttare le *JNI* (Java Native Interface) per scrivere codice e accedere direttamente al sistema operativo. Ma questo, oltre a rendere il programma dipendente dal sistema operativo, espone la JVM a possibili crash introdotti dal codice.

Il pacchetto `java.nio`, fornisce la soluzione. Le sue classi forniscono un modo di sfruttare le caratteristiche avanzate dei sistemi operativi rimanendo all'interno della macchina virtuale java.

Prima di addentrarci nelle classi che mette a disposizione, vediamo l'aspetto principale di `nio` ossia l'abilità di operare in modalità non bloccante. Questa funzionalità di I/O asincrono, preclusa con le classe tradizionali, permette l'ese-

cuzione di altri processi mentre sta terminando una fase di lettura/scrittura di un processo.

Considerando le velocità raggiunte nell'elaborazione dei dati dei computer, si può immediatamente notare quanto questa caratteristica sia fondamentale per cercare di non sprecare risorse in processi che rimangono in attesa di flussi I/O "lenti" (ossia avere processi bloccati).

Ad oggi è possibile informare il sistema operativo di monitorare una collezione di stream e informare il processo qualora uno di questi fosse pronto per operazioni di I/O. Questa caratteristica molto importante che va ad un livello superiore alla modalità non bloccante appena vista, è ripresa dalla classe `nio` e fornisce la possibilità di usare un singolo processo per monitorare un grande numero di connessioni di rete.

3.1 Caratteristiche

Questo package, ovviamente, non si limita a fornire classi per l'I/O ad alte prestazioni. Esso, infatti, mette a disposizione nuove strutture dati quali:

- Buffer per i dati di tipo primitivo;
- Character-set encoder e decoder;
- Un sistema di pattern-matching;
- Channel;
- Un'interfaccia per File che supporta i lock e la mappatura in memoria;
- Facilitazioni per scrivere server scalabili tramite I/O non bloccante.

Come il pacchetto java.io è progettato attorno al concetto di Stream, così le nuove classi di `nio` sono costruite attorno alle classi Buffer. Un oggetto di questo tipo rappresenta uno spazio di memoria contiguo con alcune operazioni di trasferimento dati. Ovviamente la parte innovativa sta nel fatto che si ha un meccanismo diretto per il trasferimento nella memoria fisica e, di conseguenza, vengono eliminati ulteriori costi di copiatura. Va tenuto presente che molti dei sistemi operativi attuali forniscono per questi scopi alcune porzioni particolari di memoria che sono leggibili o scrivibili senza necessitare della CPU.

Sono implementate classi di buffer per ogni tipo di dato primitivo in Java e tutte sono estensione alla classe java.nio.Buffer. Un buffer essenzialmente, oltre ad avere una certa dimensione, contiene un puntatore alla posizione attuale da leggere/scrivere, un limite che informa sulle posizioni usabili e un marcatore che può essere impostato dall'utente.

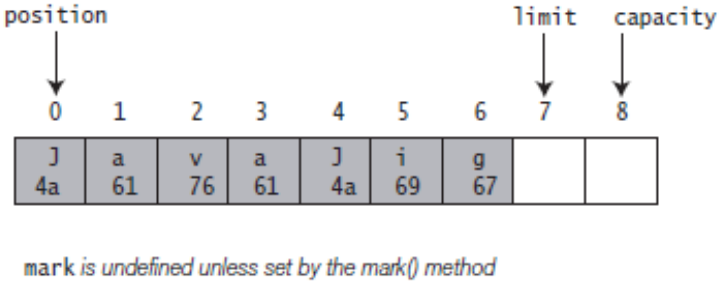

Figura 3.1: Esempio di ByteBuffer

Per quanto riguarda i metodi, invece, ecco i più importanti:

- `flip()`: metodo per passare dalla modalità scrittura alla modalità lettura; imposta il limite alla posizione attuale e la posizione all'inizio del buffer;
- `compact()`: questo metodo fa l'esatto opposto del precendente; in particolare sposta tutto il contenuto fra la posizione e il limite all'inizio del buffer e imposta la posizione al primo elemento libero e il limite alla capacità del buffer;
- `get()` e `put()`: metodi rispettivamente per leggere e scrivere nel buffer;
- `clear()`: pulisce il contenuto del buffer azzerando posizione e mettendo il limite alla capacità;

Un aspetto molto importante a cui bisogna far attenzione nell'usare i buffer è che questi non sono "thread-safe". Questo per dire che non c'è nessun controllo fatto dal sistema che impedisce a due thread di andare a leggere e scrivere in contemporanea e, di conseguenza, di andare a creare delle inconsistenze nei dati.

Ultima cosa da analizzare è la modalità di istanziamento di un buffer: questa operazione, infatti, non sfrutta la classica keyword `new` bensì avviene tramite la chiamata a un metodo statico della stessa classe che ritorna la nuova istanza.

Altra struttura introdotta in ordine di importanza sono i canali, Channel. Questi sono veri e propri portali tramite i quali avvengono i trasferimenti I/O e,

solitamente, i buffer rappresentano le sorgenti o le destinazioni dei dati. In figura 3.2 mostra come siano differenti gli approcci basati su stream rispetto a quelli basati su canali.

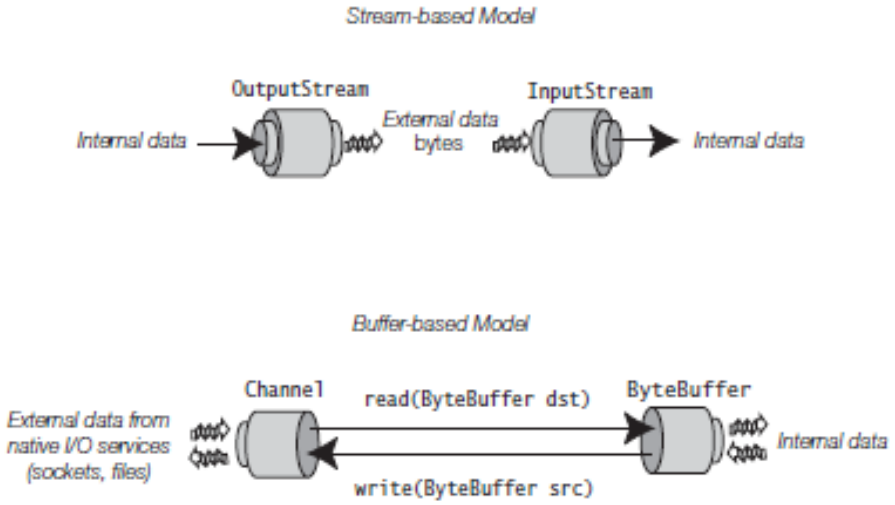

Figura 3.2: Confronto fra modello basato su Stream e su Channel

I canali si possono immaginare come veri e propri tubi tramite i quali è possibile accedere alle funzioni I/O del sistema operativo senza sovraccaricare di lavoro il sistema. Essendo che le funzioni di I/O si possono raggruppare in due categorie, file I/O e stream I/O, possiamo subito introdurre una distinzione fra la classe FileChannel che appartiene alla prima categoria e le classi ServerSocketChannel e SocketChannel e DatagramChannel che sono classi inerenti ai canali per socket.

Ci sono molte differenze nella creazione di canali file e canali socket perché, mentre i primi sono creati dalla chiamata al metodo `getChannel()` su un oggetto File, la creazione di canali per socket essistenti non è possibile. Questo tipo di canale, infatti, viene inizializzato con un socket della classe java.net; conseguentemente il metodo `getChannel()` degli oggetti socket di java.net ritorna il canale associato qualora presente.

È essenziale comprendere bene due aspetti dei canali che potrebbero trarre in inganno:

- un oggetto socket istanziato dalla classe java.net non ha nessun canale socket associato e, proprio per questo, l'invocazione del metodo `getChannel()` ritorna un valore `null`;

- il lock che la classe FileChannel mette a disposizione serve per coordinare l'accesso al file con i processi esterni alla JVM, non per sincronizzarne l'accesso fra thread eseguiti all'interno della stessa.

La caratteristica fondamentale che i canali possiedono, oltre alla possibilità di lavorare in modalità non bloccante per quelli di tipo stream I/O, è di essere "thread-safe" ossia usabili da più thread senza problemi di accessi simultanei che il programmatore deve gestire.

Ultima innovazione che andiamo a presentare risiede nella classe Selector. Questo oggetto, nella fattispecie, fornisce un meccanismo per aspettare determinati eventi su un canale e riconoscerli con prontezza qualora avvengano. In particolare viene permessa la registrazione di molti oggetti di tipo SelectableChannel in un singolo selettore e, l'invocazione del metodo `select()` del selettore, permette di bloccare il flusso del programma fino a quando almeno uno di questi non sia pronto per essere usato.

Ovviamente la classe Selector permette anche di specificare un tempo di attesa massimo tramite lo stesso metodo `select()` passando come argomento il *time-out* in formato `long` oppure di selezionare istantaneamente i canali che sono pronti tramite il metodo `selectNow()`.

L'attesa di un processo nei metodi `select()` può essere interrotta in tre modi:

1. dall'invocazione del metodo `wakeup()` dell'oggetto selettore;
2. dalla chiusura del selettore (tramite il suo metodo `close()`);
3. dalla chiamata al metodo `interrupt` del thread bloccato che, come effetto ulteriore, provoca la chiamata al metodo `wakeup()` del selettore.

Il processo di registrazione di un canale nel selettore produce un oggetto SelectionKey il quale mantiene la relazione fra il canale e il selettore e, soprattutto, mantiene le informazioni sulle operazioni per le quali il canale è in "ascolto".

In particolare un oggetto Selector contiene tre set di chiavi: uno per le chiavi registrate accessibile tramite il metodo `key()`, uno per le chiavi che sono pronte per una delle operazioni indicate e viene ritornato dal metodo `selectedKeys()`; uno per le chiavi la cui registrazione è stata cancellata ma che non sono ancora state deregistrate.

Il procedimento di pulizia delle chiavi selezionate è lasciato al programmatore, in modo da rendere possibile la rimozione della chiave nel momento in cui si è gestito l'evento che ha reso il canale pronto all'operazione di I/O. Successivi eventi, di conseguenza, rimodificheranno lo stato del canale e faranno reinserire la loro chiave nel gruppo delle chiavi selezionate.

In ottica multithreading si deve tener presente che, nonostante i selettori sono "thread-safe", i set di chiavi che sono contenuti in essi non lo sono. Conseguentemente è molto semplice andare incontro a eccezioni del tipo java.util.ConcurrentModificationException qualora il set venga modificato.

3.2 Vantaggi

I costrutti messi a disposizione da `java.nio` rendono molto più veloci le operazioni di input/output rispetto le classi tradizionali. Questo fa la differenza in programmi dove le operazioni di I/O rappresentano una parte significativa delle operazioni di elaborazioni.

In particolare questa libreria va a infrangere la più grande limitazione sulla scalabilità, ossia la natura bloccante dei socket. L'introduzione dell'I/O non bloccante unita alla comparsa dei selettori da la possibilità di gestire con un singolo thread un grande numero di canali riducendo la complessità del codice e permettendo un incremento delle prestazioni dovute, in parte, alla mancanza di codice dedicato alla gestione di molti thread.

Capitolo 4

Il Refactoring

Il Refactoring è una pratica che prevede la modifica di un sistema software per migliorarne la sua struttura interna senza alterarne le funzionalità.

Di fatto è un modo disciplinato di "ripulire" il codice esistente (clean-up code) minimizzando le possibilità di introdurre nuovi bug rendendo così il codice piu' facile da comprendere e conseguentemente da gestire (manutenzione, bug fixing o aggiunta di nuove funzionalità).

Il Refactoring è quindi un cambiamento della struttura interna di un software dopo che è già stato realizzato ma il cui codice deve essere migliorato per risultare più comprensibile e gestibile. Un buon momento in cui effettuare Refactoring è al termine di ogni funzionalità implementata e prima di passare alla successiva.

Prerequisito all'attività di refactoring è la presenza di un set completo di *unit test*[1]. Questo perché la loro esecuzione deve verificare in pochi istanti come il comportamento del modulo rimanga corretto. Tramite questi test, il refactoring assomiglia più a un ciclo nel quale ogni passo è formato da piccole modifiche al programma e esecuzione dei test per controllarne la correttezza. Qualora un test fallisse il programmatore sa esattamente le ultime modifiche apportate che hanno introdotto l'errore.

Come si può capire, il processo di refactoring, impone un netto cambiamento rispetto la classica filosofia "finché funziona lasciamolo stare" e prevede di cercare di ottenere il massimo da un programma già di per se funzionante.

[1]test rigorosi che si applicano a pezzi di codice relativamente brevi (solitamente sono scritti per testare singoli metodi).

In realtà questo processo è sempre stato eseguito in qualche forma anche se, per completezza, il primo uso del termine "refactoring" nella letteratura risale ad un articolo del 1990 di William F. Opdyke e Ralph E. Johnson.

4.1 Motivazioni

Un sistema software può degenerare per svariate ragioni e questo può portare ad avere un programma difficilmente mantenibile e soprattutto difficile da estendere se non a fronte di interventi "costosi" in termini di risorse e/o di tempo. Possiamo identificare due categorie di benefici che questa attività comporta:

Mantenibilità: Rappresenta la semplicità di sistemare bug data dalla leggibilità del codice. Questo è ottenuto semplificando i metodi lunghissimi in un set di metodi più semplici e coincisi.

Estendibilità: È la semplicità nell'estendere le funzionalità del codice derivata dall'usare modelli di sviluppo riconoscibili.

Il Refactoring, inoltre, è una pratica che permette di fronteggiare il decadimento del software (software decay) rispetto ai requisiti attuali e futuri cercando di tenere il codice il più lineare possibile. Esso, infatti, permette di effettuare modifiche al codice di tipo evolutive e/o incrementali e/o migliorative a parità di funzionalità dello stesso.

4.2 Passi Principali

Ci sono molteplici operazioni che rientrano nel refactoring. Dato che non rientra negli scopi di questa trattazione darne la lista completa e esaustiva, per un tale approfondimento rimando al sito `http://www.refactoring.com/catalog/index.html` dove è presente la lista indicizzata di tutte le operazioni con relativa spiegazione.

Principalmente, con il refactoring, si applicano tre tipologie di interventi mirate ad:

- Aumentare l'astrazione, ossia nascondere i dettagli implementativi;
- Spezzare il codice in più blocchi logici;
- Migliorare la leggibilità del codice.

Con la prima tipologia di interventi, solitamente, si procede a forzare il codice a impostare/leggere i valori dalle variabili tramite metodi (Encapsulate Field), utilizzare dei tipi di dato generici (Generalize Type) per permettere una condivisione più ampia del codice e si tende a rimpiazzare il controllo dei tipi di dato con il controllo dello stato dell'oggetto (Replace type-checking code with State/Strategy).

Lo spezzettamento del codice, in modo più semplice, prevede di spezzare i metodi lunghi in più parti (Extract Method) per aumentarne la comprensibilità e, come passo ulteriore, di spostare eventuali porzioni di codice in nuove classi (Extract Class).

L'ultima classe di tecniche, il cui unico scopo è favorire la comprensione del codice e del compito che ogni blocco svolge, si basa sullo spostamento di metodi o variabili su classi più inerenti al loro significato oppure, semplicemente, consiste nel rinominare gli stessi per rendere meglio l'idea dello scopo che hanno.

Capitolo 5

Il plugin Torrent

Il lavoro svolto per sfruttare le nuove potenzialità fornite dalla libreria `nio` ha mantenuto lo stesso principio di funzionamento nonostante i profondi cambiamenti che verranno illustrati nel capitolo successivo. Per questo motivo procediamo con una rapida ma esaustiva analisi del funzionamento originario del plugin Torrent.

5.1 La struttura precedente

La struttura del plugin può essere suddivisa in due parti: la pirma si occupa di interfacciarsi al core e, di conseguenza, di richiedere le risorse tramite PariPari; la seconda si occupa di implementare il protocollo torrent.

Per quanto riguarda l'interfacciamento con PariPari, il plugin sfrutta essenzialmente le classi TorrentCore, TorrentListener. La seconda classe va a implementare il Listener per i messaggi in arrivo dal core mentre, per mandare le richieste, viene usata la classe PluginSender che è, di fatto, lo standard del progetto per fare le richieste.

Tuttavia non andiamo ad approfondire questi dettagli dato che il lavoro di modifica del codice ha riguardato essenzialmente la struttura che implementa il protocollo torrent.

Il Core di PariPari, quando l'utente richiede il caricamento del plugin Torrent, lancia il metodo `init()` della classe TorrentCore la quale estende Plugin. Questo metodo si occupa di inizializzare i thread che servono per comunicare con il Core e, di conseguenza, con gli altri plugin di PariPari e, in aggiunta, istanzia il thread TorrentConsole che si occupa di gestire i comandi che arrivano dall'utente.

Ad oggi, tali comandi vengono ricevuti dalla console del core e la lista dei comandi disponibili, oltre ad essere disponibile nella console, è presente in [3] con una breve descrizione di ognuno.

Nel momento in cui l'utente aggiunge un download, il core del plugin crea un oggetto TorrentFile che rappresenta i metadati del file .torrent, associa a questo un TorrentID e istanzia un oggetto DownloadManager che gestisce le operazioni per scaricare i file del torrent. Le informazioni su TorrentID e DownloadManager associato sono mantenute in una hashtable all'interno del core.

La classe DownloadManager è una classe fondamentale per il protocollo torrent. Questo thread, infatti, cura le comunicazioni con il tracker, accetta le connessioni da altri peer e avvia un task per ogni peer con cui si instaura una connessione. Quando viene istanziata da TorrentCore gli vengono passati il TorrentFile, il suo id, il plugin sender globale e il logger e crea a sua volta le mappe Peers e Tasks e la lista Unchoke List, che analizzeremo in seguito.

Sempre questa istanzia tre thread: un file manager, thread responsabile della scrittura su disco sincronizzata, un oggetto RemotePeerListener, che resta in ascolto sulle porte selezionate nella configurazione per la connessione iniziata da altri peer, e un PeerListUpdater, che invece mantiene la connessione con il tracker, ricevendo e aggiungendo alla peer list le informazioni ottenute attraverso questa comunicazione.

Oltre ad istanziare tutto il necessario per il download, questa classe si occupa anche di altre di due operazioni fondamentali: l'unchoking e l'avvio di un DownloadTask per ogni peer con cui si ha uno scambio di dati attivo, mantenuto poi nella mappa Tasks.

Il thread DownloadTask è essenzialmente colui che si occupa di comunicare con ogni peer a cui ci si connette. Può essere istanziato in due occasioni: nel caso si decida di iniziare una connessione con un peer conosciuto attraverso il tracker o il peer exchange, oppure se riceviamo una connessione da un client esterno che ha ottenuto il nostro IP e la nostra porta con i medesimi metodi.

Quando viene istanziato ottiene una serie di oggetti necessari per accedere agli altri plugin di PariPari, le informazioni di hash del torrent a cui si riferisce, un valore che indica se abbiamo cominciato la connessione o meno e il file manager per la scrittura su disco.

All'avvio della connessione, se non è già stato creato in precedenza, crea il socket attraverso Connectivity e istanzia due thread: MessageSender e Message-

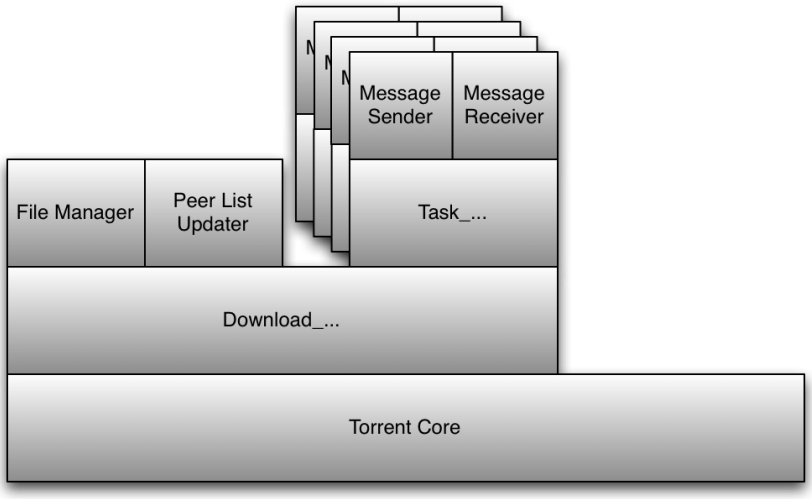

Figura 5.1: Schema delle classi del plugin Torrent

Receiver, che si occupano dell'invio e della ricezione dei messaggi tra peer.

Presentiamo ora i principali package che vengono usati:

- peer: contiene la struttura dati Peer, che immagazzina ogni informazione sul singolo peer: indirizzo, porta, bitfield, fileID, peerID e altre;
- peer.messages: oltre a MessageSender e MessageReceiver descritti precedentemente, contiene tutto il supporto alla messaggistica tra peer: Message_PP e Message_HS rappresentano in forma utile ogni tipo di messaggio, rispettivamente di peer protocol e di handshake e il loro metodo `generate()` crea il pacchetto pronto per la spedizione. `PeerProtocol` contiene le costanti necessarie ad interpretare i messaggi ricevuti e fornisce un valido aiuto alla leggibilità degli eventi gestiti da DownloadTask;
- piece: struttura dati che contiene ogni informazione sul piece: indice, stato di completezza e i byte stessi se non è ancora stato salvato su disco e controllato. Fornisce metodi per scrivere block al suo interno e accedere ai dati;
- tracker: classi per la connessione al tracker, PeerListUpdater per gestire richieste e risposte al tracker e HTTPConnection fa passare ogni tipo di richiesta HTTP attraverso il modulo Connectivity;
- util e util.bencode: forniscono vari metodi di supporto. Bits è una struttura dati comoda per la gestione degli array di bit e delle operazioni su di

essi. Utils fornisce tutta una serie di metodi utili come la conversione dalle strutture dati più comuni alle stringhe di byte utilizzate nelle comunicazioni torrent, hash e altro. Bencode invece fornisce tutti i metodi per tradurre in formato utilizzabile le informazioni in Bencode, rappresentate poi tramite la struttura dati BEValue.

5.2 I problemi

Questo schema di funzionamento presentava problemi in termini di prestazioni e di funzionalità. In particolare, nella sua esecuzione, venivano creati tre thread per ogni download attivo e, in aggiunta, tre thread per ogni peer con cui si apriva una connessione. Questo fatto di per sé portava alla creazione di un numero spropositato di thread che causava un grosso costo di gestione.

Altra questione che andava a peggiorare questo aspetto era dato dai socket bloccanti: molto spesso, infatti, i thread che rimanevano in attesa per mandare dati sugli Stream dei Socket rimanevano attivi anche dopo la chiusura del Socket stesso e non erano in alcun modo forzati a chiudersi.

Il risultato di queste due problematiche era spesso un alto uso della CPU e della memoria che, in alcuni casi, andava a compromettere perfino la normale esecuzione del Sistema Operativo.

Capitolo 6

Le modifiche apportate

Il lavoro è iniziato come un upgrade alla libreria `java.nio`. Durante l'analisi dell'intervento si è evidenziata la necessità di rivisionare anche lo schema delle classi. Tuttavia la serie di modifiche che sono state apportate al codice non sono andate ad intaccare il modo in cui si elaborano i dati.

Passando a `nio` attualmente non usiamo il plugin Connectivity per richiedere gli oggetti socket perchè quest'ultimo non sfrutta ancora questa libreria. Nonostante questo, però, si è cercato di scrivere codice che agevoli la successiva adozione di Connectivity.

6.1 Lo sviluppo

Grazie all'introduzione di `java.nio` è stato possibile progettare due thread globali per l'invio e la ricezione dei messaggi. Precedentemente, invece, per ogni peer con cui si instaurava una connessione si andava a creare un thread MessageSender e un thread MessageReceiver.

La pianificazione di questo intervento ha evidenziato come non sia più indispensabile avere un thread DownloadTask per ogni peer con cui si è connessi. Altro motivo per cui questa classe è stata smantellata nel vero senso del termine, è dovuto al refactoring del codice. Essa, infatti, manteneva variabili inerenti al peer e metodi inglobabili in altre classi. Nella fattispecie l'oggetto Socket rappresentante la connessione è stato spostato nella clase Peer e, con questo, anche tutti i campi che mantengono le informazioni sullo stato attuale della connessione, sul protocollo e sulla crittografia in uso.

Il lavoro di refactoring su questa classe, inoltre, ha portato ad una "fusione" di DownloadTask e della classe DownloadManager. Per semplificare le operazioni di

refactoring e data la necesità di modificare molto codice delle classi, si è preferito sviluppare una nuova classe.

Tale modifica è stata la prima effettuata ed ha modificato il "cuore del plugin". Queste classi, infatti, gestivano in tutto e per tutto le connessioni con peer e l'elaborazioni dei messaggi. La nuova classe che si occupa di queste importanti mansioni è stata nominata DownloadTorrent.

Nella revisione dei metodi fatta durante la migrazione in DownloadTorrent, è stato possibile risolvere problemi quali la mancanza di invio di piece agli altri peer e la mancanza della funzionalità di *seeding*[1].

Tuttavia, l'attuazione di queste prime modifiche ha avuto come conseguenza la revisione e sistemazione di un significativo numero di altre classi del plugin. Questo perché, come effetto di queste modifiche, è stato cambiato il modo in cui vengono inviati e ricevuti i messaggi.

Portando le classi che ricevono e mandano i messaggi a livello globale per il plugin, infatti, si è introdotto il problema di gestire il passaggio degli stessi fra questi thread e il thread DownloadTorrent. Dopo l'analisi di alcune soluzioni possibili, si è scelto di inserire delle liste nella classe Peer che mantengano i messaggi da inviare e i messaggi ricevuti da processare.

Altro problema presentatosi nello sviluppo della classe TorrentMessageSender, è la gestione del messaggio Keep-Alive. Questo, infatti, è indispensabile per mantenere attiva la connessione al peer nel momento in cui non c'è alcuna comunicazione. La struttura precedente prevedeva che il thread MessageSender fosse messo in stato di wait finché non terminasse il timeout per l'invio del Keep-Alive oppure non arrivasse un messaggio da mandare.

Un approccio simile non risultava più ammissibile con lo sviluppo di un thread unico per l'invio dei messaggi. Come soluzione si è scelto di mantenere una coda FIFO *(first in first out)* che mantiene i peer "registrati" e all'interno della classe Peer è stato inserito un campo contenente il timestamp dell'ultimo messaggio inviato. La lista viene gestita in modo da avere in testa il peer che necessiterà per primo di mandare il Keep-Alive. Il thread sviluppato rimane in attesa per l'arrivo di messaggi da inviare oppure per la scadenza dell'invio di Keep-Alive.

Altra questione è stata lo sviluppo della classe TorrentMessageReceiver di cui un estratto è riportato in Appendice A. Questa classe, infatti, sfrutta un Selector per gestire gli eventi sui canali. Perchè ciò avvenga è indispensabile registrare

[1] comportamento con il quale il peer va a condividere il torrent che ha già finito di scaricare.

un SelectableChannel nel selettore. Questa mansione basilare è implementata nei metodi `registerPeer` e `registerConnectedPeer`. Il primo serve per registrare un socket in fase di apertura mentre il secondo è rivolto a quelli già connessi.

Nel primo scenario alla rilevazione dell'evento *O*P_CONNECT[2], ossia quando è presumibilmente terminata la connessione del socket, viene invocato il metodo `finishConnect`. Questo metodo verifica che la connessione al socket sia stata aperta con successo provvedendo in tal caso a cambiare gli eventi per cui il canale è registrato nel selettore in *OP_READ*. In caso la connessione non fosse stabilita, lo stesso metodo provvede a informare DownloadTorrent del fatto che non si è connessi a quel peer.

Qualora, invece, il selettore rilevi la presenza di dati da leggere nel socket (*tramite OP_READ*), viene invocato il metodo `read` che, una volta letti tutti i dati in arrivo nel `SocketChannel`, li inserisce nel ByteBuffer e invoca il metodo analizeBuffer. Il buffer in cui vengono riversati questi dati, essendo relativo alla ricezione di dati dal singolo peer, è mantenuto nella classe Peer e usato nei thread in blocchi sincronizzati (*come è stato detto precedentemente gli oggetti Buffer non sono "thread-safe"*).

Il metodo analizeBuffer, basandosi sulle informazioni del protocollo in uso e sullo stato di collegamento del peer, ha due opzioni:

- Segnalare a DownloadTorrent l'arrivo di dati per cercare di sincronizzare i due peer (operazione effettuata nell'handshake della crittografia);
- Cercare di creare uno o più oggetti Message dal buffer tramite i metodi `readHS`, `readAZ` e `readPP`.

In queste fasi di lettura il buffer viene letto in blocchi di byte e, se non vi sono dati a sufficienza per costruire il messaggio, viene "ripristinato" tramite il metodo `rewind()`. I tre metodi `readHS`, `readAZ` e `readPP` cercano di leggere un messaggio. Il primo si occupa di leggere messaggi di Handshake oppure, data la possibilità di avere crittografia prima dello stesso, di leggere una chiave. Gli altri due si occupano di leggere messaggi del protocollo Azureus (`readAZ`) e del protocollo Torrent con eventuali estensioni di questo. Si noti che non sono state fatte ulteriori suddivisioni del metodo `readPP` dato che la struttura del messaggio

[2]L'oggetto SelectionKey ha definite le costanti `OP_READ`,`OP_WRITE`,`OP_CONNECT` e `OP_ACCEPT` per identificare i tipi di evento sul canale. Esse rappresentano rispettivamente la possibilità di leggere dati, di scriverli, di concludere il processo di connessione e di accettare una richiesta di connessione.

è uguale sia nel caso venga adottato il *plain torrent protocol* sia nel caso vengano utilizzate estensioni dello stesso. Tale scelta ha permesso di evitare del codice ridondante.

La strategia di lettura, che è stata scelta per la sua semplicità implementativa, ha portato con sé un grosso problema sulle connessioni con crittografia abilitata. Questo perchè il sistema di crittografia usato ha chiavi che variano ad ogni byte letto e sono sincronizzate fra mittente e destinatario. Di conseguenza, la lettura di una porzione di un messaggio induce una modifica nelle chiavi per la decodifica. Questo, unito al fatto che il messaggio viene rimesso nel buffer qualora non sia completo, comportava una desincronizzazione delle chiavi crittografiche usate nella comunicazione.

A tale proposito sono stati sviluppati tre metodi nelle classi EncryptionManager e RC4Engine atti a fare un "freeze" dello stato delle chiavi di codifica/decodifica e ripristinare o scartare il salvataggio.

Torniamo ora ad analizzare nel dettaglio come è stata creata DownloadTorrent. Questa classe, come accennato in precedenza, riprende i metodi di DownloadManager modificati per rimuovere ridondanze sulle variabili e semplificarne la leggibilità. Oltre a ciò ha inglobato alcuni metodi che prima erano in DownloadTask mentre altri metodi di questa classe sono stati semplicemente abbandonati dato che rappresentavano, per la maggiore, richiami a metodi di DownloadManager. Va sottolineato come i metodi in questione hanno necessitato di una parametrizzazione per specificare l'oggeto peer rispetto al quale si sta lavorando. Il fatto che prima fossero nella classe DownloadTask, infatti, implicava la presenza dell'oggetto peer come variabile globale mentre ora sono stati inseriti in una classe che si occupa di molteplici peer.

Una sostanziale differenza si è evidenziata nella scrittura del metodo che inizializza la connessione. Questo perchè ora l'apertura del socket viene fatta in modo non bloccante e, di conseguenza, bisogna tener conto che l'apertura del socket non corrisponde all'effettiva connessione inzializzata. Per questo motivo il metodo `initConnection`, a differenza di come funzionava precedentemente, ora ha il solo compito di inizializzare il socket e registrarlo nel nuovo TorrentMessageReceiver. È stato introdotto il metodo pubblico `endConnection` che si occupa di gestire la procedura di handshake quando il socket è connesso con successo. Tale metodo, infatti, viene chiamato dal ricevitore di messaggi qualora venga stabilita

la connessione sul socket.

Altro compito importante è stato la riscrittura totale della classe in ascolto per connessioni iniziate dagli altri peer. Durante la revisione della classe ConnectionFromPeer, infatti, è stato rilevato dalla semplice lettura del codice come il comportamento fosse scorretto rispetto alla mansione da svolgere. Il compito di tale classe è di restare in ascolto su una porta pubblica impostata per ricevere richieste di connessione da altri peer. Il vecchio funzionamento, al contrario, istanziava un thread per ogni download e, ovviamente, tutti questi thread non potevano avere un proprio socket in ascolto sulla stessa porta.

Per questo motivo, sempre introducendo `java.nio`, la nuova versione della classe è stata sviluppata in modo da avere un thread globale per tutto il plugin che rimane in ascolto per connessioni sulla porta indicata nelle configurazioni del plugin. Questo ha comportato anche l'introduzione nella stessa del codice per la gestione dell'handshake con il peer. Ciò perchè è necessario conoscere a quale download il peer che instaura la connessione si riferisce in modo da poterlo registrare nel corrispondente ConnectionFromPeerListener.

Similmente a quanto detto per TorrentMessageReceiver, questa classe sfrutta un selettore che rimane in ascolto per eventi di accettazione di nuove connessioni (evento prodotto dal ServerSocketChannel registrato in esso), per la lettura di dati nonché per la finalizzazione della connessione da parte dei socket accettati.

6.2 Il nuovo schema

Lo schema dei thread istanziati dal nostro plugin ottenuto con questo intervento evidenzia molte differenze rispetto a quanto presentato nel capitolo precedente.

Il fatto più significativo è proprio la scomparsa dei gruppetti di thread che DownloadManager creava per gestire le connessioni dei singoli peer, ossia DownloadTask, MessageSender e MessageReceiver. Il compito del primo è stato integrato in DownloadTorrent mentre a fare le veci degli altri thread ora sono presenti il TorrentMessageSender e TorrentMessageReceiver a livello globale.

Per presentare una stima in cifre si può pensare di connettersi ad una decina di peer. In questa situazione la struttura vecchia farebbe partire il Manager che, a sua volta, avvierebbe i thread per la scrittura su disco, per la connessione al tracker, per l'ascolto di connessioni da remoto e un DownloadTask per peer da

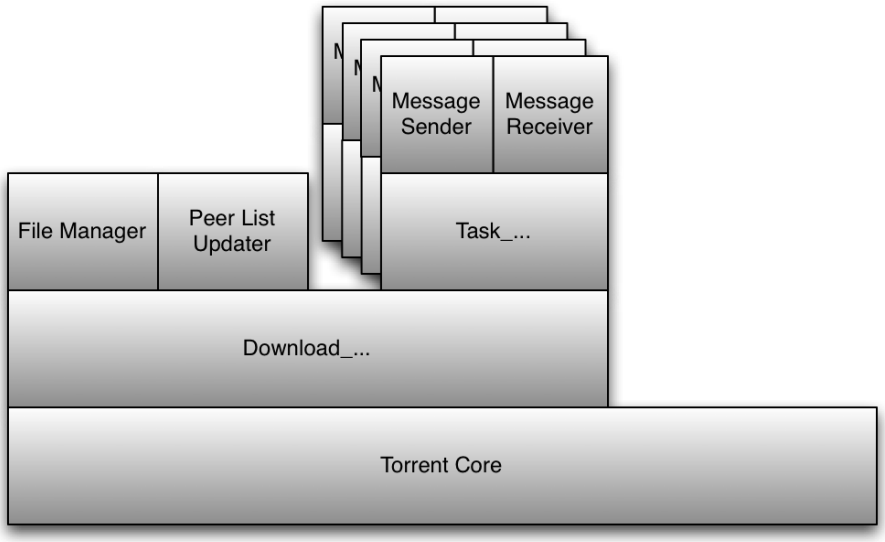

Figura 6.1: Schema delle classi del vecchio plugin Torrent

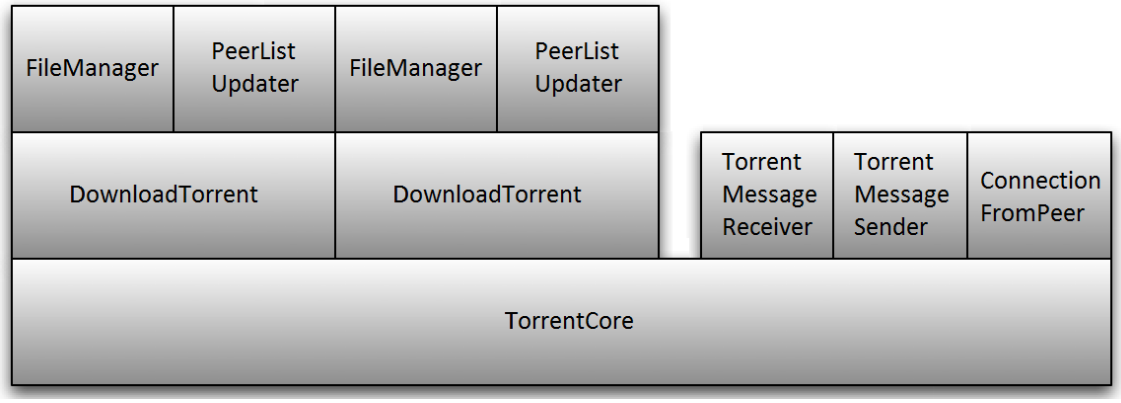

Figura 6.2: Nuovo schema delle classi del plugin Torrent

contattare. A sua volta il task creerebbe i gestori dei messaggi e la conta dei thread arriverebbe a ben 34.

Al contrario la nuova struttura si limiterebbe a creare i thread a livello globale per i messaggi e per la connessione da remoto mentre per il download inizializzerebbe solamente il DownloadTorrent, il FileManager e il PeerListUpdater per un totale di solo 6 thread.

Ma la differenza si apprezza maggiormente nel caso in cui si facciano partire altri download. La vecchia struttura, infatti, istanzierebbe altri 34 thread mentre con i nuovi sviluppi si avrebbe un aumento di solamente 3 thread. Questo evidenzia che la crescita di thread è ancor più contenuta!

Se pensiamo al scenario di 6 download attivi con 10 peer connessi per ciascuno, il vecchio plugin arriva a ben 204 thread mentre il nuovo si limiterebbe a 21: si ha un risparmio di quasi il 90% di thread.

6.3 I vantaggi

L'introduzione di `java.nio`, come già detto più volte, porta un sensibile miglioramento nelle prestazioni di input/output sui socket che, nel caso del protocollo Torrent, sono il punto critico dato il grande traffico di dati fra peer.

Sempre l'introduzione di questa libreria ha permesso di sfruttare una nuova struttura dati per la lettura da socket: i Buffer; questa struttura dati ha permesso la lettura a blocchi di byte e, conseguentemente, ha consentito alla rimozione dei cicli di lettura che andavano ad appesantire il codice e a comprometterne la leggibilità.

La sostanziale riduzione del numero di thread che vengono istanziati ed avviati dal plugin presentato nel paragrafo precedente, inoltre, porta un beneficio sulle risorse utilizzate dal nostro plugin. Questo perché riducendo i thread si ha una riduzione sull'overhead dovuto allo scambio di contesto e, ancora più importante, la limitazione della memoria necessaria per istanziare i vari oggetti.

Capitolo 7

Le performance

Analizziamo ora dei paragoni fra le prestazioni prima e dopo il refactoring. Tali test sono stati effettuati sullo stesso computer in due fasi distinte. Si è scartata la possibilità di eseguire i test in parallelo su due macchine simili data la possibilità di arrivare ad avere una mutua connessione, ossia che i due computer scarichino l'uno dall'altro.

7.1 Gli strumenti

Per controllare le risorse è stato sfruttato *JConsole*. Questo tool, il cui nome esteso è Java Monitoring and Management Console, fornisce un sistema per monitorare e gestire la piattaforma Java con le applicazioni che sono in essa eseguite.

Per andare a raccogliere dalla Java Virtual Machine le informazioni prestazionali e sui consumi di risorse delle applicazioni sfrutta la tecnologia *Java Management Extension* (JMX) .

Ovviamente, per usare questo tool, bisogna fare in modo che l'applicazione da monitorare venga fatta partire con un agente che gestisca il monitoraggio. Per specificare l'avvio di questo bisogna inserire l'argomento `-Dcom.sun.management.jmxremote` nel momento in cui si fa partire il Core di PariPari. Successivamente tramite JConsole verrà listata fra le applicazioni monitorabili il `Core.jar` e sarà possibile iniziare a monitorare.

Ovviamente va sottolineato che le statistiche raccolte sono riferite all'intera esecuzione di PariPari. Tuttavia ciò non disturba in modo eccessivo l'analisi che faremo dato che i moduli caricati da PariPari sono gli stessi.

7.2 Old VS New

Come primi test sono stati messi a confronto il vecchio plugin e il nuovo nel download di un file .torrent di dimensioni contenute (ordine di 10MB), e si sono già potute notare sostanziali differenze.

Iniziamo il confronto di performance con il "piatto forte", ossia la differenza in termini di thread. Come abbiamo analizzato nei capitoli precedenti, la riduzione di thread dovrebbe mostrare risultati evidenti. In figura 7.1 è riportato l'andamento dei thread che si sono creati con il vecchio plugin.

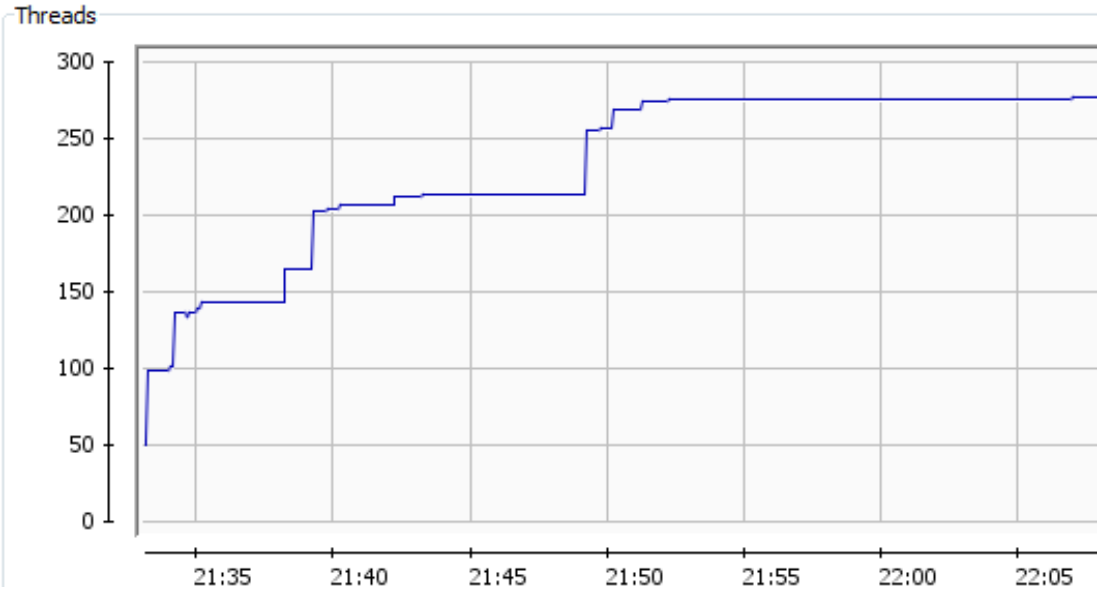

Figura 7.1: Andamento thread PariPari con plugin Torrent prima dell'intervento

Da notare come direttamente all'avvio del download, il numero cresca significativamente e continui a crescere con un andamento a gradini (tali gradini si riferiscono al momento in cui si contatta il tracker e si iniziano nuove connessioni) fino al picco di ben 281 thread. In opposizione con la nuova versione si rileva dalla figura 7.2 che si ha un piccolo aumento all'avvio che poi si stabilizza nell'ordine di 60 thread.

Come già detto, assumendo che i thread che non sono del plugin torrent siano uguali in entrambi i casi (cosa non vera ma presumibilmente saranno in misura simile), si ha una differenza rilevata su un singolo download di più di 200 thread rispetto alle stime teoriche dei capitoli precedenti che erano dell'ordine di 30 thread. Questo avvenimento può essere motivato dalla presenza di thread bloccati

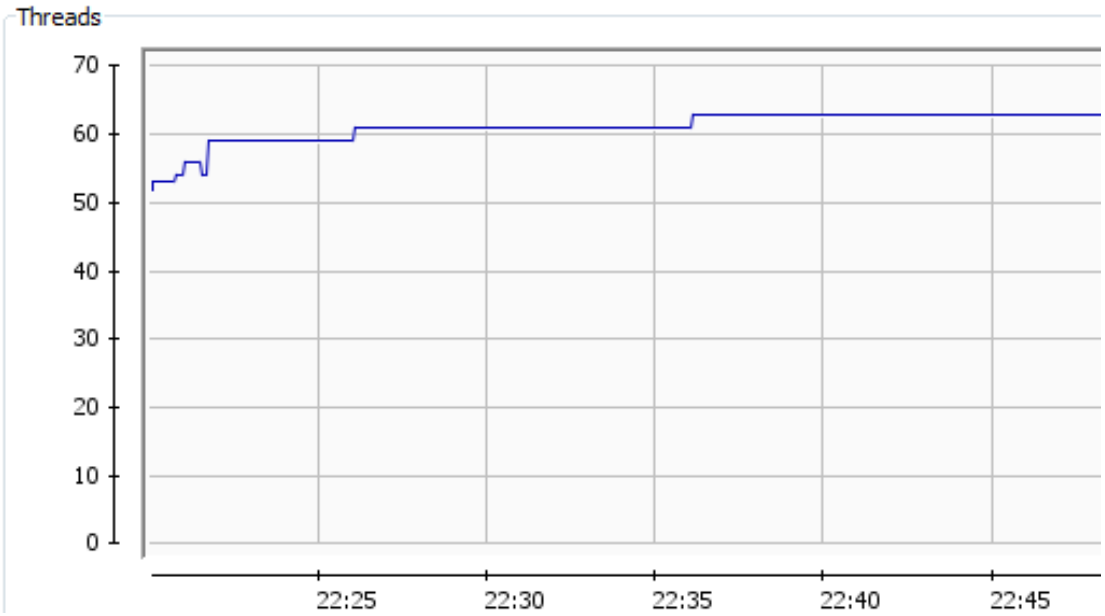

Figura 7.2: Andamento thread PariPari con plugin Torrent attuale

su socket che rimangono attivi in stato di wait.

Proseguendo l'analisi delle prestazioni, passiamo all'andamento del consumo di memoria illustrato nelle figure 7.3 e 7.4. Anche qui si nota una sostanziale differenza: si passa dagli oltre 40MB di picco ai più modesti 15MB per la nuova versione. Chiaramente una parte di questo risparmio è dovuto alla riduzione dei thread anche se, confrontando i grafici di thread e memoria con gli stessi riferimenti temporali, si nota come l'incremento a gradino dei thread non corrisponda a un comportamento simile della memoria.

Come ultimo parametro analizzato in JConsole, vediamo l'andamento dell'utilizzo della CPU nelle immagini 7.5 e 7.6. In questo caso, se non altro, non è visibile una netta differenza anche se sono spariti molti picchi di utilizzo e, nel complesso, la percentuale di utilizzo è leggermente inferiore.

Come parametro ricavabile dai grafici confrontiamo il tempo impiegato per il download: la configurazione precedente all'intervento presentato ha impiegato ben 35 minuti per effettuare il download mentre la nuova configurazione "solo" 30 e questo equivale ad un risparmio del 14%.

È doveroso precisare che i tempi indicati per il download non sono dovuti a inefficienze o problematiche del nostro plugin, bensì al fatto che i test sono stati effettuati su una connessione alla rete mobile e, quindi, con banda limitata.

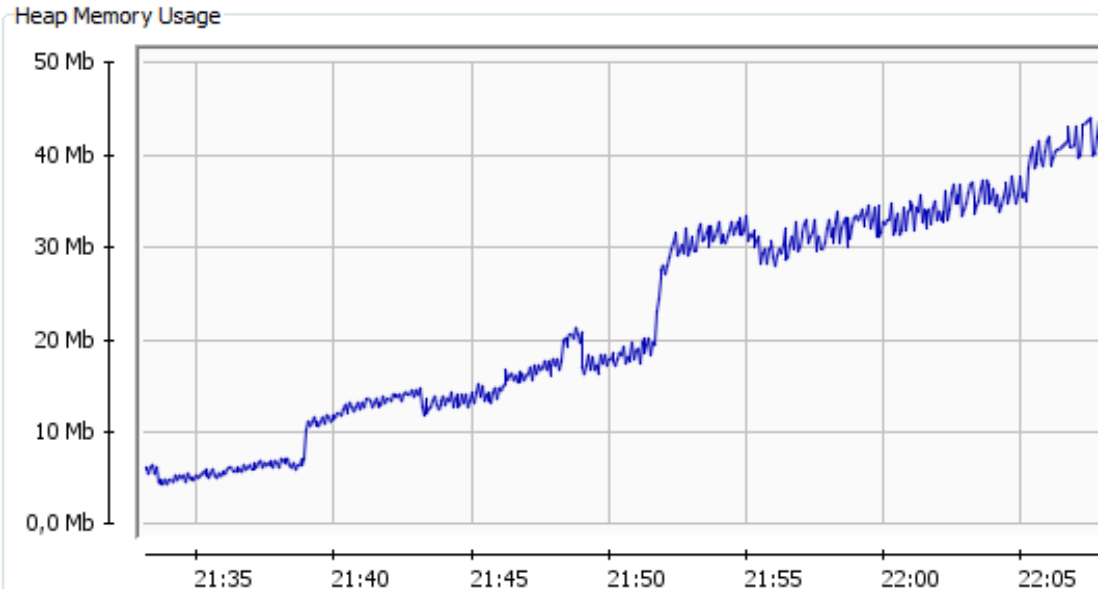

Figura 7.3: Utilizzo memoria PariPari con il vecchio plugin Torrent

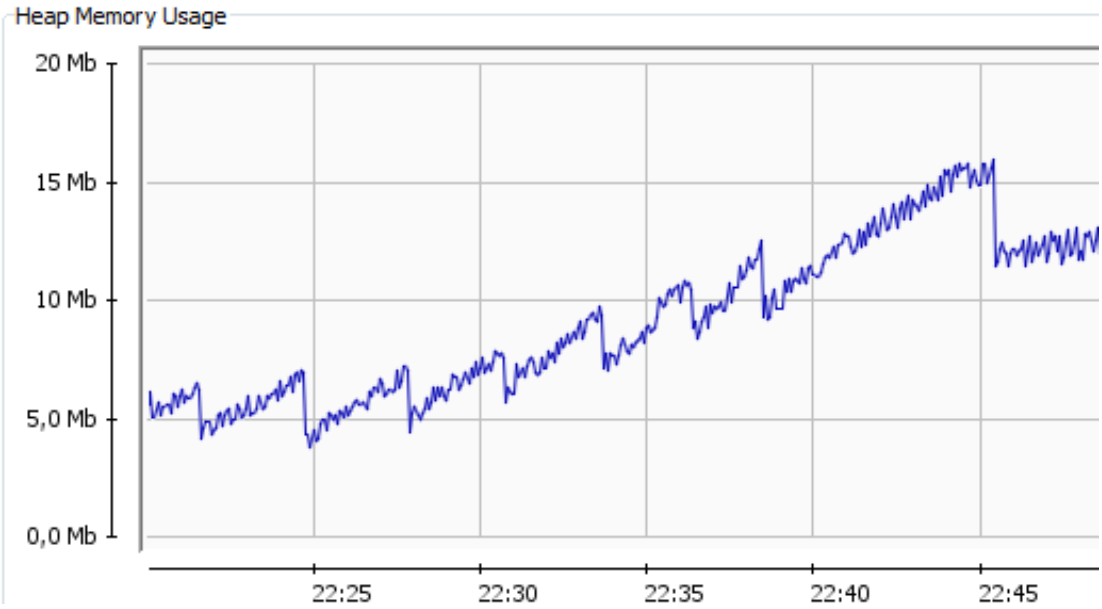

Figura 7.4: Utilizzo memoria PariPari con plugin Torrent dopo la sistemazione

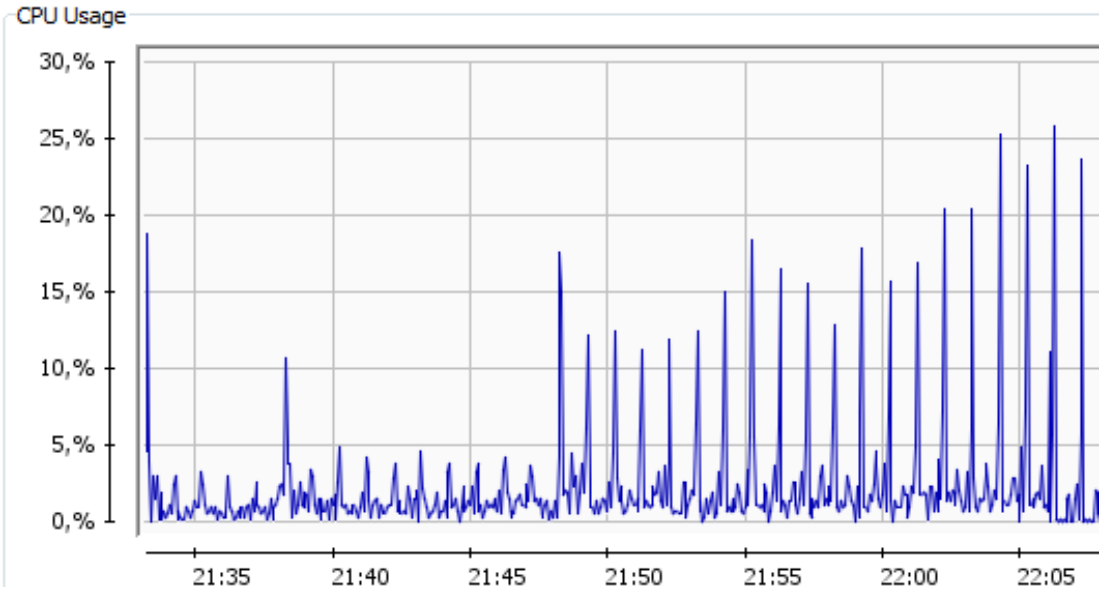

Figura 7.5: Sfruttamento CPU PariPari con plugin Torrent originario

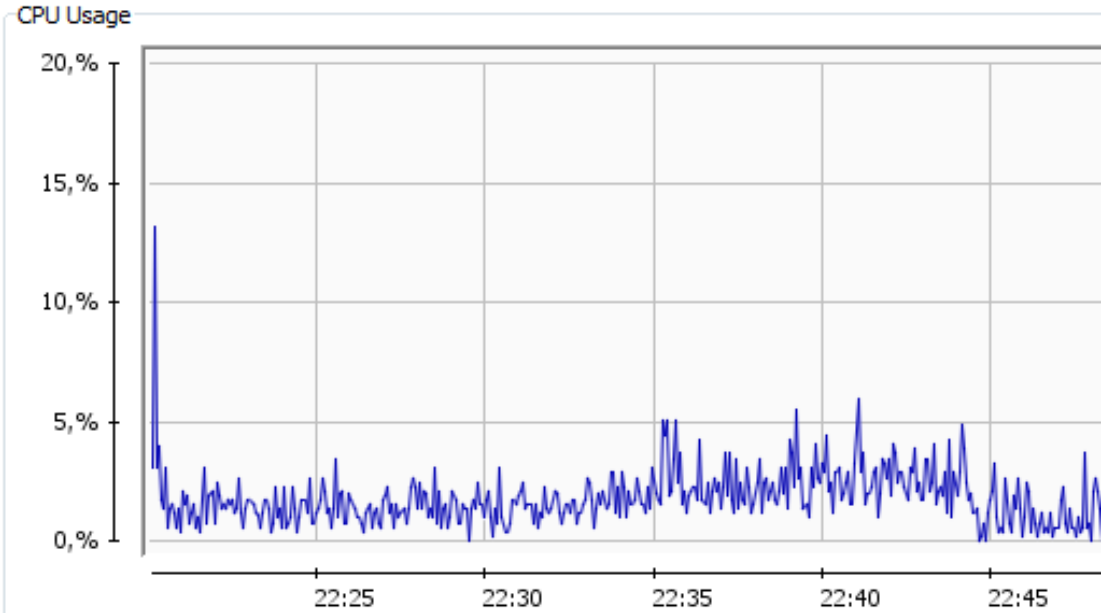

Figura 7.6: Sfruttamento CPU PariPari con nuovo plugin Torrent

7.3 Confronto con altri Client

Per avere una stima delle prestazioni rispetto ad altri client torrent sono state fatte alcune prove fra il nostro plugin e il programma *uTorrent*. I primi test hanno riguardato il download di file di dimensioni contenute, ossia intorno ai 5MB. I risultati ottenuti sono molto simili, ossia il tempo impiegato con una configurazione equivalente in entrambi i casi rimane sull'ordine del minuto.

I successivi test hanno avuto come termine di paragone il download di un file di dimensioni dell'ordine di 500MB. Per scaricare questo file, il nostro plugin ha impiegato ben 27 minuti mentre *uTorrent* ha richiesto 20 minuti. Questa differenza è circa il 25% del tempo globale per il download.

Un tal risultato ci fa capire come il nostro plugin possa ancora migliorare. A tal proposito alla pagina 49 sono riportate alcune possibili migliorie future.

Sviluppi Futuri

Il margine di miglioramento per il nostro plugin è ancora elevato.

In primo luogo si potrebbe ridurre ulteriormente il numero di thread istanziati utilizzando il *thread pool pattern*, ossia un numero di thread che varia in base all'effettivo carico di lavoro da gestire in modo da controllare il rapporto fra prestazioni fornite e risorse richieste.

Altra miglioria possibile risiede nel modo in cui si gestisce il transito delle informazioni dal buffer al disco. Il non passare da oggetti bensì scrivere direttamente dal buffer ai file, porterebbe un considerevole risparmio di cpu e di costo computazionale. Tuttavia c'è da considerare come un simile lavoro comporti la modifica di molto codice dato che implica lo stravolgimento del sistema di scrittura dei pezzi su disco, di lettura e gestione dei messaggi.

Conclusioni

L'intervento di refactoring svolto sul codice ha portato molti benefici al nostro plugin. L'introduzione di `java.nio`, come già detto, ha portato ad un sensibile miglioramento nelle prestazioni di input/output sui socket che, nel caso del protocollo torrent, sono il punto critico dato il grande traffico di dati.

Riuscire a limitare i thread necessari per il funzionamento del plugin, come analizzato nel capitolo 7, ha portato a risparmiare molte risorse del sistema e, come effetto indotto, si è migliorata la stabilità del plugin.

L'intervento di sistemazione del codice, inoltre, ha portato un profitto in termini di leggibilità e semplicità di intervento. Questo è molto importante dato che il lavoro di sviluppo non è seguito da una sola persona. Tale beneficio è essenziale per evitare un dispendioso lavoro di analisi del codice da parte di programmatori che necessitano di apporre modifiche.

Lo sviluppo in team a cui ho preso parte svolgendo questo lavoro ha evidenziato le difficoltà di analisi di codice scritto male. Questo problema, tuttavia, è stato limitato dalla presenza di colleghi molto disponibili e preparati che hanno reso il lavoro molto confortevole. La mia esperienza in questo gruppo è stata molto positiva e ho potuto apprendere una metodologia di lavoro molto importante per il futuro inserimento nel mondo del lavoro.

Bibliografia

[1] Paolo Bertasi, *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, 2005

[2] Michele Bonazza, *PariCore*, 2009

[3] Simone Pozzobon, *PariPari: modulo Torrent*, 2008

[4] Alessandro Calzavara, *PariPari: Testing del Modulo Torrent*, 2008

[5] Dario Turchetto, *PariPari: Torrent - LibTorrent, Fast Extension*, 2009

[6] Mattia Meneguzzo, *PariTorrent: Azureus Messaging Protocol*, 2009

[7] Andrea Gallo, *PariPari: Crittografia Torrent*, 2008

[8] Ron hitchens, *Java^{TM}NIO*, O'REILLY©, 2002

[9] Wiki PariPari, `http://www.paripari.it/mediawiki/index.php`

[10] Wikipedia, `http://en.wikipedia.org/wiki/Extreme_Programming`

[11] Wikipedia, `http://en.wikipedia.org/wiki/Test-driven_development`

[12] Wikipedia, `http://en.wikipedia.org/wiki/New_I/O`

[13] Wikipedia, `http://en.wikipedia.org/wiki/BitTorrent`

[14] Vuze Wiki, `http://azureuswiki.com/index.php`

[15] Sito, `http://sourcemaking.com/refactoring`

[16] Wikipedia, `http://en.wikipedia.org/wiki/Code_refactoring`

[17] Sito, `http://www.methodsandtools.com/archive/archive.php?id=4`

[18] Guida Sun, `http://java.sun.com/j2se/1.5.0/docs/guide/management/jconsole.html`

[19] Emiliano Tramontana, *Dispense del Corso di Ingegneria del Software 2*, `http://www.dmi.unict.it/~tramonta/se2/`

Elenco delle figure

Appendice A

Classe TorrentMessageReceiver

```
/**
 * Thread created to listen for incoming message from all peers
 * This thread listen ciclically all socket and, when message is
 * received, message type is determined and a new Message object
 * is created and added in peers queue.
 */
public class TorrentMessageReceiver extends PariPariRunnable
       implements ITorrentMessageReceiver{
   [...]

   public TorrentMessageReceiver(ITorrentLogger log){
      [...]
   }

   @Override
   public void go() throws InterruptedException {
      [...]
      while(running){
         if(this.selector.keys().size()>0){
            try {
               selected = this.selector.select();
               Iterator<SelectionKey> i =
                  this.selector.selectedKeys().iterator();
               synchronized(i){
                  while(i.hasNext()){
                     key = i.next();
```

```
                    if (key.isValid() && key.isConnectable())
                       finishConnection(key);
                    else if (key.isValid() && key.isReadable())
                       read(key);
                    i.remove();
                 }
              }
           } catch (IOException e) {
              [...]
           }
        }
        [...]
     }
  }

  protected void finishConnection(SelectionKey key) throws IOException{
     //retrive SocketChannel and Peer informations
     SocketChannel socketChannel = (SocketChannel)key.channel() ;
     IPeer p = (IPeer)key.attachment();

     //try to finish connection process
     try{
        if(socketChannel.finishConnect()){
           p.getDownloadTorrent().endConnection(p);
           key.interestOps(SelectionKey.OP_READ);
           [...]
        }else{
           p.getDownloadTorrent().disconnect(p);
           key.cancel();
        }
     }catch(ConnectException ce){
        ((TaskListener)p.getDownloadTorrent()).taskCompleted(p,
           IDownloadTorrent.ERROR_TIMEOUT);
        key.cancel();
     }
  }
```

```
protected void read(SelectionKey key) throws IOException{
   //retrive SocketChannel and Peer informations
   SocketChannel socket = (SocketChannel)key.channel() ;
   IPeer p = (Peer)key.attachment();

   //allocate a temporally buffer for read
   ByteBuffer buff = ByteBuffer.allocate(32*1024);
   int readed;

   //Read from socket
   readed = socket.read(buff);
   //prepare for read buffer
   buff.flip();
   //Stream closed
   if (readed < 0){
      throw new IOException("Received -1 so the stream is closed.");
   }
   //I've read something so i need to enter the cicle!
   [...]
   //put read data to peer buffer, create a message
   //and add it to peer queue
   //lock is necessary because Buffers are not thread safe!!
   synchronized(p.getByteBuffer()){
      //append data to peer buffer if i've read something
      if(!(readed == 0))
         p.getByteBuffer().put(buff);
      //clear temporally buffer
      buff.clear();
   }
   this.analizeBuffer(p);
}

public void analizeBuffer(IPeer p){
   //Attempt to create messages from data
   //and append it to queue in peer
   boolean enough = true;
   //If peer is in resynching mode try to do it
```

```
if(p.getResynchType() == IPeer.RESYNCH_VC){
   enough = resynchOnVC(p);
}else if(p.getResynchType() == IPeer.RESYNCH_HASH){
   enough = resynchOnHash1S(p);
}
//Read messages are in peer queue
synchronized (p.getByteBuffer()){
   while (enough){
      //try to read messages
      if( p.hasElaboredHS() || !p.hasReceivedHS() ){
         //prepare for read data
         p.getByteBuffer().flip();
         if(!p.hasReceivedHS()){
            //Read HansShake
            enough = readHS(p);
         }else{
            if(p.getInUseProtocol()==IPeer.PROTOCOL_AZMP){
               //Read AZMP
               try {
                  enough = readAZ(p);
               } catch (Exception e) {
                   [...]
               }
            }else{
               //Read LTEP, fastext, pp
               enough = readPP(p);
            }
         }
         //prepare for write data
         p.getByteBuffer().compact();
      }else{
         enough = false;
      }
   }
}
//Inform TaskListener that p may have received something
((DownloadTorrent)p.getDownloadTorrent()).parseMessage(p);
```

```
}

private boolean readPP(IPeer p) {
   //Read LTEP,Fast and PP message
   [...]
}

private boolean readAZ(IPeer p) throws Exception{
   //Read Azureus message
   [...]
}

private boolean readHS(IPeer p) {
   //Read handshake message or Key
   [...]
}

public boolean resynchOnHash1S(IPeer p) {
   //Try to resynch message bytes
   [...]
}

public boolean resynchOnVC(IPeer p) {
   //Try to resynch message bytes
   [...]
}


public void registerPeer(IPeer p){
   //Register SocketChannel of peer for ending connection
   try {
      p.getSocket().register(selector,
         SelectionKey.OP_CONNECT).attach(p);
   } catch (ClosedChannelException e) {
      [...]
   }
   [...]
```

```
    }

    public void start() {
        running = true;
    }

    public void stop() {
        running = false;
        selector.wakeup();
    }

    public void deregisterPeer(IPeer p){
        //Cancel the registration of peer's SocketChannel in the Selector
        p.getSocket().keyFor(this.selector).cancel();
        this.selector.wakeup();
    }

    public void registerConnectedPeer(IPeer p) {
        //Register SocketChannel of peer for reading operation
        try {
            p.getSocket().register(selector, SelectionKey.OP_READ).attach(p);
        } catch (ClosedChannelException e) {
            [...]
        }
        [...]
    }
}
```

Ringraziamenti

Ringrazio il Prof. Enoch Peserico e l'Ing. Paolo Bertasi che hanno permesso la realizzazione di questo progetto. Oltretutto sono molto grato per avermi dato la possibilità di lavorare in un gruppo di lavoro stupendo quale è PariPari.

Ringrazio tutti i colleghi presenti e passati con cui ho potuto lavorare e avere uno scambio di conoscenze molto proficuo.

Un doveroso grazie è rivolto a Paolo Bertasi, Simone Pozzobon, Dario Turchetto, Alessandro Calzavara, Nicola Moro e Francesco Castellani per il loro aiuto nella correzione di questa tesi.

Ringrazio le sorelle Perini per l'ospitalità e per il loro apporto festoso alla vita d'appartamento.

La mia gratitudine a tutti i membri della mia compagnia: grazie per tutti i momenti spensierati e per avermi sempre sopportato.

A tutti gli amici universitari che in questi sei anni hanno contribuito a rallegrare le pesanti giornate di studio.

Alla mia famiglia un grazie speciale per aver sempre creduto nelle mie scelte e per il supporto, spirituale e materiale, datomi in questi lunghi anni universitari.

Non può mancare la Tina in questa pagina dato tutto il lecchinaggio fatto pur di comparire nei ringraziamenti.

Grazie!

I computer sono incredibilmente veloci, accurati e stupidi.
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.
L'insieme dei due costituisce una forza incalcolabile.

— Albert Einstein