Università Degli Studi di Padova
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

PariTorrent: Refactoring

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

CORRELATORE:
Ing. Michele Bonazza

LAUREANDO:
Simone Pozzobon
Matr. N.607257

Anno Accademico 2011/2012

a Maddalena

Sommario

Questa tesi rappresenta la parte conclusiva del lavoro svolto sul plugin PariTorrent di PariPari. PariPari è una rete P2P multifunzionale ed estensibile, progettata e sviluppata dagli studenti del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova. Per come è stato progettato e reingegnerizzato durante gli anni, PariPari non è in grado solo di offrire molteplici funzionalità all'utente finale: dal file-sharing alla comuncazione sia testuale che via voce, ma anche di fornire servizi alla rete stessa, attraverso la sua DHT e le sue librerie di distribuzione dedicate come ad esempio DiESeL. La sua struttura aperta consente a chiunque voglia avvicinarsi al progetto di poter usufruire della sua struttura di rete e delle API offerte per accedere alle varie risorse per poter scrivere il proprio plugin.

PariTorrent è uno dei suoi moduli attualmente funzionanti ed in continuo sviluppo, e ne consente la connessione alla rete BitTorrent. Lo scopo di questa tesi è quello di descrivere il refactoring che è stato necessario per consentire a questo plugin di poter utilizzare la nuova versione di PariConnectivity, il modulo preposto all'accesso alla rete Internet, basata su Java NIO, che offre maggiore scalabilità e performance globali, ed un minor utilizzo di memoria rispetto alla versione precedente. Il lavoro verrà contestualizzato descrivendo complessivamente il funzionamento di PariPari, di BitTorrent e del plugin, di Java NIO e di PariConnectivity. Verrà poi analizzato il processo di adattamento, con uno sguardo finale alle performance.

Abstract

PariPari is a multifunctional and extensible P2P network, designed and developed by the students of the Department of Information Engineering of the University of Padova.

Through extensive reengineering and thorough design, PariPari is capable of offering both services to the final user, from file-sharing to voice and text chat, and services to the network itself, through its DHT network and its dedicated distribution libraries such as DiESeL. Its open structure lets anybody who wants to join the project have easy access to its overlay network and APIs, granting everything that is necessary to write your own plugin.

PariTorrent is the BitTorrent module of PariPari. It is currently working and in continuous development. The goal of this thesis work is to describe the refactoring that the plugin has undergone in order to use the new, improved, version of PariConnectivity, the inner circle module that manages access to the network. The new PariConnectivity offers better performance and scalability, while reducing memory footprint and keeping ease of use, exploiting the asynchronous capabilities of the Java NIO library.

The work will be put into context by describing the overall functioning of PariPari, the BitTorrent Protocol and the PariTorrent plugin, Java NIO and PariConnectivity. We will then analyse the refactoring process and end with a final overlook at the current status of the plugin.

Contents

Introduction

PariPari is a P2P platform in development at the Department for Information Engineering of the University of Padua. It is currently entirely designed, managed and developed by students alone, more than sixty take part in the project at the moment. It was first introduced by Paolo Bertasi in his Master Thesis [1] and developed through his PhD [2], he was chief manager and architect of the project since 2006. Recently this role has been passed to Michele Bonazza, which has been the main designer and developer of the PariPari Core [3] and is currently following the project as part of his PhD duties.

PariPari pursues a very ambitious goal: to unite most of the most widely used services of the Internet in one decentralised, secure and highly performing platform. It currently encompasses many traditional P2P services like eMule, BitTorrent, VoIP and IM and ventures into more ambitious ones such as a distributed web-server, backup system and DBMS.

The development of PariTorrent, PariPari's BitTorrent plugin, started very early, and has undergone several modifications since its inception by Alessandro Calzavara [4] and me, Simone Pozzobon [5]. Most of the time it was to accommodate new features and protocols, such as support for alternative peer messaging systems (Azureus [14] and LTEP [15]) or encrypted messages to prevent ISP throttling [16], other times to correct improper design choices or to adapt it to new versions of the core, the credits system or such as this case, the connectivity module.

PariTorrent was originally made to work with the old PariConnectivity and its several, and sometimes buggy, versions, which wrapped around standard Java sockets and proved to be quite inefficient. The synchronous nature of Java sockets forced us to use one processing thread for each one of them, which, while getting the job done quite easily, also consumed a lot of system memory as well as CPU power. Therefore, to communicate with each peer and keep track of its download and upload statistics, which is fundamental for the BitTorrent protocol to work, we needed a socket for each one of them and thus a thread too. The number of peers can vary in the 1-50 range for each torrent download, evidently driving the number of threads dangerously high. To temporarily thwart this glaring problem we decided to switch, ahead of its due time, to Java NIO, and to use the asynchronous sockets it offered directly, effectively bypassing the PariPari plugin system and circumventing its security module. While this solved the problem egregiously, it caused us to need to adapt once again when the connectivity module would have been ready. It finally is, and, while differing significantly from the original Java NIO implementation, offers solid performance and a quite developer-friendly interface.

Chapter 1 will provide a presentation of the PariPari project as a whole, focusing on its main features and some of its inner working mechanisms. It will also give an outlook of how the team is managed and what tools and techniques are used during development.

Chapter 2 will focus on the revised PariConnectivity, its inner workings and external APIs. It will also present some of the most interesting aspects of Java NIO, the New I/O library used in both PariConnectivity and the previous version of PariTorrent.

Chapter 3 delves into the BitTorrent Protocol, providing all the necessary insight to understand how the network works, how peers communicate and share data between them. It will also present the extensions to the protocol that are currently supported by PariTorrent.

In Chapter 4 the status of PariTorrent before the refactoring is described in detail. Both in its original structure and what it came to be after years of minor and major modifications, with a particular focus on the main classes and data structures that were altered during the refactoring process.

Chapter 5 describes all the main changes that the plugin has undergone in order to fully exploit the new capabilities of PariConnectivity. It presents clearly how the plugins interact and some key algorithms that were used to achieve perfect compatibility.

CHAPTER 6 draws some conclusions on the work that was done, with an eye to possible future development.

1 PariPari

1.1 Introduction to PariPari

PariPari is a server-less P2P multi-functional application currently under development at the Department of Information Engineering of the University of Padova. It differs from traditional P2P applications like eMule, Skype or µTorrent in that it provides a multifunctional and extensible platform: a large number of heterogeneous services, ranging from the traditionally P2P ones (e.g. file sharing and VoIP) to more centralised ones (like Web and e-mail hosting, IRC chat and DNS) are accessible through a collection of simple and uniform APIs. Its highly standardized nature allows a large body of students (presently, more than sixty) to cooperate in its development. In order to provide access to all these services, to potentially allow everyone (not only students of the University of Padova) to take part in the project and develop his own plugin, and to guarantee security throughout the entire system, PariPari uses a highly modular structure in which every service is managed by a plugin. While the user will be more interested in the so called "outer circle" plugins, that are the ones that offer services to the user himself, such as IM and file-sharing, PariPari also offers a set of tested and secure[1] APIs to developers (the so called "inner circle"), with services ranging from direct network and disk access, to communication through our very own DHT[2]. Every request passes through the Core which is responsible for forwarding it to the correct plugin and also integrates a highly sophisticated credits system, which ensures that the User Experience is always top-notch[3], by effectively offering an integrated QoS system.

[1]Currently wishful thinking
[2]Distributed Hash Table
[3]Also wishful thinking

PariPari currently offers several outer circle plugins:

- VoIP;
- Distributed Storage and Backup;
- Distributed Web-server;
- IRC client and distributed server;
- DNS;
- DBMS;
- ed2k and Kad client;
- BitTorrent client.

While also offering these inner circle plugins:

- Connectivity;
- Storage;
- DHT

1.2 PariCore

The modular architecture of PariPari relies on a central kernel: PariCore. Its main functions are managing plug-ins, routing their messages and protecting users and good plug-ins from malicious ones.

Managing plug-ins One of the main features of the Core is granting a high level of standardisation among plug-ins. All the services plug-ins provide are specified by a set of abstract classes: the APIs. Every API implements the super interface `paripari.API.API` (which contains, in particular, some methods needed by the Credit System) and must be extended by the specific class

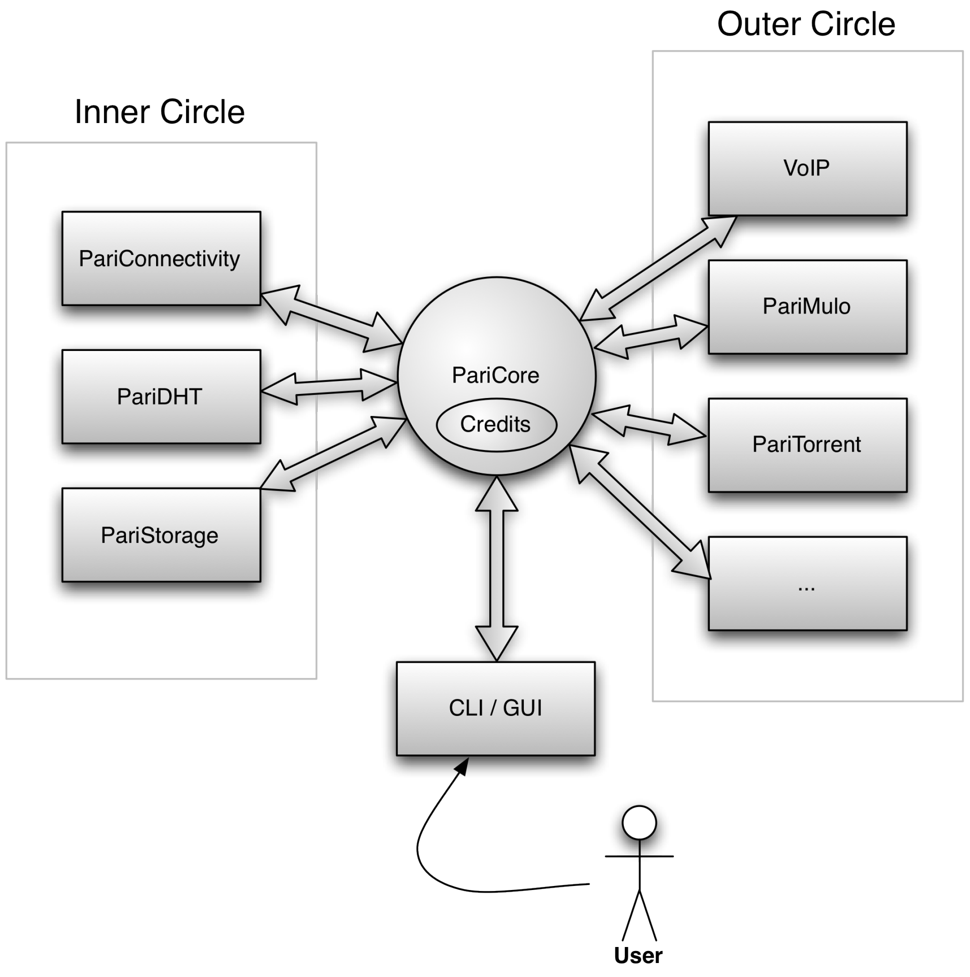

Figure 1.1: Architecture of PariPari

implementing the service. This distinction between definition and implementation makes it easier, in particular, for different plug-ins to provide the same service: interferences among such plugins (arising, for instance, when using identical class names) are avoided by instantiating a different class-loader for each plug-in. In every plug-in jar, the file (descriptor.xml) must be present: it lists plug-in dependencies as well as provided services.

Granting security PariCore manages security-related aspects by means of the PariPariSecurity-Manager, which replaces the standard Web Start's Security Manager. Only the inner circle plugins are trusted by default and are allowed to perform potentially dangerous operations, such as disk and network I/O; such inner plugins are developed by the PariPari Team and require strict signature checks before being executed.

1.2.1 Credits System

A dedicated module handles the Credit System of PariPari. It was developed as a separate entity from the Core but security reasons forced us to integrate it into the Core.

Roughly speaking, we can divide the whole Credit System into two layers:

- **Inter-peer Credits layer**: regulating communications among hosts;
- **Intra-peer Credits layer**: regulating communications among plug-ins lying in the same host.

The Inter-peer Credits mechanism shares some purposes with other usual P2P credit systems: namely, it aims at encouraging participation and discouraging freeloaders. It also has the ambitious goal of establishing a scalable, independent and, most importantly, transitive barter-based economy among peers.

1.3 Distributed Hash Table

The PariPari Network layout is based on a Distributed Hash Table (DHT) implementation that is based on Kademlia [12]: PariDHT, which provides a high degree of scalability, decentralisation and fault tolerance.

The classic mechanisms a DHT relies on are well known. Broadly speaking, each node is assigned a number in a d-bit address space: the nodeID. Each resource is represented by a key-value `(k,v)` pair: the key usually corresponds to a keyword associated with the resource itself and is mapped into a keyID, belonging to the same d-bit address space of nodeIDs, by means of a well defined hash function v (that is, `keyID = v(k)`). A specific notion of distance is defined among nodeIDs and keyIDs; in these regards, PariDHT adopts a value of d equal to 256, and the same XOR metric already employed in

Kademlia. Resources are stored and retrieved by nodes thanks to the shared address space, by sending your requests to the nodeID that is closest to the desired keyID.

1.4 Management and Development Techniques

Due to the high number of students that approach the PariPari project, and because of their high "churn rate" (students can work on PariPari for as little as a single semester), the need for a standardised development method and means for a common development environment quickly arisen.

Eclipse & SVN

Eclipse[4] is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, other programming languages. Eclipse includes a source code editor, a compiler and interpreter, and allows for advanced refactoring techniques, debugging and code analysis. Eclipse is released under the EPL[5]. Eclipse is being actively developed by The Eclipse Foundation, which is comprised of many software companies, among which are IBM, Oracle and Nokia. It is on a strict yearly release schedule: a new Eclipse version is released every june. Its releases are named after solar system bodies, and are ordered alphabetically since 2009.

Eclipse is currently "warmly[6]" suggested as IDE for PariPari development. As such, many guides on how to configure and proficiently use it have been produced by students over the years. Its seamless integration with SVN, the versioning control system currently in use by PariPari, grants an easy start to even the most inexperienced students.

[4]http://www.eclipse.org/

[5]Eclipse Public License. The Eclipse Public License is designed to be a business-friendly free software license and features weaker copyleft provisions than contemporary licenses such as the GNU General Public License (GPL)

[6]As in "use something else at your own risk entirely"

Subversion (often abbreviated as SVN after its command line name `svn`)[7], is a software versioning and a revision control system. PariPari plugins use SVN suggested source tree structure thus dividing their code base as trunk, branches and tags. Tags are final, published versions, the trunk is the current version in development and branches are used to separately implement various features before being merged with the trunk. Subversion is currently maintained by the Apache Foundation.

Another tool by the Apache foundation used in PariPari is ANT[8] which greatly simplifies the operations needed to accomplish in order to compile, package, sign and run a plugin. Due to PariPari's security design every plugin needs to be packaged up as a JAR[9] file, signed with the provided private plugin key and then run. Thanks to ANT, this can all be done in one click. Through an XML file (*build.xml*) every programmer can specify a set of operations to be carried out in order to run the plugin. Several optional operations can also be added, such as JavaDoc creation and automatic package upload. ANT is also perfectly integrated with Eclipse and requires no external plugin.

XP: eXtreme Programming

Extreme programming (XP)[10] is an agile software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. It advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints to verify the status of the ongoing development. It also encompasses several other techniques, which are only in part adopted by PariPari. The main one is TDD or Test Driven Development, which advocates continuous use of Unit Tests[11] not only to assess the quality of code and prevent bugs from emerging in later stages, but also as a design aid, that helps defining requirements, interfaces and implementations. Simplicity and continuous feedback, between

[7]http://subversion.apache.org/
[8]Another Neat Tool - http://ant.apache.org
[9]Java ARchive
[10]For more on XP: http://www.extremeprogramming.org/
[11]A unit is the smallest testable part of an application. As little as an individual method can and should be tested.

programmers, managers and, in enterprise environments, customers, are also key tenets of this philosophy. Pair programming, although not in the strict definition of two programmers sharing a workstation, is also sometimes used in PariPari. Students are paired and they each other test the other's code. This has although proved quite difficult to manage and is not currently enforced, although it is still suggested.

Team Management

XP suggests a flat management structure for teams following its philosophy. Due to the high number of students participating in PariPari and their low half-life, a completely flat structure isn't possible. At the head of the project there is the main Architect, currently Michele Bonazza, that manages teams at a high level, promotes communication and meetings when necessary and ensures deadlines are (somewhat) satisfied. The Master Tester, currently Alessandro Calzavara, ensures that testing is carried on properly and thoroughly. Testing is often neglected because it may be perceived as a waste of time, but has proved itself very valuable and is therefore strictly enforced. Students are then divided in teams, usually a team per plugin, each with its own Team Leader which is responsible for helping new students integrate, giving them directions on what should be done (and, more often than not, how it should be done) and keeping in check the overall plugin structure. They're also responsible for code base management, branches and subsequent merges. Team Leaders are chosen for their capabilities and willingness to commit themselves to the project for a longer period of time. Seldom plugin teams can be "confederated" into larger entities with shared goals as is currently the case for Messaging plugins (IM, IRC, VoIP).

2 PariConnectivity

To better understand how the old version of PariTorrent worked and what the new PariConnectivity is based on we will start this chapter by introducing how the Java NIO package works, focusing on its network components and buffers.

2.1 Java NIO

New I/O, usually called NIO, is a collection of Java programming language APIs that offer features for intensive I/O operations. It was introduced with the J2SE 1.4 release of Java by Sun Microsystems to complement an existing standard I/O. It addresses various complaints with the classic java.net package: mainly concerns with its poor performance with a high number of connections and its dated feature set. Thanks to its adoption of block-oriented network I/O through the use of buffers, managed by channels and selectors Java NIO satisfied these complaints and finally provided a capable and performing networking interface.

This section will introduce, with no pretense of completeness, the three key structures cited earlier: Buffers, Channels and Selectors. They are used both in PariConnectivity and the version of PariTorrent prior to the refactoring. For a more exhaustive treatise on Java NIO see [13].

Buffers

A buffer is a region of memory storage used to temporarily hold data while it is being moved from one place to another. It is useful to compensate for the different speed of two means or processes. Java NIO uses buffers as the building block for its improved infrastructure. Granting almost direct access to lower

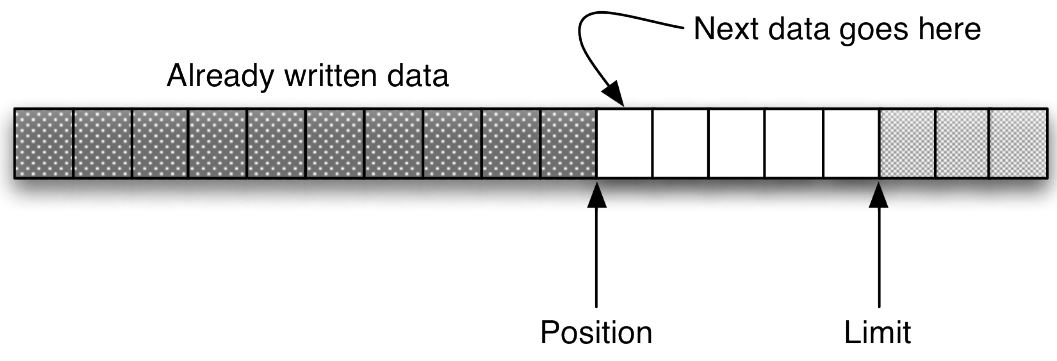

Figure 2.1: Buffer representation in Java NIO

level I/O operations of modern Operating Systems they allow for improved efficiency and performance. The improvements do not come with particular burdens, as NIO buffers can be accessed in a totally random fashion, just like common arrays, also providing simple, and yet powerful, primitives for bulk data access.

In the NIO implementation a buffer can be seen as an array with two pointers: *position* and *limit* (Figure 2.1). A buffer's limit is the index of the first element that should not be read or written. A buffer's limit is never negative and is never greater than the its capacity. A buffer's position is the index of the next element to be read or written. Bulk read (*get*) and write (*put*) operation calls should thus be preceded by two provided methods:

`flip()`: makes a buffer ready for a new sequence of channel-write or get operations: It sets the limit to the current position and then sets the position to zero;

`clear()`: makes a buffer ready for a new sequence of channel-read or put operations: It sets the limit to the capacity and the position to zero;

that prepare the buffer. The buffer class in Java has various specializations: one for each of the primitive data types, but for our kind of applications only ByteBuffers were used.

Channels

Channels are designed to provide bulk data transfers to and from NIO buffers. They're a low-level data transfer mechanism that exist in parallel with the classes of the higher-level I/O library (packages `java.io` and `java.net`). They're not a specialization of that model, but a new implementation altogether. A channel object can be obtained from a high-level data transfer class such as `java.io.File`, `java.net.ServerSocket`, or `java.net.Socket`. Channels can be requested in blocking or non-blocking mode. In blocking mode, every I/O operation invoked upon the channel will block the caller until it completes (as it is usually the case with java.net objects). In non-blocking mode, an I/O operation will never block, even transferring fewer bytes than were requested or possibly no bytes at all. One of the advantages of asynchronous I/O is that it allows to do I/O from many inputs and outputs at the same time: one can listen for I/O events on an arbitrary number of channels, without the need for polling and without unnecessary threads managing them.

We will concentrate now on SocketChannels, which are the most prominently used in our particular application. A SocketChannel provides methods for establishing a connection with a remote computer and transfer data to and from it. To connect to a server socket you simply call:

```
boolean connect(SocketAddress remoteAddress)throws IOException
```

on the SocketChannel. If the connection can be immediately established that method returns true, while returns false if it needs to be completed in a second moment. In that case invoking

```
boolean finishConnect() throws IOException
```

at a later time effectively completes the connection sequence.

Reading and writing to a channel can be achieved by means of two straightforward methods:

```
int read(ByteBuffer destinationBuffer) throws IOException
int write(ByteBuffer sourceBuffer) throws IOException
```

In blocking mode, the read method waits until at least one byte is available, while the write method waits until all the data remaining in the buffer is sent; in non-blocking mode, the call immediately returns. In the latter , if either no data is available in the socket input buffer or the socket output buffer is full, read and write methods return a value equal to 0.

A ServerSocketChannel is also used to listen for incoming connections. It works pretty much in the same way of a normal server socket, but conveniently returns the socket channel connected with the remote host requesting the connection.

All of these channels (and others) extend the SelectableChannel superclass and can therefore be multiplexed through a *Selector* object.

Selectors

A selector provides a mechanism for waiting on channels and recognising when one or more of them becomes available for data transfer. When a number of channels are registered with the selector, it enables blocking of the program flow until at least one channel is ready for use, or until an interruption condition occurs. Although this multiplexing behaviour could be implemented with threads, the selector can provide a significantly more efficient implementation using lower-level operating system constructs. For the selector to know which channels to check for availability they need to be *registered* with it. This can be obtained by calling:

```
SelectionKey register(Selector sel, int ops)
```

The SelectionKey object also allows us to attach an object to the key. In PariTorrent, for instance, this was exploited by attaching a *peer* object, allowing us to keep trace of what peer that particular channel belonged to.

By invoking the

```
int select()
```

method on the selector object we select the channels that are ready (it also returns the number of available channels) on the selector object. Following it with the

```
Set selectedKeys()
```

method, we obtain the subset of registered channels that have data available to be read. Enabling us to easily process whatever data is available on those channels. The `select()` call is blocking, thus effectively pausing the thread when there is nothing to be read. Every single SelectionKey, has four type of events, defined by constants: `OP_READ`, `OP_WRITE`, `OP_CONNECT` and `OP_ACCEPT`, that represent each of the possible states of the socket/channel.

2.2 PariConnectivity

PariConnectivity has been designed by Francesco Peruch and his team [10] to overcome most of the flaws of standard Java I/O while ensuring backwards compatibility to plugins using it and yet providing a simple migration path for plugins wanting to use the New I/O library. It wraps around the design paradigms introduced in the first section and abstracts them even further by presenting them in a new and developer-friendly way. PariConnectivity doesn't rely on select loops, such as the ones presented earlier, but uses *Completion Queues* instead. By using a completion queue, I/O requests can be issued asynchronously, while notifications of completion are provided sequentially in a blocking queue and processed accordingly.

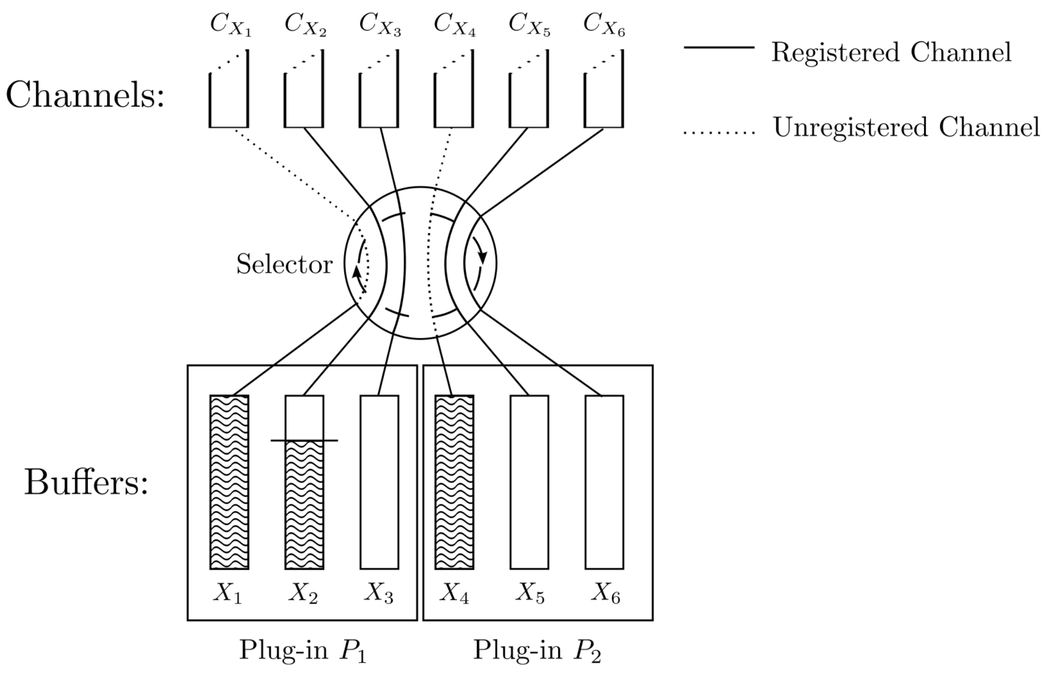
C_{X_1} C_{X_2} C_{X_3} C_{X_4} C_{X_5} C_{X_6}
Channels:
Registered Channel
Unregistered Channel
Selector
Buffers:
X_1 X_2 X_3 X_4 X_5 X_6
Plug-in P_1
Plug-in P_2

Figure 2.2: Connections management in PariConnectivity

2.2.1 Internals

When a plugin requests a socket for data transfer through one of the provided APIs, PariConnectivity translates the request into a sequence of basic I/O operations and starts performing them. The channel involved in it is registered with a selector, and a thread periodically executes a selection process on that selector. When that channel is ready another channel, for a second operation (if different from the first), is registered. All the channel operations waiting for completion are registered with the same selector, and monitored by the same thread. In such a way, the number of threads instantiated is dramatically reduced, without affecting performance: every elementary I/O operation simply corresponds to a non-blocking call made on a single channel.

The asynchronous I/O components wrap around the previous constructs and expect for every outbound socket created a BlockingQueue for each of the three type of events possible, that is a “ReadQueue” a “WriteQueue” and a “ConnectQueue”. Conversely Non Blocking Server Sockets work over an “AcceptQueue”.

2.2.2 APIs and Interaction

PariConnectivity offers several API that allow plugins to interact with the network in many different ways. It offers Blocking and NonBlocking sockets through TCP and UDP, a wraparound URLConnection that offers convenient HTTP sockets and finally Blocking and Non Blocking TCP server sockets. It also provides simple and powerful means of communications with other PariPari nodes through Tunnelling and NAT Traversal services, but this is outside the scope of this thesis work.

While interaction with blocking sockets is kept consistent with the classic java implementation (even though it has to pass through PariPari constructs), asynchronous I/O is quite different. For asynchronous I/O operations a generic plug-in P has to instantiate a queue Q implementing the BlockingQueue interface; such an interface is used by Connectivity in order to notify the plug-in itself about the accomplishment of the requested operations. Each call to asynchronous API's methods (like `send()`, `receive()`, `connect()` etc.) forces the calling plug-in to specify a notification queue Q and an object O (implementing the AsynchronousNotification interface), through which it will be notified. O has to provide a `process()` method, whose code contains all the operations that need to be performed after the notification is received. Further explanation of how this mechanism was implemented in PariTorrent is available in chapter 5.

3 BitTorrent

3.1 Protocol Overview

The BitTorrent protocol was created in april 2001 by Bram Cohen. It is different from other P2P networks in that it isn't based on a shared, centralised and server-based overlay network but is almost completely decentralised. Its initial scope was aiding web-servers in the distribution of hefty files by exploiting the peer's outgoing bandwidth. While this use is still popular today, it was nevertheless quickly adopted for piracy means, allowing quick distribution of movies, games and music. Due to its popularity it now accounts for at least 43% of all internet traffic, with some estimates going as high as 70%.

The protocol requires the user to obtain outside of the network a metadata container called the torrent file. That file can be distributed by various means, but is usually found on the creator's webpage or through dedicated websites called BitTorrent indexes, which aggregate torrents from various (often anonymous) sources. The container includes everything that is necessary to identify the file(s) we want to download and the means to connect to other peers. The original protocol required the torrent file to include, among other things (mainly different hashes, more at section 3.2) a *tracker*, that is an URL to an HTTP server to which every peer must report whenever he wishes to start or stop a download that is managed by that tracker. When contacted for the first time the tracker answers with a random, limited list of the peers that are currently downloading or seeding[1] that torrent, allowing the peer to connect to them directly. The peer then proceeds autonomously, periodically reporting back to the tracker.

[1]Sharing after they completed their download.

The need for trackers is now overcome with the use of direct exchange of peer contact information between the peers themselves (Peer Exchange) and the use of a shared DHT to obtain the initial torrent metadata.

The actual content is downloaded in pieces of variable size (but fixed for each torrent), which are themselves requested in blocks of 16KB to various peers.

3.2 Torrent Files

As stated earlier, the torrent file is a metadata container that includes everything that is needed to download that specific packet of files. It is simply a text file, encoded through a technique called *Bencoding*. Bencoding is also used for communications between peer and tracker.

Bencoding provides a (bit convoluted) way to encode four data types:

- Integers;
- Strings of Bytes;
- Lists;
- Dictionaries.

While it is outside of the scope of this thesis work, bencoding is explained in detail in my first one [5].

3.2.1 Metadata

Torrent files need to include at least this information:

- **announce**: the URL of a tracker, encoded as a string;
- **info**: a dictionary that describes the files that can be downloaded using that torrent file. It can be in two distinct formats: for a single file and for multiple files.

Other, non mandatory data can be:

- **announce-list**: a list that includes several trackers for the peer to try, if one is not available;
- **creation date**: unix timestamp of the date when the torrent was created;
- **comment & created by**: self explanatory fields, encoded as strings.

The info dictionary can be in two modes: Single File Mode and Multiple Files Mode. In both cases it has to include this information:

- **piece length**: nominal piece size, in bytes. It can be freely chosen, keeping in mind that using chunks that are too big can lead to inefficiency (in case the piece we downloaded is corrupt and we need to download it again) and may cripple the network, limiting the peers' bartering abilities, while too little ones can make the size of the torrent file grow too much and put excessive strain on web-servers offering them. Popular lengths are 256KB, 512KB and 1MB;
- **pieces**: concatenated SHA-1 hash of all the pieces [2]

It also contains **name**, **length** and an optional MD5 **checksum** if in Single File Mode, while it contains **name** (that refers to a folder though) and a dictionary containing relative **path** and **lengths** of all the files to be downloaded if in Multiple Files Mode.

3.3 Trackers

A tracker is an HTTP or HTTPS web-server that answers to standard GET requests. Parameters are url-encoded and appended to the url string using standard CGI methods (appending '?' to separate the url from the parameters and separating each value with '&'). The tracker replies with a bencoded plaintext file.

[2]SHA-1 produces 20 bytes hash, allowing us to simply split the resulting string. SHA-1: http://en.wikipedia.org/wiki/SHA_hash_functions#SHA-0_and_SHA-1

3.3.1 Client requests

These are the main requests a tracker understands and requires to answer correctly:

- **info_hash**: the info hash is the only way to distinguish between different torrents. It is the hash of the bencoded info dictionary and therefore uniquely identifies it;
- **peer_id**: 20 byte peer identificator usually in `-XXVVVV-RRRRRRRRRRRR` format. XX stands for the type of client (e.g. AZ is Azureus Vuze, PP is what we chose for PariPari), VVVV for the version number while R are just random numbers. Its uniqueness is not enforced;
- **port**: incoming connections port;
- **event**: signals starting, pausing, or finishing a download;
- **uploaded, downloaded** and **left**: unchecked statistics.

Optionally through the **num_want** parameter a client can specify the amount of peers it wishes to get back. A **compact** optional boolean parameter is also available, if set to true it changes the way the tracker lists peers in its answers.

3.3.2 Tracker answers

The bencoded answer contains the following data:

- **failure reason** or **warning message** (optional): error messages from the tracker, failure reason prevents the client from proceeding, while warning messages can be ignored;
- **interval** and **min interval**: respectively time waiting interval and mandatory minimum time interval to wait between requests;
- **tracker id**: tracker identifier string, to be sent on next requests;
- **complete** and **incomplete**: number of seeders and leechers;

- **peers**: in canonical format the peer field is a list of dictionaries. Each of them contains:
  - **peer id**: 20 byte peer identificator;
  - **ip**;
  - **port**.

Compact format simply lists peers as 6 bytes strings: 4 bytes for IP address, 2 bytes for port. This compact representation is the most widely used, with some trackers going as far as not supporting the canonical format.

3.4 Peer Protocol

Peer communication uses a well defined protocol, the so called Peer Protocol. It uses TCP to exchange simple coordination messages and data.

3.4.1 Choking

Before we delve into the actual protocol we need to establish one of the mechanics that makes BitTorrent work. The protocol defines two mandatory boolean statuses for each peer: *choked* and *interested*. Choking a peer means that we won't send any message (save for the keep-alive), until he gets to be unchoked. A peer is interested if he wants to receive one of the pieces we currently have. A peer can signal its intentions by using the appropriate messages. Each peer starts choked and not interested. Conversely we also are either choked or unchoked by that peer and interested or not interested in what it has to offer. Thus leading us to have for each peer four boolean variables:

- am_interested;
- am_choking;
- peer_interested;
- peer_choking.

When we're interested (am_interested is true) and unchoked by the peer (peer_choking is false) we can start sending requests for pieces. When the peer is interested and unchoked he may do the same.

Unchoking is managed through a peculiar algorithm that can be found in [5].

3.4.2 Handshake

The first mandatory message is the handshake that identifies the peer and allows us to correctly populate its information. It is in the following format:

```
<pstrlen><pstr><reserved><info_hash><peer_id>
```

that is:

- **pstrlen**: length of pstr in bytes (always 19);
- **pstr**: protocol identifier string (always "BitTorrent protocol");
- **reserved**: 8 reserved bytes for extensions and future use;
- **info_hash**: SHA-1 hash of the info dictionary, as usual this is used to uniquely identify the torrent;
- **peer_id**: 20 bytes string that identifies the peer (see 3.3.1).

3.4.3 Messages

Actual message exchange can start after the handshake. A message is in the following format:

```
<length><message ID><payload>.
```

Length is 4 bytes long and encodes the total length of the message, both message ID and payload. Message ID is a single byte representation of the type of message sent. The payload is, of course, heavily dependent on the type of message.

keep-alive is the simplest message, being merely a message of 0 length and no ID or payload. **choke**, **unchoke**, **interested** and **not-interested** are also very bare, being messages of length 1 and increasing single digit id, respectively 0, 1, 2 and 3. We won't describe in detail every possible message in the PeerProtocol, such a description is available at [5]. It suffices to know that the protocol lets us send and receive a **bitfield**, that is a boolean array of all the pieces that the peer possesses, a more specific **have** message to signal the possession of a single piece, to be sent to peers after a client completed a piece download to signal its new availability, a **request** message to ask for a piece in 16KB chunks (usually called a *block*), and a **piece** message, to send the actual data. Please note that a *piece* message actually sends a *block* of data, not the entire piece. When all the blocks belonging to a piece are received the whole piece is hashed and compared to the given SHA-1 hash in the torrent metadata file.

3.5 Protocol Extensions

The efficiency and popularity of the BitTorrent protocol spawned several extensions. Some of them are recognised as official and organised in three tiers in descending order of official acceptance: Final, Accepted and Draft. The official repository for BEPs (BitTorrent Enhancement Proposals) is available at *http://bittorrent.org/beps/bep_0000.html.* The strict accepting procedure and specific refusals that verge on the philosophical, more than actual technical nature of proposed extensions has lead to a parallel set of extensions that are as widely used as the official ones. Message Encryption for instance is now de-facto standard and supported (and sometimes even enforced) by all the major BitTorrent clients despite never actually achieving any kind of official acceptance.

3.5.1 PariTorrent currently supported extensions

There are no actual proper extensions that made it to final status yet. The only ones are the protocol specification itself and some naming conventions. There are also only two accepted extensions, PariTorrent supports both.

3.5.1.1 Accepted Extensions

Extension for peers to send metadata files The purpose of this extension is to allow clients to join a swarm and complete a download without the need of downloading a .torrent file first. This extension instead allows clients to download the metadata from peers. Finding content is made possible thanks to magnet links: a link on a web page only containing enough information to join the swarm (mainly the info-hash). This extension is pretty much useless without DHT support (which is still a draft extension), but is nevertheless implemented in PariTorrent, pending future DHT support. It also uses the libtorrent Extension Protocol to convey its messages (which is also present as draft).

Tracker returns compact peer lists Specifying `compact=1` in tracker requests returns a compacted peer list. This is now de-facto standard in tracker communication. See also 3.3.1.

3.5.1.2 Draft Extensions

While only a handful are actively used, a great many draft extensions are present. PariTorrent currently supports only some of them.

Fast Peers Extension The Fast Extension packages several new messages: **Have None**/**Have All**, **Reject Requests**, **Suggestions** and **Allowed Fast**. These are enabled by setting the third least significant bit of the last reserved byte in the BitTorrent handshake:

```
reserved[7] |= 0x04
```

The extension is enabled only if both ends of the connection set this bit. Have All/Have None messages can be sent in place of the bitfield and signal, respectively a full piece set or an empty one. Suggestions allow a peer to suggest a piece to download to a peer, to compensate for the perceived scarcity of that piece in the known network. While an uploading client can send an Allowed Fast message to allow a peer to ask for a small subset of specified pieces even if it is currently choked, to help bootstrap newly connected peers. Fast Extensions use normal Peer Protocol messages, using new message IDs to identify the new ones. For more information on how this was implemented check [6]

libtorrent Extension Protocol The intention of this protocol is to provide a simple and thin transport for extensions to the BitTorrent protocol. Supporting this protocol makes it easy to add new extensions without interfering with the standard protocol or clients that don't support this extension or the one you want to add. This extension is enabled by setting

```
reserved[5] |=0x10
```

in the handshake message. This adds yet another message with id 20 to the PeerProtocol standard set. This type of message embeds all kind of messages that can be sent through this new protocol. Currently it supports two kind of messages: `ut_pex` and `LT_metadata`. The messages supported need to be specified in a separate protocol handshake, which is simply a bencoded dictionary of the name of the supported messages. LT_metadata effectively implements the "Extension for peer to send metadata files" extension, while ut_pex provides exchange of peer contact information. More information on this extension and its PariTorrent implementation is available at [6, 15]

Multitracker Metadata Extension In addition to the standard **announce** key an **announce-list** key can be present in a torrent metadata file. This key refers to a list of lists of URLs, tiered by priority that the BitTorrent client can contact to receive peer information. It is used by almost every torrent file. See also 3.3.1.

Superseeding When superseeding is active a different algorithm for seeding is used. Instead of sending the full bitfield or a Have All message, a peer in superseeding mode will “guide” the distribution of pieces by advertising only selected ones. This can aid in evening out significant fluctuations in piece distribution among the known peers, effectively helping the global network health.

3.5.1.3 Unofficial Extensions

Message Stream Encryption / Protocol Encryption MSE/PE is a cryptographic encapsulation protocol that aims to make BitTorrent traffic unrecognisable by Internet Service Providers, thus preventing bandwidth throttling. It uses a lightweight cryptographic algorithm to encrypt the whole message exchange between peers (or optionally the header only). It is not designed to be intrinsically secure, but to offer quick means of traffic obfuscation and deobfuscation that are easy on computing power. It is also designed to provide limited protection against active Man in The Middle attacks and port-scanning by requiring a weak shared secret to complete the handshake. It mainly uses a Diffie-Hellman key exchange to establish a shared encryption key and then uses RC4’s symmetrical properties to encrypt and decrypt the stream. The resulting message exchange appears effectively as a completely random stream.

Real effectiveness of this method has been questioned several times. Some ISPs are now using more sophisticated measures to detect BitTorrent traffic. This means that even encrypted BitTorrent traffic can be throttled. Analysis of MSE/PE has shown that statistical measurements of packet sizes and packet directions of the first 100 packets in a TCP session can be used to identify the obfuscated protocol with over 96% accuracy.

The sheer number of peers that use and enforce this extension makes it mandatory for every client that tries to compete for a, albeit slim, slice of market share. For a complete treatise on how this works on a low level and how it was implemented in PariTorrent you can have a look at [8, 16].

Azureus Messaging Protocol The AZureus Messaging Protocol (AZMP) is a completely different set of messages that takes over the original PeerProtocol. By setting to 1 the most significant bit in the traditional handshake a client declares its support for the feature. If both peers support it they should switch to this communication method. The new format for messages is:

```
<pLength><nameLength><pName><messageVersion>
[<paddingLength><padding>]<payload>
```

Where:

- **pLength**: packet length in bytes;
- **nameLength**: length of the message type identifier;
- **pName**: message type, as a string;
- **messageVersion**: version of the message type, (not of the whole protocol); if the fourth most significant bit is set, a padding section is expected after this field;
- **paddingLength** and **padding**: padding is optional but can aid in cryptography operations;
- **payload**: well, it's the payload.

The AZMP requires another handshake to be sent after the original one, containing additional information, such as the whole set of supported messages in a bencoded dictionary. Being a complete replacement of the original protocol, the AZMP also encapsulates the original PeerProtocol messages by preceding them with a "BT_" prefix. For instance a normal unchoke is defined as `BT_UNCHOKE` while a keep-alive is `BT_KEEP_ALIVE`. It also includes new message types: `AZ_HANDSHAKE`, `AZ_PEER_EXCHANGE`, `AZ_REQUEST_HINT` which is akin to suggestions in the fast peers extension, `AZ_HAVE` which allows for a more precise, down to the block, piece communication and `AZ_BAD_PIECE`, that signals to its owner that he may have sent a corrupted piece. For more information on AZMP and its PariTorrent implementation check [14, 7].

3.5.2 PariTorrent planned support extensions

We plan to add these features in the upcoming releases as they're fundamental for a complete BitTorrent client.

DHT

Due to national policies in attempts to thwart piracy, BitTorrent trackers and indexes are becoming increasingly rare. By taking advantage of the fully decentralised approach of a DHT overlay network, and using magnet links and peer exchange this extension allows clients to fully bypass the need for trackers. The BitTorrent Distributed Hash Table specification is currently available in draft form and is widely adopted. The draft is based on Kademlia, just like the PariPari DHT described at chapter 1.3. The underlying mechanism is quite simple: an user fetches a magnet link through various means, gets through bootstrap phase (usually transparently managed by the client), contacts the peer that is responsible for that resource and gets back an initial set of peers, which he can then use to get to other peers through peer exchange. The support for DHT is of primary importance as is used ever more often, with some torrents being available only through its use. Azureus Vuze offers a different DHT implementation that is not compatible with the official specification, while being the first one to emerge, its usage is steeply declining, and its support is not planned.

UDP Tracker Protocol

Using HTTP to transfer information between clients and trackers introduces significant overhead. There's overhead at the ethernet layer (14 bytes per packet), at the IP layer (20 bytes per packet), at the TCP layer (20 bytes per packet) and at the HTTP layer. To send a full request and receive a response of about 50 peers, 10 packets are needed and about 1200 bytes of data exchange. This overhead can be reduced significantly by using an UDP based protocol. The UDP tracker protocol uses 4 packets and about 600 bytes, reducing traffic by 50%. For a client, saving 1KB every hour isn't significant, but for a tracker serving millions of peers, reducing traffic by 50% can matter a lot. Many trackers, while still supporting the old HTTP method, are quickly moving to UDP. Among others, OpenBitTorrent, a free, open and popular tracker, recently switched to UDP only communication, making this feature a top priority.

3.5.3 Other Extensions

Some other extensions are gaining steam with the BitTorrent community, one of them - µTorrent Transport Protocol (µTP or uTP) - tries a new, interesting approach at data transfer between peers. While its support is not yet planned it's still worth noting.

µTorrent Transport Protocol

µTP seeks to exploit UDP's distinctive features (mainly its being connectionless and its lack of congestion control) to better shape BitTorrent traffic, offering integrated QoS support and punching through NATs and firewalls while offering the same, or better, throughput to its TCP counterpart. µTP reimplements parts of the features TCP offers adapting them as needed. A 3-way handshake is for instance recreated, as well as ACKing mechanisms. Congestion control is also reimplemented, but is based on increasing delay between packets much more than on packet loss. If other TCP streams try to use bandwidth and effectively slow down the µTP message exchange the latter will yield back, leaving to higher priority TCP packets (such as HTML pages, e-mails and other user related content) most of the available bandwidth.

Tests on this protocol have been quite controversial, with some claiming it is very effective and others claiming it only disrupts internet traffic further while offering no real benefit. Being fostered by the most popular BitTorrent client: µTorrent, it is nevertheless quickly being adopted.

4 PariTorrent

To better understand how PariTorrent's structure came to be what it is now we will briefly introduce how the plugin structure was when it was first introduced. The main classes are still the ones which, through many modifications to allow for extensions or other improvements, are in use today. This is far from complete, for a more thorough dissertation you can read [5, 4], and further modifications applied to the original source base: [6, 7, 8, 9].

4.1 Original Structure

We can identify two core parts in the PariTorrent plugin: PariPari interaction and BitTorrent protocol implementation. PariPari interaction is handled through some core classes which remained mostly untouched through the years. PariTorrent uses PluginSender to conveniently manage resource requests and renewals (the PariPari credits system enforces that every resource needs to have an expiration time). For instance:

```
  ps.requestSingleTCPNonBlockingSocketAPI(
new IFeatureValue[] { new FeatureValue("time", 60000) }, true)
```

uses PluginSender to request a TCP Socket in non blocking mode. The array of FeatureValues is how the credits system knows how to bill for this resource, i.e. this socket will be billed for the time feature, for a minute. The second parameter specifies whether to auto-renew this resource or not.

The TorrentListener class listens for possible incoming messages from the Core and acts accordingly. While TorrentCore, the main class, is the one that

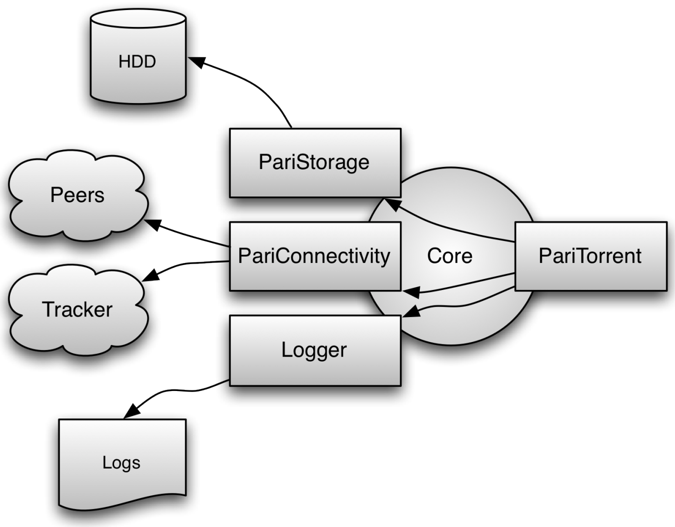

Figure 4.1: Resource access in PariPari

handles startup and shutdown of the plugin and, through TorrentConsole, input from the user.

Every torrent added by the user used to get its very own DownloadManager instance that created and managed everything that is necessary to download that file. To get the gist of how download manager used to work it mainly started two threads: RemotePeerListener and PeerListUpdater which, respectively, opened a server socket to listen for incoming connections and managed the mandatory periodical connection with the Tracker.It also instantiated a FileManager that managed disk operations. With the peers it received from the tracker it instantiated a further set of threads, called *Tasks*, one for each peer, which carried on the necessary operations with each peer through two ohter threads: MessageSender and MessageReceiver, which, evidently, managed outbound and inbound Peer Protocol messaging. DownloadManager managed global torrent operations, such as the peer unchoking while DownloadTask was the real workhorse, sending and receiving pieces and messages according to the peer status and the shared FileManager. The Peer itself was a mere data structure, with no function other than keeping information about the peer's capabilities.

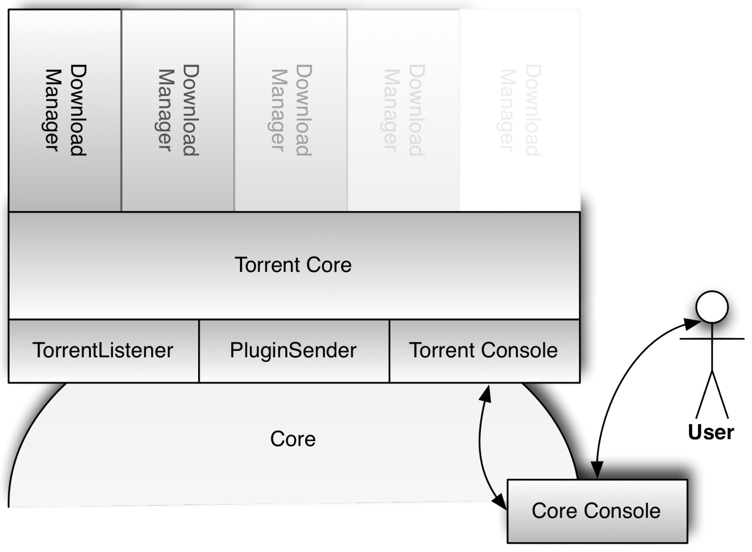

Figure 4.2: PariTorrent base structure and interactions

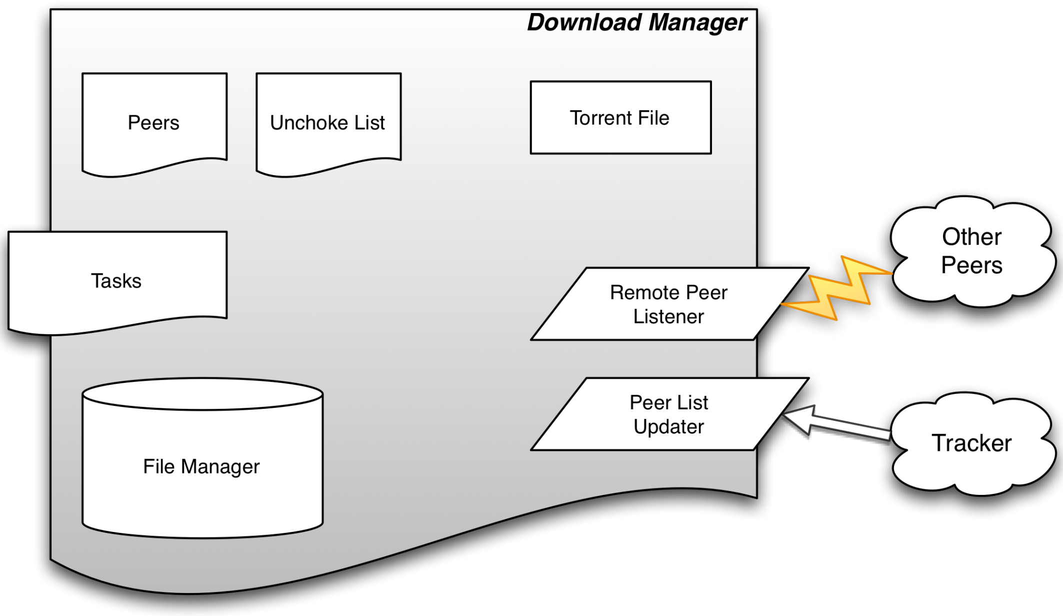

Figure 4.3: DownloadManager's Structure

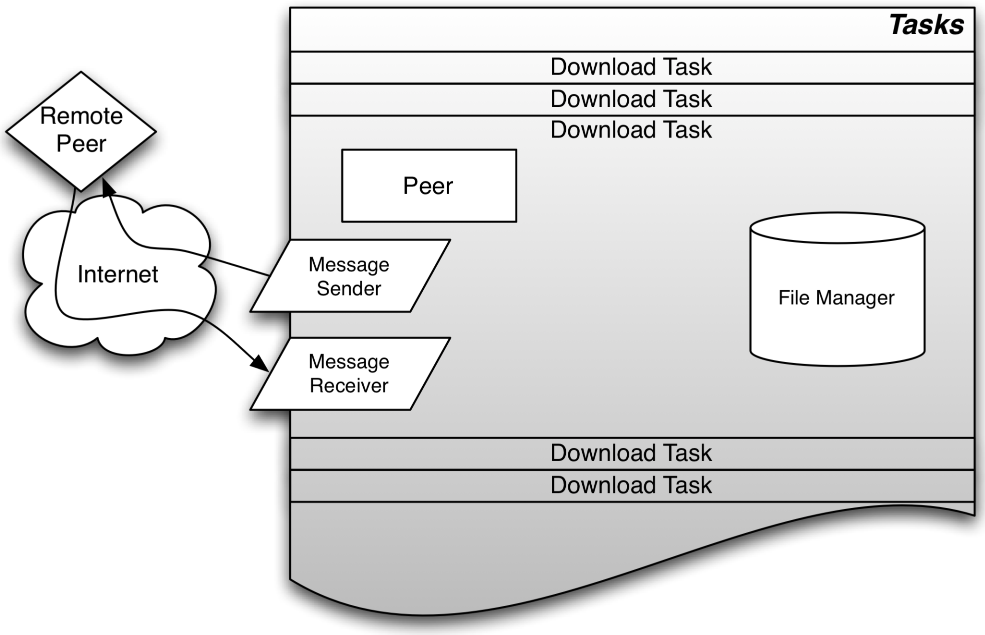

Figure 4.4: Tasks Structure

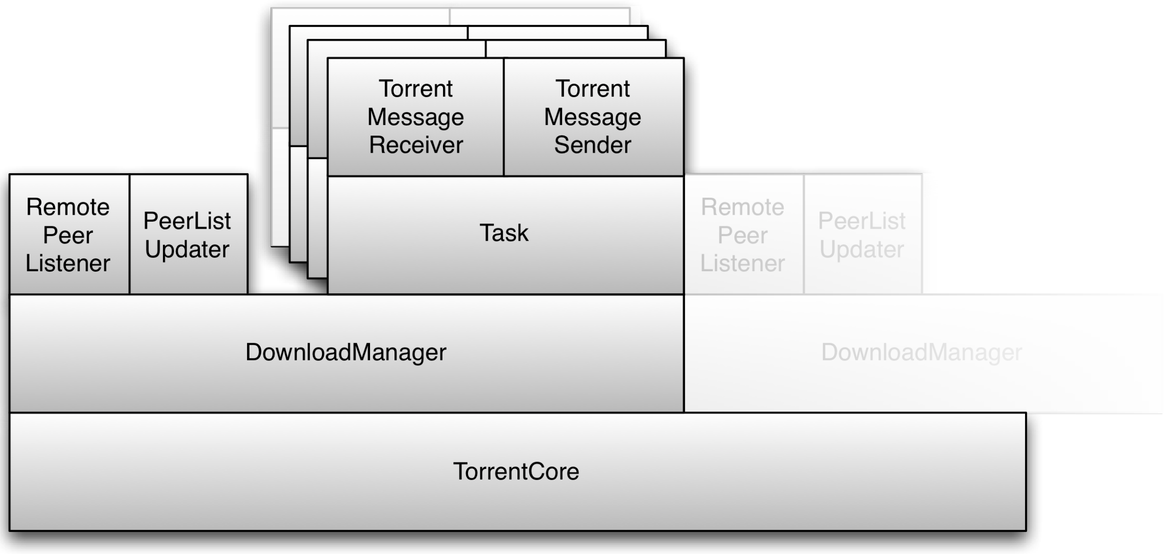

Figure 4.5: Threads structure prior to NIO refactoring

As you can probably see, this quickly led to a huge number of threads needed for merely maintaining the bare minimum communications with connected peers. Each active torrent had two global threads (PeerListUpdater and DownloadManager) plus three threads for each peer. Couple this with a bit of reckless programming due to inexperience and your system was quickly rendered unusable just by trying to download a torrent.

4.2 Structure after Java NIO adaptation

Thanks to the asynchronous nature of Java NIO's new constructs, the need for a MessageSender and a MessageReceiver thread per peer was easily overcome. Therefore the entire *Tasks* structure the previous version was based on had no reason to exist. Peer was upgraded, as it now contains also its socket, its related buffer, its in and out message queue and also, more coherently, its own status, which was previously managed by DownloadTask. This led to DownloadTask and DownloadManager being merged in what is now the DownloadTorrent class. MessageSender and MessageReceiver had of course to

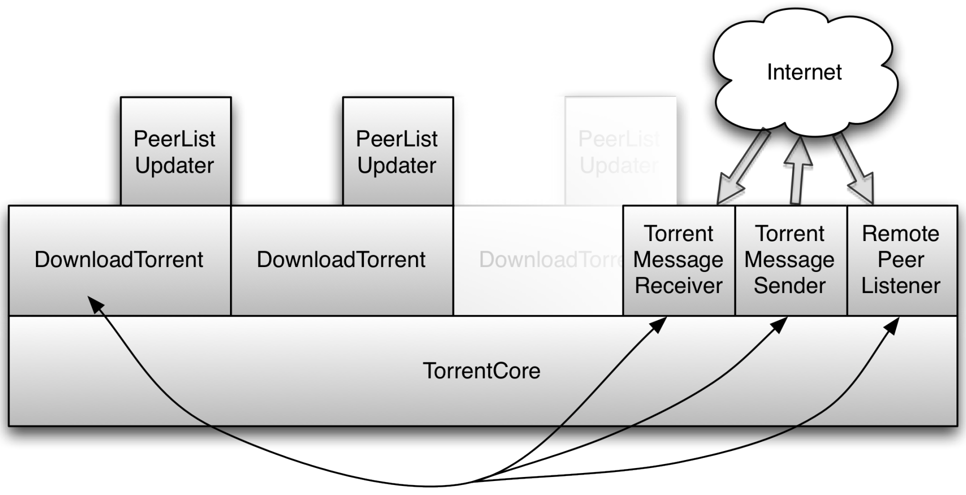

Figure 4.6: Threads structure and interactions after NIO refactoring

be removed but got each a new life as threads managed by DownloadTorrent and were the harbours to pretty much most of the refactoring that took place.

TorrentMessageReceiver used a Selector to iterate over its registered channels. To register a new peer and its channel, the methods `registerPeer(IPeer peer)` and `registerConnectedPeer(IPeer peer)` were called by DownloadTorrent when either starting a new connection or wrapping one received from RemotePeerListener - the inbound connection listener. The selector cycle checked whether the connection was fully established or waiting to be completed and in this case called `finishConnect(IPeer peer)`. If it was ready, all the data was copied over to the peer's ByteBuffer, and called the analyzeBuffer method. For the sake of clarity, let's assume that Message Encryption has been disabled. (For more information on Message Encryption see 3.5.1.3)

`analyzeBuffer(IPeer peer)` read the contents of the buffer block by block and attempted to create, based on the state and capabilities of the peer, a Message object, that is an object representation of the BitTorrent package received. If we were still expecting the handshake, the `readHS()` method was called, which parsed the packet and fully populated the necessary peer information, `readPP()` and `readAZ()` were called if the peer was using respectively plain PeerProtocol or Azureus Messaging Protocol, which are completely different. If there was not enough data to parse, the buffer was compacted and restored. When further data was received it would have been parsed again.

TorrentMessageSender iterated through its registered peers and simply sent messages ready in each peer's queue through their own socket.

RemotePeerListener was also adapted to use ServerSocketChannels and moved from its original position in DownloadManager, it was no more directly bound to the torrent it was referring to, but was now rendered global, thus removing the need for more than one open listening port and further diminishing the number of active threads. This required RemotePeerListener to handle handshakes from remote peers itself, as it was impossible to distinguish to which active torrent they were trying to connect to before receiving the data contained in the handshake message. The resulting socket channel was then registered with the message sender and receiver selector.

4.3 Support classes

Many other classes comprise the PariTorrent Plugin. Most of them weren't touched (or were only marginally modified) by the NIO refactoring, but are nevertheless instrumental in comprehending how the plugin currently works.

FileManager FileManager is responsible for disk access and file operations. It manages the whole piece array and carries the most current information on pieces availability and status. It is therefore responsible for checking piece integrity when they're completely downloaded and for deciding which piece to ask for next. When a piece is complete and correct it is saved to disk, paying attention to the peculiar system the BitTorrent Protocol uses for multiple files. Multiple files are treated just like one big, concatenated file, and thus pieces (and, at a lower level, blocks) can span across multiple files (see Figure 4.7)

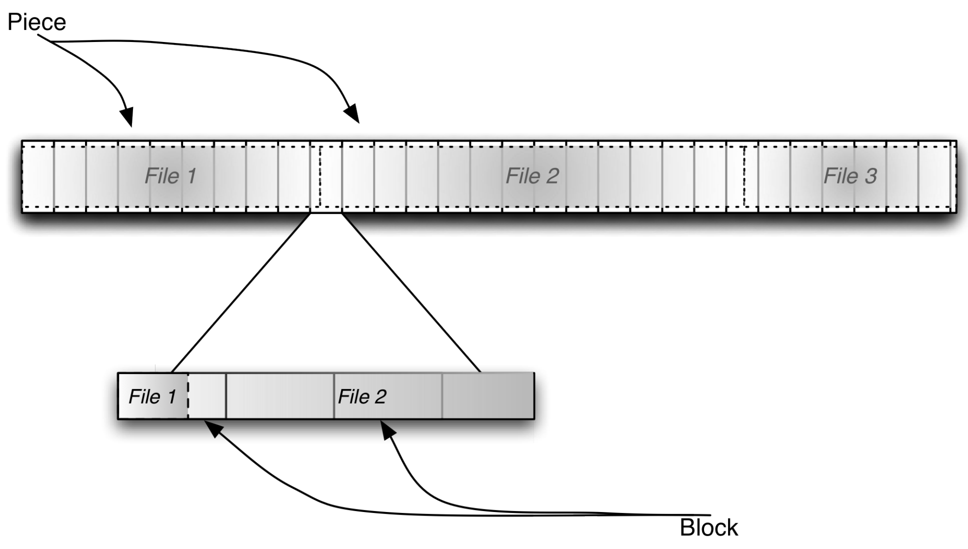

Figure 4.7: Piece, file and blocks management

Other packages

- **config** contains useful constants and the XML reader/writer needed to interpret the configuration file TorrentConfig.xml;

- **crypto** contains cryptography support classes, such as the RC4 engine;

- **ltep** mainly manages ltep based peer exchange;

- **peer.messages** contains every message in object form. Each one of them also provides a `generate()` method that prepares the byte array that is to be sent. Its subpackages `peer.messages.azmp`, `peer.messages.fastextensions` and `peer.messages.ltep` implement the messages of the related protocol extension;

- **piece** is a file manager support package, it's a data structure that manages piece informations such as index, completion, and also hosts a temporary buffer for the incomplete data;

- **util** is a collection of utility methods, such as the ones needed to interpret and created bencoded values, data format conversions and hash functions.

4.4 Considerations

The NIO refactoring allowed us to consistently reduce the number of threads (see Figures 4.8 and 4.9) and memory footprint (see Figures 4.10 and 4.11).

Our careful design took into consideration the future PariConnectivity implementation. While at the time of this refactoring the new PariConnectivity should have been a mere mimicry of the structures used by the original Java NIO and it actually came out to be quite different, this asynchronous peer management implementation paved the way for what will be presented in the next chapter.

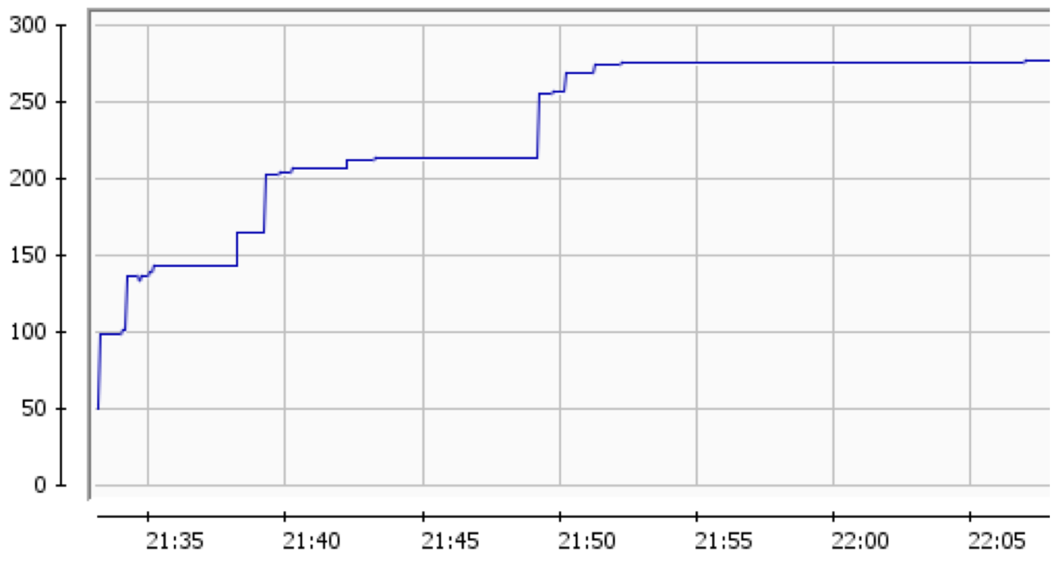

Figure 4.8: Number of threads in the first version of PariTorrent

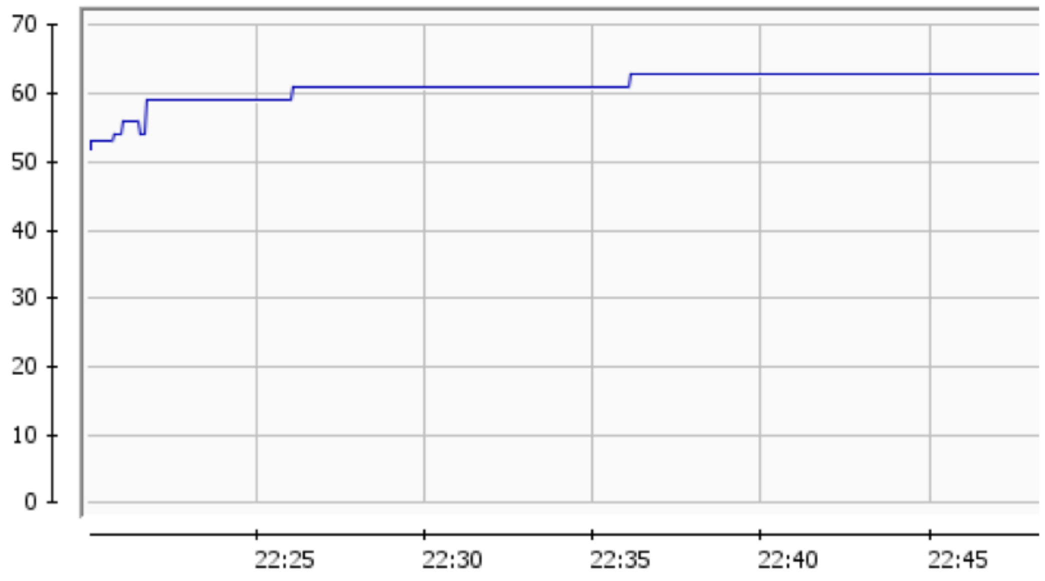

Figure 4.9: Number of threads after NIO refactoring

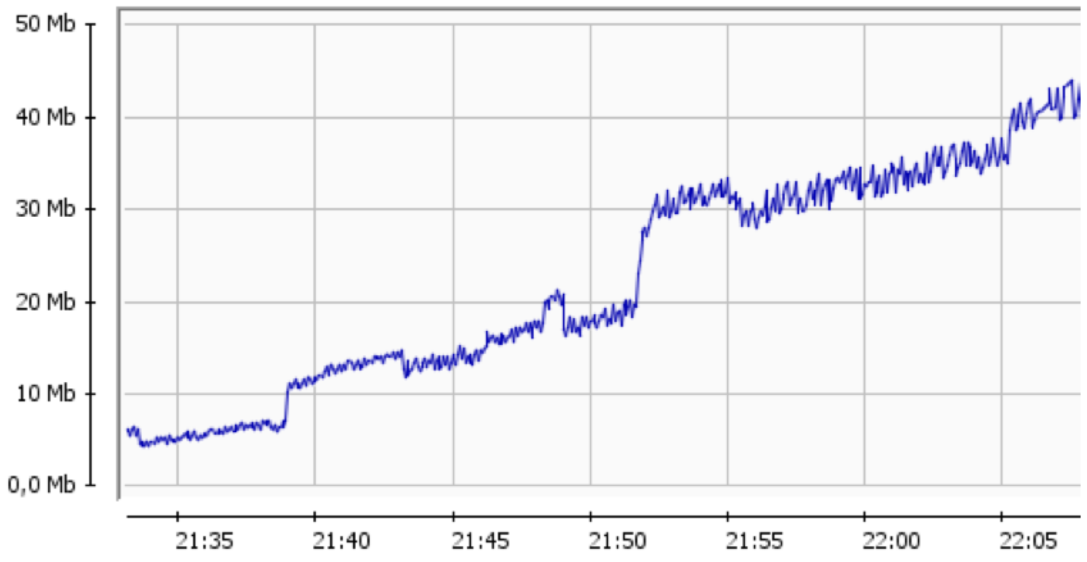

Figure 4.10: Memory usage in the first version of PariTorrent

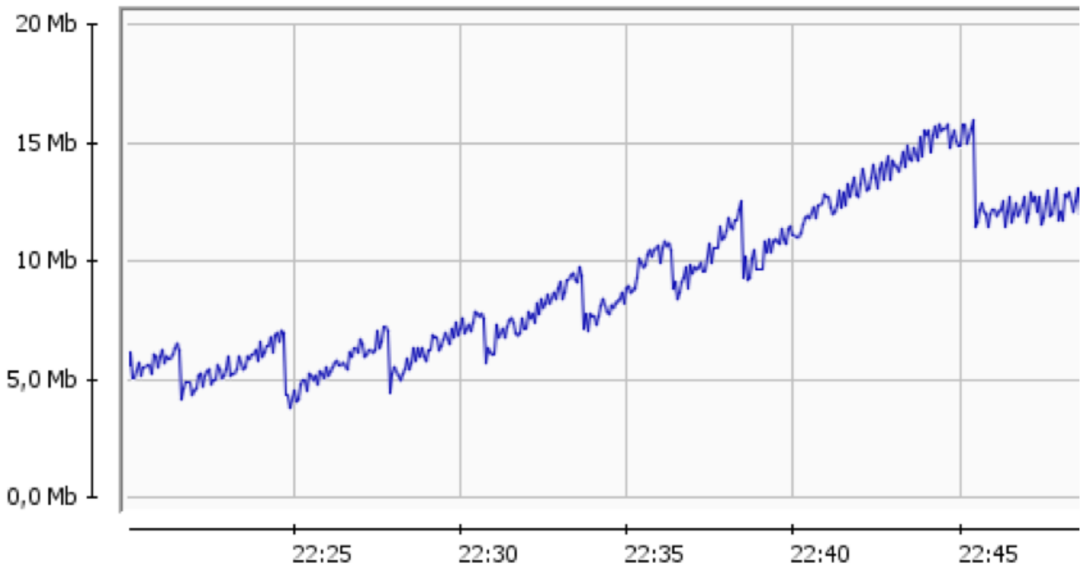

Figure 4.11: Memory usage after NIO refactoring

5 PariTorrent Refactoring

5.1 New working mechanisms

As stated earlier in chapter 2, PariConnectivity manages asynchronous sockets through the use of notifications delivered on Blocking Queues. We have already seen how to request a auto-renewed, non-blocking TCP socket using PluginSender:

```
TCPNonBlockingSocketAPI socket =
ps.requestSingleTCPNonBlockingSocketAPI(new IFeatureValue[] {
new FeatureValue("time", 60000) }, true);
```

but that is not the only thing we need to do in order to have a fully functional socket, the queues need to be set up first:

```
socket.setDefaultQueues(ReadListener.getQueue(),
WriteListener.getQueue(), ConnectListener.getQueue());
```

ReadListener, WriteListener and ConnectListener are three separate threads with one simple job: to read incoming notifications from PariConnectivity. Each one of them has its own queue, defined as:

```
private final static BlockingQueue<AsynchronousNotification
<TCPNonBlockingSocketAPI>> queue = new
LinkedBlockingQueue<AsynchronousNotification
<TCPNonBlockingSocketAPI>>();
```

Algorithm 5.1 {Read, Write, Connect, Accept}Listener's simple go() method

```
public void go() {
    try {
        while (!this.mustStop()) {
            queue.take().process();
        }
        // Thread is quitting
    }
    catch (InterruptedException e) {
        // Thread Interrupted
    }
    catch (Exception e) {
        // Other exception, restart thread
    }
}
```

that is, evidently, a blocking queue of AsynchronousNotifications of type TCPNonBlockingSocketAPI.

When a notification is added to the queue by PariConnectitity it will eventually be taken and processed by one of these threads.

```
queue.take().process()
```

gets the first available notification (possibly suspending itself until one is available), and calls the `process()` method on it. The process method simply calls the

```
public boolean process(
AsynchronousNotification<TCPNonBlockingSocketAPI> parameters)
```

on our implementation of the notification and gets processed. Being able to override this method is key to providing the correct flow of the program running.

It might seem to get a bit overcomplicated, but it's actually quite simple, once you get the gist of how it works. This simple example should further clarify this matter. Suppose we want to continuously read from an already

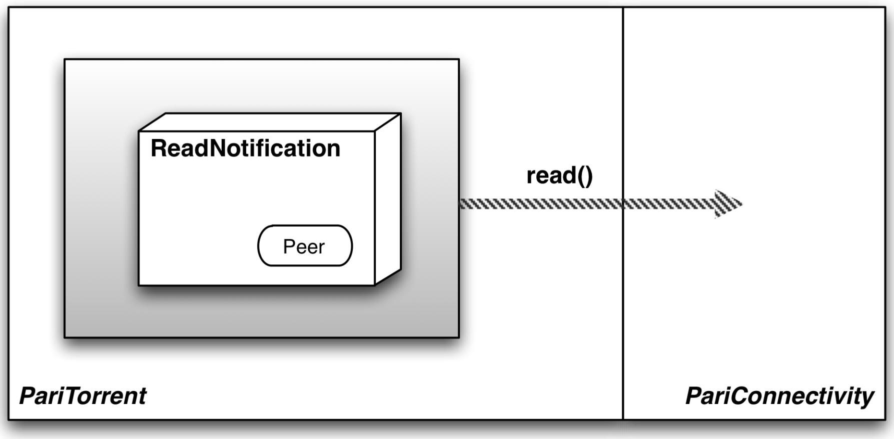

Figure 5.1: Read Request from PariTorrent to PariConnectivity

established socket with a remote peer. We will need a ReadListener thread, as the one described earlier, and a ReadNotification class that implements the `PluginNotification<S extends SocketAPI>` interface. This interface requires only that a `process(AsynchronousNotification<S> availableData)` method is made available, but also allows us to embellish it as much as we would like, by embedding whatever data we need. In our peer to peer environment we would like to be able to keep track of which peer that socket belongs to, and therefore we add to our notification implementation the mandatory `peer` object, which uniquely identifies it. Algorithm 5.2 describes a simplified version of the actual PariTorrent implementation of such a class. Whenever we wish to read from that socket we would simply call:

```
socket.read(new ReadNotification(peer));
```

effectively embedding our notification, along with its additional information, in our read request (see Figure 5.1). PariConnectivity receives the request, tries to read on the socket (potentially postponing until it can be satisfied) and returns an AsynchronousNotification object in the delegated queue (Figure 5.2). That object contains our initial notification, the data it received, and other potentially useful information. The ReadListener extracts the Asynchronous-

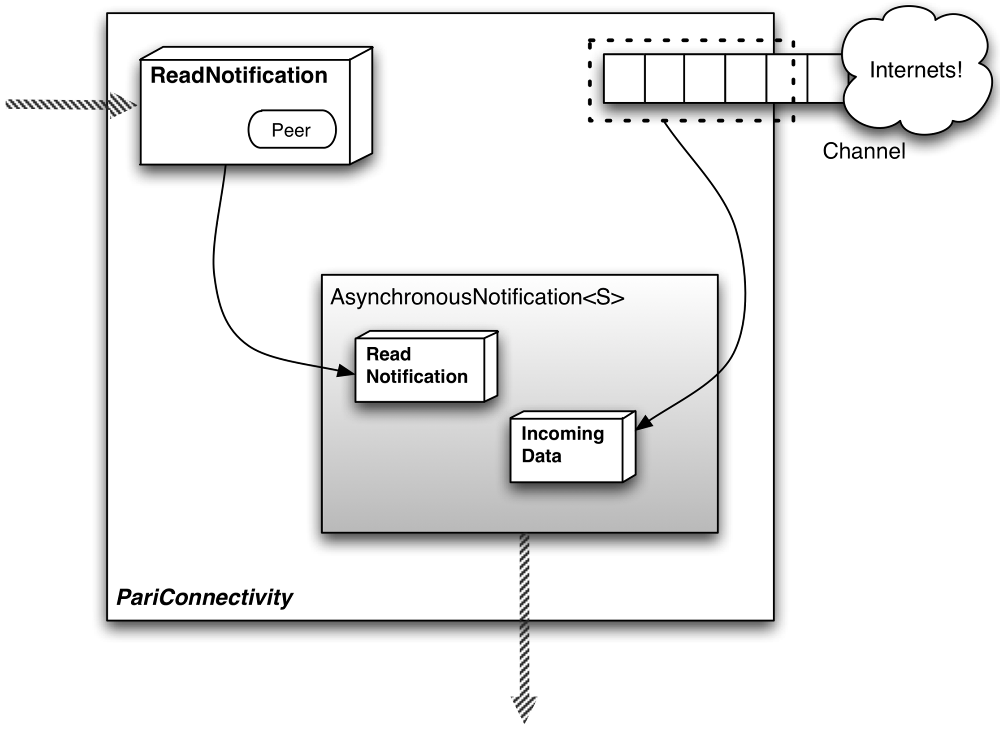

Figure 5.2: PariConnectivity packages the ReadNotification and the incoming data into an AsynchronousNotification

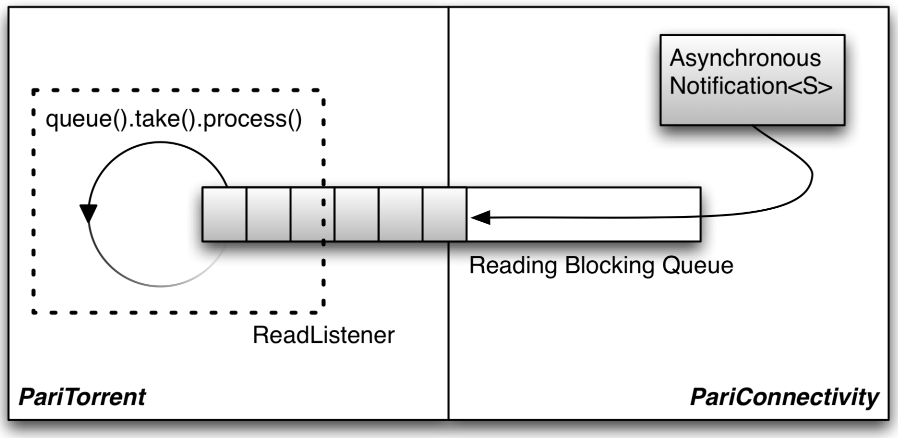

Figure 5.3: PariConnectivity stores the notification in the ReadQueue, PariTorrent's ReadListener takes and processes it

Notification from the queue, and calls its `process()` method (Figure 5.3). The `process()` method calls ReadNotification's (or whatever PluginNotification implementation we embedded) `process(AsynchronousNotification <TCPNonBlockingSocketAPI> parameters)` method, with its owner object as parameter, finally allowing the original request to be satisfied. If we want to continuously read from that socket, we just need to issue another read request in this process method, effectively restarting the procedure.

Read, Write, Accept and Connect notifications are all similarly managed, with core differences residing only in their process methods. This, for instance, is the write method signature for a TCPNonBlockingSocket:

```
public void write(PluginNotification<TCPNonBlockingSocketAPI>
notifyObject, byte[] data);
```

which simply requires a PluginNotification compliant object (which, again, we can customise as much as we want) and the data we want to write to the socket.

5.2 Adapting PariTorrent

As stated in chapter 4, PariTorrent managed peer to peer communications through Java NIO channels, selectors and buffers directly, using three main threads: RemotePeerListener, which managed new inbound connections, TorrentMessageReceiver and TorrentMessageSender, which, respectively, managed inbound and outbound packets through already established sockets.

5.2.1 RemotePeerListener

The RemotePeerListener thread was removed and substituted by the AcceptListener thread and the AcceptNotification class. The AcceptListener is akin to to the mock class presented in the section before. The AcceptNotification class is kept very simple, it doesn't require any other objects to be attached as the only information we need: the incoming socket and the server socket, are already included in the AsynchronousNotification object. The process method is, again, very bare, it issues another accept request on the server socket to keep the system working, creates a peer object for the new incoming connection with incomplete information, finalises the socket, registers it with TorrentMessageSender and issues the first read request on the incoming socket. Notice how we don't have to handle the handshake (plain or encrypted) anymore, because we don't actually read any data. That operation is carried on by the new MessageReceiver thus allowing us to remove redundant code.

5.2.2 TorrentMessageReceiver

In a similar fashion, TorrentMessageReceiver was no longer necessary and was substituted by ReadListener and ReadNotification. ReadListener is again, pretty much the standard listener, while ReadNotification required extensive work. First of all it embeds the peer to which that notification refers to, just like the example presented in the previous section. The process method is where actual work is started though. A stripped down version of the method is available in algorithm 5.2. Its main function is to put the data received from the notification into the peer's buffer, call `analyzeBuffer()` and issue another reading request to the peer's socket (`tryReading()`). AnalyzeBuffer is a modified version of the one that was available before in TorrentMessageReceiver, it now accounts for the new working mechanisms and doesn't need the peer parameter as it is now an instance variable of the notification. The original `readHS()` only had to manage handshake messages that were an answer to connections initiated by us, now it also has to handle incoming connections, possibly answering to an encryption handshake attempt. This was achieved by simply exploiting the preexisting peer status structure and changing it accordingly. Some code was also reused from the previous RemotePeerListener implementation. If the peer is new it is also added to the peer list in the corresponding DownloadTorrent thread. Other packet parsing methods such as `readPP()` and `readAZ()`, were simply imported from the older version. Other cryptography related methods also required only minor tweaking.

Algorithm 5.2 A stripped down version of PariTorrent's implementation of a ReadNotification

```
public class ReadNotification implements PluginNotification<
    TCPNonBlockingSocketAPI> {

     private final IPeer peer;

     public ReadNotification(IPeer peer) {
          // Our attached object
          this.peer = peer;
     }

     public boolean process(AsynchronousNotification<
         TCPNonBlockingSocketAPI> parameters) {
     if (parameters instanceof AsynchronousReadNotification){

          AsynchronousReadNotification<TCPNonBlockingSocketAPI>
              readParameters = ( (AsynchronousReadNotification<
              TCPNonBlockingSocketAPI>)parameters );
          if (readParameters.getException() != null) {
               // Exception!
               this.peer.disconnect();
               return false;
          }
          if (readParameters.endOfStream()) {
               return false;
          }
          byte[] readBuffer;

          //Read from socket
          readBuffer = readParameters.getData();

          //Synchronized access to buffers is highly suggested, as
              they're not inherently thread safe
          synchronized(this.peer.getByteBuffer()){
               //Append data to peer read buffer
               this.peer.getByteBuffer().put(readBuffer);
          }
          analyzeBuffer();
          // Resume reading
          this.peer.tryReading();
          return true;

     }
          //Implicit else
          //Wrong notification type in queue, shouldn't happen,
              resume reading
          this.peer.tryReading();
          return false;

}
```

5.2.3 TorrentMessageSender

TorrentMessageSender was treated differently. Of course we still needed a WriteListener and a WriteNotification implementation, but the bulk of the work is still carried on by a modified TorrentMessageSender. WriteListener is, again, kept very simple. WriteNotification too as it merely enables the peer to write again (a simple lock was implemented to prevent multiple write notifications to be issued to the same socket as PariConnectivity didn't cope very well with them), and wakes the MessageSender thread.

TorrentMessageSender required only minor modifications to work with the new PariConnectivity. It works with peers with already established sockets, either finalised by RemotePeerListener or ConnectNotification and cycles through them sending whatever message is in their outbound queue. It then suspends itself until it is woken up either by a notification from DownloadTorrent or WriteNotification, or by its own timeout, set to manage the need to send a keep-alive message to every peer we didn't send anything to in the last two minutes.

Integrating the whole sending system in WriteNotification was also counted as an option, but was quickly ruled out as it would have needed extensive modification to DownloadTorrent, which is currently worked on by other students in other side projects and would therefore lead to a daunting merging experience. The gain associated with the operation would also have been minimal, resulting to one mere thread less.

5.2.4 Other modified classes

DownloadTorrent

DownloadTorrent is the main thread responsible for the program flow that is followed to download a torrent. There is one DownloadTorrent for each one of the torrents that are currently active. As DownloadTorrent was responsible for the initial creation of the socket for new peers, some minor modifications were needed, only to comply to the new specifications in matters of initiating and finalising a connection.

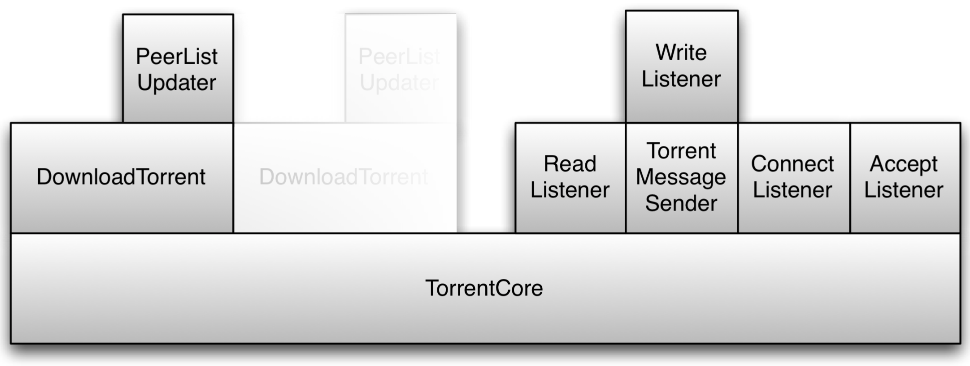

Figure 5.4: Structure after PariConnectivity refactoring (WriteListener is actually created by TorrentCore, but interacts only with TMS)

Peer

As stated earlier Peer, and its interface IPeer, got a simple write locking mechanism, new statuses to manage the new handshake procedure and the tryReading method, which, evidently, sends a read request with the according read notification.

ConnectListener & ConnectNotification

PariConnectivity also requires the implementation of a ConnectNotification queue, which is first created when initiating a connecting with a peer and received back on its own queue when a peer is ready to finalise a connection initiated by us. As you may have guessed, the listener is just like the others presented earlier, while ConnectNotification's process method handles connection finalising, enables writing on the peer, registers it to the global TorrentMessageSender and issues the first read request.

PeerListUpdater

PeerListUpdater is responsible for tracker connections and was updated to use the new URLConnectionAPI, offered by PariConnectivity. Asynchronicity didn't offer any real advantage in such a sparse request-response environment, and thus was deemed not necessary.

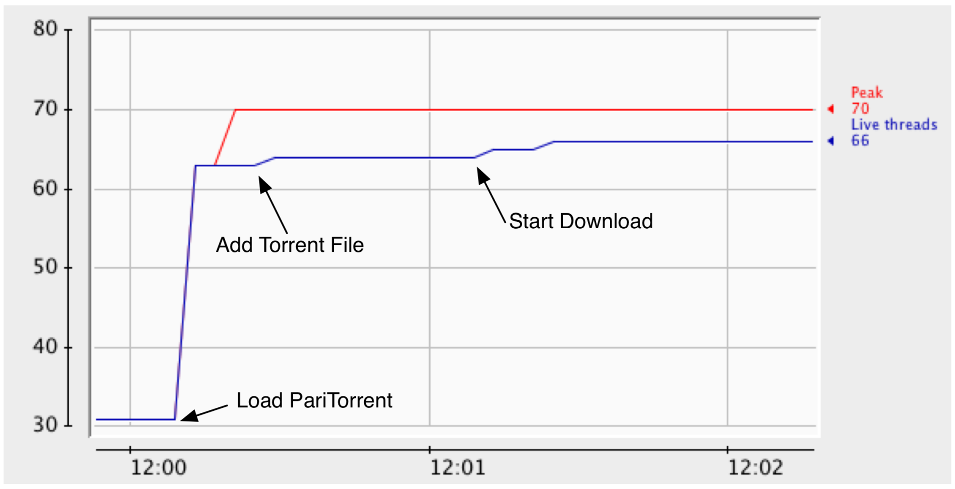

Figure 5.5: Threads Usage in PariTorrent after refactoring

5.3 Results

The number of threads used by PariTorrent needed to raise, in order to satisfy the requirements of the new PariConnectivity. This was nevertheless kept to a minimum. To support PariConnectivity we needed at least four new listener threads: one for each one of the possible channel statuses and consequential notifications (Read, Write, Connect and Accept). Exploiting PariConnectivity's use pattern we managed to remove the TorrentMessageReceiver thread by spreading its functionality between ReadListener and ReadNotification. Similarly RemotePeerListener was spread across AcceptListener and AcceptNotification, thus effectively limiting the number of threads to only two more than the previous version. Those threads are not related to the number of peers we established a communication with, nor to the number of active torrents, their presence does not affect memory or cpu efficiency. Memory footprint was also kept low and is directly comparable with the previous version.

The new structure also allowed us to remove much duplicate code, such as the redundant handshake management in RemotePeerListener, thus granting us cleaner and more maintainable code, which is something that is always much sought after in PariPari as part of the philosophy of Extreme Programming.

Download speed is not directly comparable as it depends on many external factors (basically sheer luck...) and would thus require a large number of tests to assess its actual performance. In the few tests we were able to make download speed was uncompromised by this new implementation and reached the peak speed of 1,13MB/s in a 20Mbit DSL environment. Download speed is consistent and comparable with many of the most popular clients.

6 Conclusions and future work

6.1 Current state of the project

After the adoption of PariConnectivity as the networking interface of the PariTorrent plugin, PariTorrent is both correctly performing and security-manager compliant. Downloading a torrent does not lead to the inevitable crash, as was the case with earlier versions using mismanaged threads, and yet it complies with all of PariPari's security rules, by accessing resources only through its plugins.

Download speed, which is, evidently, a key performance, is adequate and while it doesn't look particularly bad when compared with other clients, there's still room for improvement, especially in the early phases of the download.

Stability is also very important. Crashes still happen and, although they're rare and not always PariTorrent's fault, need to be dealt with.

The User Experience is admittedly still quite poor. There's still no Graphical User Interface, and the current Command Line Interface could use to be a bit more informative and responsive. The lack of a proper GUI and a satisfying UX, makes painstakingly obvious that we're still not ready for external user adoption.

6.2 Future development

What PariTorrent needs now is a thorough testing (both real world and unit testing) phase, to hone down its performance, squash some bugs and clean up a bit of the code base. We're currently implementing some minor features, such as an integrated torrent file search (based on web page scraping) to aid the user in the, sometimes painful, process of torrent discovery.

The most glaring missing feature is the **DHT**. Magnet links are growing ever more popular (*thepiratebay.com* for instance uses only MLs) and without a proper support our client is effectively crippled, both in utility and performance. As a matter of fact the initial download phase, when additional peer discovery is the most useful, greatly benefits from DHT's added peers. The DHT is somewhat complex if compared to other extensions and needs a high level of commitment for the student that is to undertake that project. This has led to the DHT being neglected and postponed many times. We still hope to get this part done in next few months.

PariGUI (PariPari's graphical interface) is finally starting to become useful and will be adopted by PariTorrent in the upcoming months. While its integration doesn't look trivial, the use of an innovative, web based GUI will finally let us overcome the current frustrating UX and give us an edge over competing clients by exhibiting a fresh approach in file-sharing interface design and providing a cohesive environment for all of PariPari's features.

Mi&Ti / **Metacoso** is a side project of both PariTorrent and PariMulo (PariPari's ed2k and kad client) that has the ambitious goal of building an unified client for both networks. While others (e.g. MLDonkey) have a client that supports both, our goal is different. What we're aiming for is a client that is able to download the same file from both networks at the same time, benefiting from higher download rates and a healthier peer database. This has proved to be quite difficult to achieve and has not a deadline set as for now, it will be done when it's done. Such a feature, in a fully working condition, has the potential to completely set PariPari apart from other competing P2P systems.

List of Figures

Bibliography

[1] **Paolo Bertasi**: *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, 2006.

[2] **Paolo Bertasi**: *PariPari: design and implementation of a resilient multi-purpose peer-to-peer network*, 2010.

[3] **Michele Bonazza**: *PariCore*, 2009

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

[5] **Simone Pozzobon**: *PariPari: Modulo Torrent*, 2009.

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

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

[8] **Andrea Gallo**: *PariPari: Crittografia Torrent*, 2009

[9] **Andrea Aldegheri**: *PariTorrent: Performance Refactoring*, 2010

[10] **Francesco Peruch**: *PariPari: Connectivity Optimization*, 2011

[11] **Bram Cohen:** *The BitTorrent Protocol Specification*: http://www. bittorrent.org/

[12] **Petar Maymounkov** and **David Mazières**: *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*: http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf

[13] **Ron Hitchens**: *Java NIO. O'Reilly*, 2002.

[14] *Azureus Messaging Protocol Specification*:
http://wiki.vuze.com/w/Azureus_messaging_protocol

[15] *libtorrent Extension Protocol Specification*:
http://bittorrent.org/beps/bep_0010.html

[16] *Message Stream Encryption Specification*:
http://wiki.vuze.com/w/Message_Stream_Encryption