PARITORRENT: SEEDING STRATEGIES

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Ing. Michele Bonazza

LAUREANDO: Dario Turchetto

Corso di laurea in Ingegneria Informatica

A.A. 2010-2011

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

TESI DI LAUREA

PARITORRENT: SEEDING STRATEGIES

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Michele Bonazza

LAUREANDO: Dario Turchetto

A.A. 2010-2011

To all the people I have met,
since every single one of them
has taught me something.

Contents

Abstract

One of the most widely adopted *peer-to-peer* protocols nowadays is BitTorrent, of which numerous implementations exist. The Torrent plugin developed in the PariPari project is a BitTorrent client written in Java that supports many of the functionalities provided by the most common clients now available freely on the Internet.

In this thesis we will mainly discuss some methods that aim at optimizing the uploading efficiency of the pieces, in order to maximize uplink utilization, therefore reducing the download time for the peers that are participating at the download of the same `.torrent` file, while at the same time penalizing the malicious peers that do not upload pieces to the network (also known as *freeriders*).

Sommario

Uno dei protocolli *peer-to-peer* più diffusi ai nostri giorni è BitTorrent, di cui esistono numerose implementazioni. Il plugin Torrent sviluppato all'interno del progetto PariPari è un client BitTorrent scritto in Java che supporta molte delle funzionalità fornite dai client più comuni disponibili gratuitamente su Internet.

In questa tesi discuteremo principalmente alcuni metodi che mirino ad ottimizzare l'efficienza dell'upload dei pezzi che compongono un `.torrent`, al fine di massimizzare l'utilizzo del canale di upload, riducendo così i tempi di download per i peer che stanno condividendo lo stesso file `.torrent`, penalizzando al contempo i peer malevoli (noti anche come *free-rider*) che non effettuano upload nella rete.

Acknowledgments

Show gratitude. – Randy Pausch

First and foremost, I'd like to thank my family for all the support they gave me during these five years: mom and dad for putting up with my frequent complaints about more or less everything (the food, the weather, the books, the girls and so on) and for always giving me precious advices (such as "No sta 'ndar sui pericoi" and "Always think twice about selling this house, because, look: if you build a wall here and..."); my brother Riccardo, for everything he has taught (and is still teaching) me; my sister-in-law Tracy, for being so kind by informing me about job offers in California and for putting up with my and my family's peculiarities every time she comes to Italy.

I really, really need to thank my friends and coursemates, because I still think I would never have gotten this far without them: Martina, for the cakes, the piadinas, the Skype chats, the many (and I do mean many) emails, the important reminders she constantly gave me (if you have her around you don't need calendars or post-it notes) and for the patience of having read through all of this thesis; Elisa, for her hilarious text-messages and for making even the most boring classes funny; Giacomo, for the extremely appreciated frequent hospitality in his apartment and for the TV series he suggests me (*Blue Mountain State* above all); Lorenzo, for the frequent car lifts, the talks at odd hours of the night, the ping-pong games and the unforgettable party nights.

If I was to remember one by one all the people and situations at the Saint Justin College that cheered up my days I would need at least ten pages, so I can only make a short list and apologize for the people that are not mentioned here. I believe there aren't many other places in which you can find someone ready to jump in a shopping cart before being thrown into a wall made of chairs, or offers to play PC games with you until 4 a.m., or has the idea of writing inappropriate words addressed to the monks with bricks on the grass, or throws fruit inside the windows of military buildings, or... Well, you get the idea. Many thanks to Anthony Pauletto, Francesco "Poker" de Simoi, Alberto Bittolo, Matteo "Lello" Filippi, Stefano Pletti, Marco Vallar, Alberto Gobbato (John McGivern was really a great idea), Guido Morina, Matteo Pachera, Enrico Cappelletti, Alberto Gozzo (the most laid back engineering student I've ever seen), Andrea Sterzi, Roberto Brasola, Luca Lorenzato, Francesco Gava, Francesco "non capisc' più nienda" Testa and Alberto Boschiero, who deserves a special thank for the many talks and beers in the *loggia*, the chess games, the guitar lessons, the various experiments, for greeting me with songs and strange noises every time I got back from the department and for trying to kill me by taking me to the *Buso della Rana* cave. Many thanks also to the former members of the Saint Justin College, who contributed to make the place as it is today and with whom I have spent pleasant hours: Paolo Girardello, Nicola Barbon, Massimo "Bobo" Verona, Piero Veronese, Francesco Cavallin, Tommaso Stecca, Stefano Poletti, Giorgio Quer, Giorgio Ruaro, Giulio Cisamolo, Michelangelo Cao, Enrico Garbuio.

Another bunch of thanks goes to the people in the ACG Laboratory, since I've managed to interrupt their work every once in a while with my silly questions and requests: Marco Bressan, Paolo Bertasi, Michele Bonazza (who gave me precious advices and corrections for this thesis) and Federica Bogo (for the weekly drinks and the funny anecdotes she tells me); thanks to EP too, for the advices he gave me, especially in this last year.

Thanks also to the people in the PariPari project, in particular the ones who didn't kill me even if I kept forgetting things as their team leader: Andrea Aldegheri, Eugenio Valente, Felix Mendoza, Matteo Fincato, Alessandro Dal Corso and Paolo Martin. Special thanks to the tutors of the Software Engineering class: Francesco Peruch, Francesco Mattia, Vincenzo Cappelleri and Mattia Samory, for being so kind every time I bugged them with some question or another.

Life in San Donà is not very exciting compared to the one in Padova, but if I had some funny days even there is mainly thanks to Roberta, for the drinks and the walks and the long talks, and Carlo and his company, for all the barbecues and the nights out that almost always start or end with someone taking out a bottle of red wine.

Finally, I want to thank my teddy bears Tizio, Caio, Sempronio and Spelo, with whom I have spent many joyful hours as a child.

1 PariPari

In this Chapter we will describe the PariPari project: what it is, how it's structured, the plugins that are part of it and the basic messages that are exchanged between plugins in order to make it fully functional in a modular fashion.

1.1 PariPari and its structure

PariPari is a software engineering project currently in development at the Department of Information Engineering of the University of Padua. It is being developed mainly by students belonging to the Information Engineering and Computer Engineering courses; at the present time there are about 40 of them involved in the project. Every student is assigned to a specific group working on a plugin – we will see briefly that PariPari is composed in a modular fashion that provides its own practical advantages.

PariPari is a multi-functional platform based on decentralized services, such as peer-to-peer file sharing, VoIP[1], instant messaging, distributed storage of data and so on. It is entirely written in the Java programming language, since it allows PariPari to be instantly available for all the major operating systems (thanks to the portability of the Java environment).

PariPari is completely *serverless* and *decentralized*, that means that there aren't privileged nodes. This makes the network robust against DoS[2] attacks, whose aim is precisely to block the service offered by a central node, thus impairing all the nodes relying on it.

PariPari is developed with a modular structure in mind: in this way it will be easier for future plugins to be developed and benefit from the services offered

[1] Voice over IP

[2] Denial of Service

by the already existing plugins. Also, in this way a user can get a customized PariPari clients on its machine, since he doesn't need to load all the available modules but just the ones he likes to have. Loading a plugin is very easy: writing `add plugin_name` through the available console is all that is needed to add the plugin called `plugin_name`.

For structural and functional reasons, plugins can be divided into two categories: inner- and outer-circle plugins. All the plugins need to interface themselves with the **Core** module, which is the one that manages all the communication between plugins.

1.2 Inner-circle plugins

These plugins are the ones that mainly offer services and resources (such as TCP and UDP sockets and disk storage space) to the other plugins. The following is a list of the inner-circle plugins:

- **Connectivity** This plugin offers and manages connectivity services, in particular it provides a wide range of socket APIs (TCP[3], UDP[4], blocking, non-blocking) that are necessary for the outer-circle plugins in order to work properly. Its latest version, developed using Java NIO[5], provides non-blocking calls based on *channels* and *selectors*.

- **Credits** The Credits system is embedded in the Core and is the one that manages the *credits*, the currency used in PariPari. We will describe shortly how it works and why it is important in a separate Section.

- **Local Storage** This is another fundamental plugin, since it is the one that lets every other plugin write and read data to and from mass storage (for example the hard disk).

- **DHT** This is the plugin that implements a *Distributed Hash Table*, with the usual node and resource lookup based on the XOR metrics (as in Kademlia). We will briefly describe how it works in a separate Section, since it is fundamentally the way it works in BitTorrent.

[3]Transmission Control Protocol.
[4]User Datagram Protocol.
[5]New Input/Output.

Figure 1.1: The PariPari logo.

1.2.1 Core

The latest version of the Core is also called **T.A.L.P.A.**, which stands for *The Ancronym for Lightweight Plug-in Architecture*. It manages the various plugin, as we already said, providing a common set of interfaces with which programmers must comply in order to make their plugin work (this set of interfaces is in package `paripari.API`) and including the Credits feature, which we will very soon talk about in more detail.

More information about PariCore can be found in [6].

1.2.2 Connectivity NIO

The Connectivity plugin has been recently refactored using *Java NIO*, which is a collection of APIs that offer features for intensive I/O operations. One of the main advantages of NIO is the availability of objects such as *buffers*, *channels* and *selectors*, which provide an effective communication system that in some cases does not need to use the CPU to transfer data and, most importantly, lets the programmer design his code in a non-blocking fashion, which is highly desirable when dealing with file-sharing systems, for example.

For more information about Connectivity NIO see [4].

1.2.3 Credits

As we said, the Credits system is the one that manages the PariPari currency called *credits*. Inspired by real world economies, every plugin that needs *resources* has to spend some credits to buy them: if it has the necessary credits, it will spend

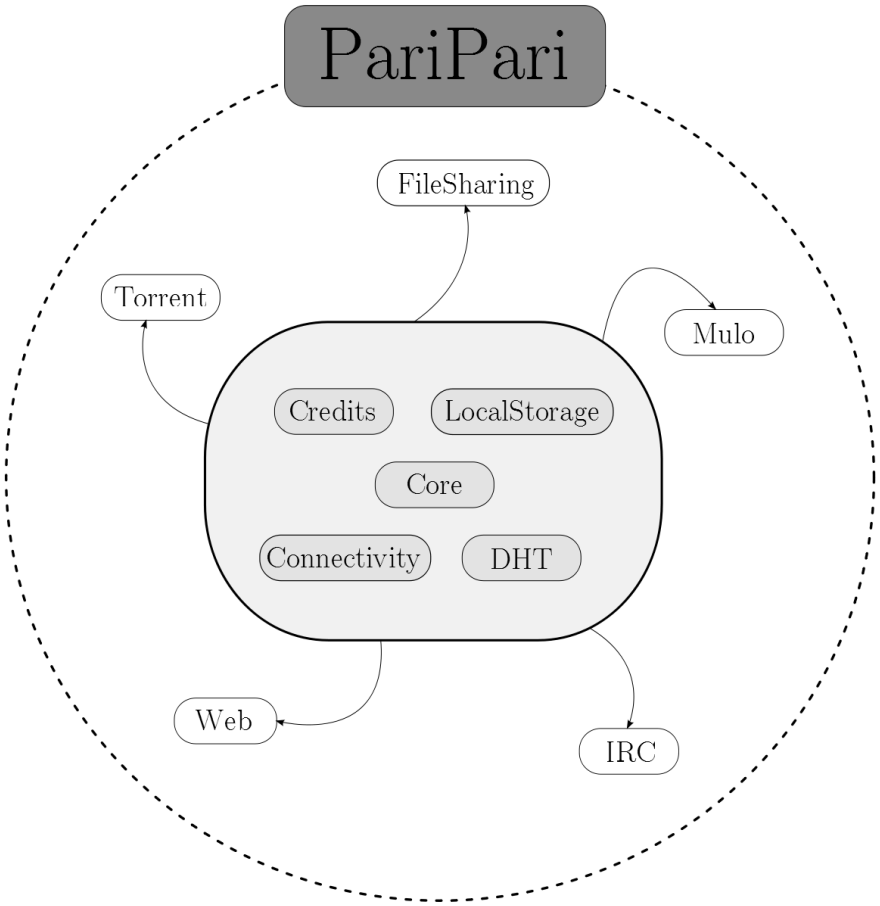

Figure 1.2: Representation of the plugins divided into the inner and outer circles.

them and they will be subtracted by its total amount of credits, otherwise it will not be able to get them. Also, credits can be used between mutiple PariPari nodes.

We can then divide the credits in two categories:

- **Intra-peer credits** used between the plugins that constitute a single node, to buy and sell resources
- **Inter-peer credits** used between different PariPari nodes

Why do we need a credits system in PariPari? Because otherwise we would not be able to provide resources to plugins in a proportional manner, granting for example lots of sockets to a plugin that is not using them effectively while others are starving for bandwidth that would otherwise let them download huge amounts of data. Credits aim at solving this unfairness problem.

1.2.4 DHT

PariDHT is the plugin that implements a DHT based on an address space of 256 bit (in order to minimize collisions when choosing the key of a resource). The metric employed is the usual XOR metric, defined as $d(x, y) = x \oplus y$, which calculates the distance between two nodes as the binary XOR between their 256 bits identifiers.

Looking for a node in the DHT (the operation also known as *node lookup*) requires at most $O(\log N)$ steps, where N is the number of active nodes in the network.

1.3 Outer-circle plugins

Next up are the outer-circe plugins; we list here a few of them:

- **Torrent** We will describe in fairly good detail this plugin in Chapter 3; here we just say that it is a client for the BitTorrent protocol.
- **Mulo** This plugin is a client for the eMule protocol and network. It is a fully functional client that supports many of the most important features provided by the most common clients available nowadays, including some features if its own. More information about this plugin and some of its features (including Kad support) can be found in [13], [14] and [15].

- **IM**[6] and **IRC**[7] These plugins are both clients for instant messaging (also colloquially known as *chat*). They are based on the *Jabber* protocol, now known as *XMPP*[8], which is a series of open protocols and technologies specifically designed for instant messaging.

- **Web** This plugin is a distributed web server; this means that pages are not saved on a single machine but they are distributed over the network and requests are served by the appropriate node.

- **GUI** This very important plugin provides the *Graphical User Interface* that lets the average user enjoy PariPari without the pain caused by the command line (which is very far from what modern-day users are accustomed to). This plugin is still in its experimental phase, but much is expected from it.

[6] Instant Messaging
[7] Internet Relay Chat
[8] Extensible Messaging and Presence Protocol, `http://xmpp.org/`

2 The BitTorrent protocol

In this Chapter we will present the BitTorrent protocol, in order to give an overview of the essential concepts that will let the reader understand what we have done to improve it.

2.1 Introduction to the protocol

BitTorrent is a peer-to-peer file sharing protocol used for distributing large amounts of data over the Internet. It was designed by Bram Cohen in April 2001 and in July 2001 a first version of the client, programmed in Python by Bram Cohen himself, was released free-of-charge. BitTorrent quickly grew to be one the most widely adopted protocols for transferring files over the Internet, even becoming a company of its own (registered as BitTorrent, Inc.); current estimates[1] claim that at least 40% of the internet traffic (depending on geographical location) can be attributed to BitTorrent.

Lots and lots of BitTorrent clients now exist[2], written in a number of programming languages; the very first BitTorrent client, developed by Bram Cohen, almost does not circulate anymore, replaced by the very efficient and lightweight *uTorrent*, since the BitTorrent company acquired the uTorrent source code.[3].

Over the years, many other clients appeared and some important extensions to the original protocol were designed and developed. Let's see how the original protocol works, since it is still the basis of the actual file distribution.

[1] `http://torrentfreak.com/bittorrent-still-dominates-global-internet-traffic-101026/`
[2] `http://en.wikipedia.org/wiki/Comparison_of_BitTorrent_clients`
[3] `http://torrentfreak.com/bittorrent-inc-buys-utorrent/`

Figure 2.1: Official BitTorrent logo.

2.2 Trackers and torrents

In order to begin downloading a file (or a group of files) distributed with BitTorrent, a client needs to acquire the `.torrent` file for that download. This can be done in several ways, and we will discuss later some of the most recent advancements regarding this step.

The `.torrent` file is composed of:

- *a list of trackers*: these are some URLs to contact in order to get the initial list of peers from which the client will begin downloading
- *the pieces' hashes*: the data to be distributed is divided into many pieces (the size of which usually depends on the size of the whole data) and for each piece its SHA-1 hash is computed and written on the `.torrent` file for future error-checking

Figure 2.2: Example of hash computation for every piece composing a torrent.

A `.torrent` file is not written in clear text; instead, it is coded with a particular encoding called *BEncode*, which we described in Appendix A.

The client then begins parsing the `.torrent` file and contacting the trackers (through HTTP or UDP) in the order in which they are listed. But what are these trackers?

A tracker is a server machine with the appropriate software that lets it manage the peers associated with a given `.torrent` file. When a tracker receives a request from a peer that is trying to participate in the download of a `.torrent`, it

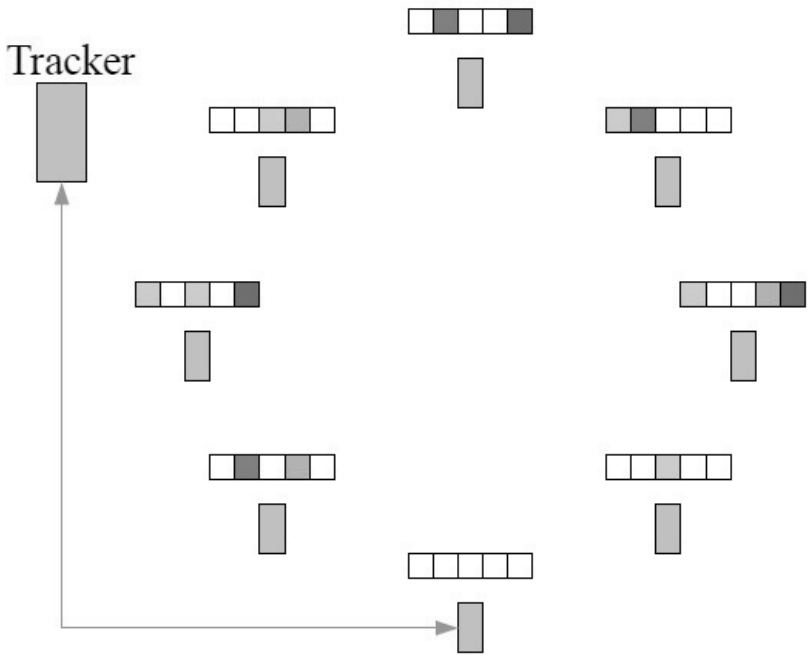

Figure 2.3: A client sends its first request to the tracker.

sends the peer a list of at most 50 randomly chosen peers which are downloading or uploading the same `.torrent` file. The peer can then begin contacting the received peers in order to start the download.

When a peer has finished downloading a file it sends a message to the tracker stating that it has completed the download; the tracker will then register it to the list of peer that have the complete data associated with a `.torrent` file (these peers are also called `seeders`).

Let us now have a look at how a peer communicates with other peers.

2.3 Peer-wire protocol

The set of rules and messages which lets the peers communicate with each other is called the *peer-wire protocol*. It consists of 11 messages (the structure of these messages can be found in table 2.1):

- **handshake** It's the first message sent to a peer that has been found available for connection

- **keep-alive** Sent generally every two minutes if no other messages have been exchanged, in order to keep the connection open

message	length	id	payload
keep-alive	0000	-	-
choke	0001	0	-
unchoke	0001	1	-
interested	0001	2	-
not interested	0001	3	-
have	0005	4	<piece index>
bitfield	0001+X	5	<bitfield>
request	0013	6	<index><begin><length>
piece	0009+X	7	<index><begin><block>
cancel	00013	8	<index><begin><length>

Table 2.1: The typical messages forming the peer-wire protocol

- **choke** Sent to a peer to inform it that we are going to stop uploading to it
- **unchoke** Sent to a peer to inform it that we are going to begin uploading to it
- **interested** Sent to a peer that has pieces we need
- **not interested** Sent to a peer to inform it that is doesn't have pieces we need
- **have** It contains the number of a piece we possess
- **bitfield** Its payload is an array of bits: every element is set to 0 if we don't have the corresponding piece, otherwise it is set to 1. This message must be sent immediately after the handshake phase
- **request** Request for a given piece
- **piece** Its payload is the chunk of a piece we requested
- **cancel** Sent to a peer if we no longer need a piece we previously requested from it

The typical download cycle is shown in figure 2.4.

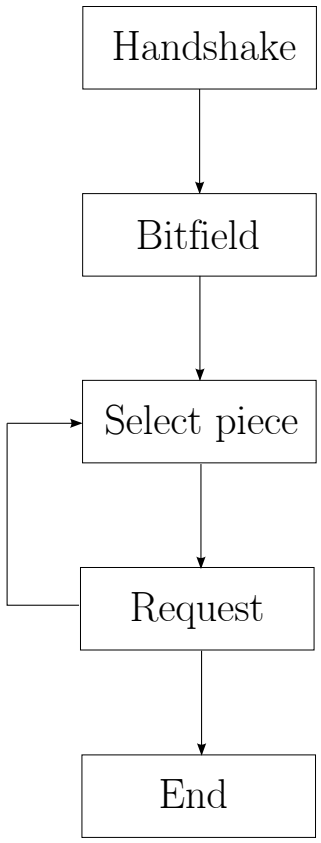

Figure 2.4: Torrent download workflow.

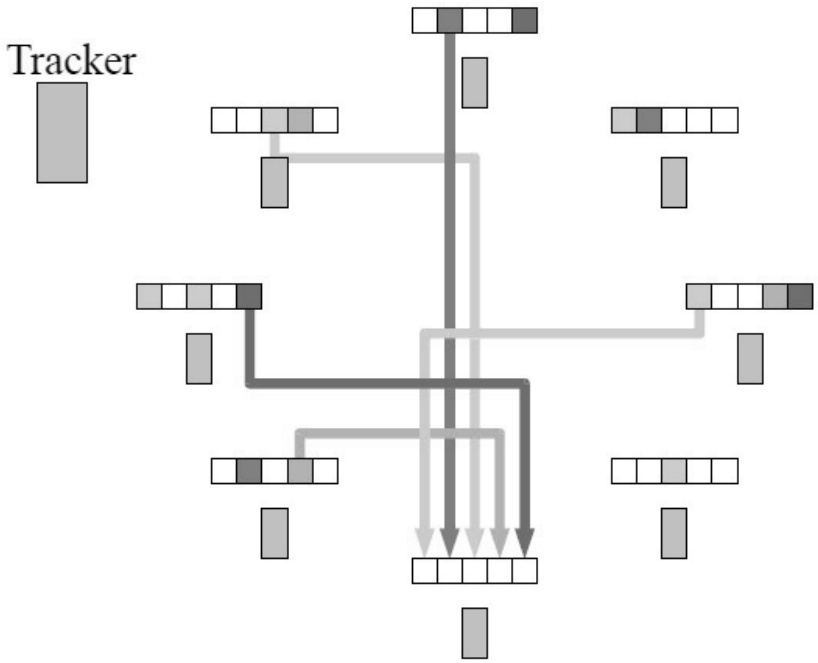

Figure 2.5: After a peer has received other peers' IP addresses by the tracker, it begins contacting them.

2.4 Choking and unchoking

We have seen that a peer A can begin downloading from another peer B only if B has *unchoked* A. By *unchoking* we mean that B has sent an `Unchoke` message to A, stating that A can begin asking pieces to B. How does B choose A over one other arbitrary peer, say C? And how many peers can B unchoke at the same time?

The traditional unchoking algorithm implemented in various BitTorrent clients allocates `5 slots` for unchoking peers: 5 of these slots are used by what we will call the *rational unchoking algorithm*, while the remaining slot is reserved for the so-called *optimistic unchoking*.

2.4.1 Rational unchoking

As we will see in the following Chapters, numerous variants exist to the standard unchoking algorithm; here we will discuss its main and most popular implementation.

In rational unchoking, a peer A first sorts the peers it is connected to in a way that's convenient for itself; there are many ways to do so, for example the peers could be sorted based on their upload rates or the interesting pieces they have.

After A has finished sorting its known peers, it sends `Unchoke` messages to the first 4, in order to let them upload from it. This operation is repeated every 10 seconds; it is of course possibile that, in the following round, one or more of the 4 previously unchoken peers gets unchoken again (or, to be more precise, it *stays* unchoken).

2.4.2 Optimistic unchoking

This is a particular kind of unchoking method described by the official protocol that, although at first may seem counter-intuitive, is all to the advantage of the peers that send the `Unchoke` message.

Consider this scenario: a peer A knows peers B, C, D, E and F. A has unchoken B, C, D and E and it is fairly pleased with them: they upload to him interesting pieces at a decent rate, so why bother to even take F into account? Well, the wise reader may think, what if F has both interesting pieces and an extraordinarily high upload rate? It would be stupid for A not to make profit of such a good peer.

This is exactly what optimistic unchoking was designed for: to widen the view

of a peer in order to get a taste, here and there, of possibily interesting peers – even *more* interesting than the peers it is already connected to.

The optimistic unchoking algorithm is usually called once every 30 seconds (instead of 10, as in the rational unchoking).

2.5 Extensions to the main protocol

Now that we have seen how the basic BitTorrent protocol works, we will discuss some of the most important extensions that have been designed in the last decade in order to make it more efficient and secure.

2.5.1 Protocol encryption

Since BitTorrent occupies a large share of the Internet's traffic and it is frequently used for the illegal distribution of copyrighted material, some ISPs[4] began throttling BitTorrent's traffic. The main and most effective counter-measure adopted by some developers was the creation of an encryption system that would mask the content of the packets sent by the various clients, making it look as generic TCP traffic.

Various versions of the protocol have been designed, but the one adopted nowadays[5] lets a client encode the header of the packets (which suffices as a bandwidth-throttling counter-measure), the payload or both. In order to achieve complete randomness from the first byte on, the protocol uses a *Diffie-Hellman* key exchange.

More information about the encryption algorithm can be found in [8].

2.5.2 Extension Protocol

The Extension Protocol[6] is a layer that aims at simplifying the introduction of subsequent extensions. It operates by activating one the reserved bytes in the handshake message, so that the peers exchanging the handshake can see whether a peer supports a given extension or not.

The vast majority of modern clients support this extension, in particular the very useful **Peer Exchange** (also known as PEX), which lets two peers exchange two lists of peers each: one of these lists contains the IP addresses of good peers,

[4]Internet Service Providers

[5]`http://wiki.vuze.com/w/Message_Stream_Encryption`

[6]`http://www.bittorrent.org/beps/bep_0010.html`

which are available for contact and are uploading good data, while the other list contains the IP addresses of bad peers, which are unreachable or upload data that does not pass the hash check. Each of these lists can contain a maximum of 50 peers.

More information on the extension protocol and the Peer Exchange can be found in [11].

2.5.3 Fast Extension

Fast Extension[7] is an official extension to the BitTorrent protocol. It consists of an extended set of messages that aims at making the protocol more efficient and, as the name says, fast, in order to achieve faster downloads.

The messages forming this extension are:

- `Have all`
- `Have none`
- `Suggest piece`
- `Reject request`
- `Allowed fast`

The most interesting of these messages is the `Allowed fast`, which contains the index of a piece of a given torrent and is sent to a piece when we want to tell him: *you can ask me this piece and I will send it to you even if you are choked*. The purpose of this message is to shorten the duration of the start-up phase of a peer which has no pieces and therefore can't participate in the traditional tit-for-tat for receiving pieces it needs.

More information about Fast Extension can be found in [11].

2.6 Distributed Hash Table

The proposed[8] Distributed Hash Table (DHT) for BitTorrent works as a standard Kademlia-based DHT, with a 160-bit addressing space. When a peer bootstraps

[7] `http://www.bittorrent.org/beps/bep_0006.html`
[8] `http://www.bittorrent.org/beps/bep_0005.html`

into the DHT, it can start looking for peers which are downloading the same torrent, connect to them and then exchange the traditional peer-wire messages. The lookup for peers and resources is based on the XOR metric defined for Kademlia.

The DHT extension is now widely adopted by the majority of clients since it is the foundation of the so-called `trackerless torrents`, which are becoming the most effective counter-measure to the various lawsuits that have led to the shutdown of major public trackers (such as *The Pirate Bay*).

3 The Torrent plugin

In this Chapter we will talk about the plugin: what it is and what it does, its structure and organization. We will also see the typical workflow for the download of a torrent, in order to understand the basic details that enable it to participate in a download in the BitTorrent network.

3.1 Introduction to Torrent

The Torrent plugin is the PariPari module responsible for the management of `.torrent` files. It is a BitTorrent client that complies with all the core features and rules of the BitTorrent protocol that we described in Chapter 2, but it also implements some of the most widely used extensions to the protocol itself, such as the Extension Protocol, the Fast Extension, protocol encryption and the Azureus Messaging Protocol.

Some important features are now being actively developed, such as the metadata transfer and the BitTorrent DHT. These are of fundamental importance for the trackerless torrents we talked about earlier; through the DHT a peer can find the resources it needs, and in particular through the metadata transfer it can obtain the `.torrent` file it needs to initiate the data download.

3.2 Plugin structure

The Torrent plugin is composed of a series of packets containing the various classes and interfaces that make the communication with other BitTorrent peers (and consequently the download) possible. In table 3.1 we have listed the main packages that constitute the Torrent plugin and their contents, in order to give

the reader a rough idea of how the plugin is structured.

The main package, called `torrent`, contains 5 classes:

- TorrentConsole
- TorrentCore
- TorrentID
- TorrentListener
- TorrentLogger

The most important of these 5 classes is `TorrentCore`, whose job is to instantiate all the necessary classes for a download, whereas the `TorrentConsole` interprets the input given by the user through the PariPari terminal.

The basic workflow is the following: the user starts PariPari, then adds the Torrent plugin through the command `add torrent`, which loads the Torrent JAR file and starts `TorrentCore`. At this point, the user can `reload` pre-existing downloads or `add` new torrents (by having them saved on disk or directly by URL, through the `addUrl` command). Once a torrent file has been added (and the associated instance of `DownloadTorrent` has been created), the user can `start` the download, which creates the `TorrentMessageSender` and `TorrentMessageReceiver` instances which, together with the `HTTPConnection` instance, begin contacting the tracker and the peers that are downloading/uploading the same torrent.

It is worth noting that `TorrentMessageSender` and `TorrentMessageReceiver` are written using Java NIO[1], the library provided by Sun Microsystems that, among other things, provide methods for asynchronous I/O. More information about the introduction of Java NIO for Torrent can be found in [9].

3.2.1 Configuration file

The XML configuration file for Torrent is a handy tool when we want to tune some parameter without modifying the source code directly. We just need to open the XML file (called `TorrentConfig.XML`), change the settings, save the file and restart PariPari. The changes will be immediately visible, without having to modify the source code and recompile it.

[1] `http://en.wikipedia.org/wiki/New_I/O`

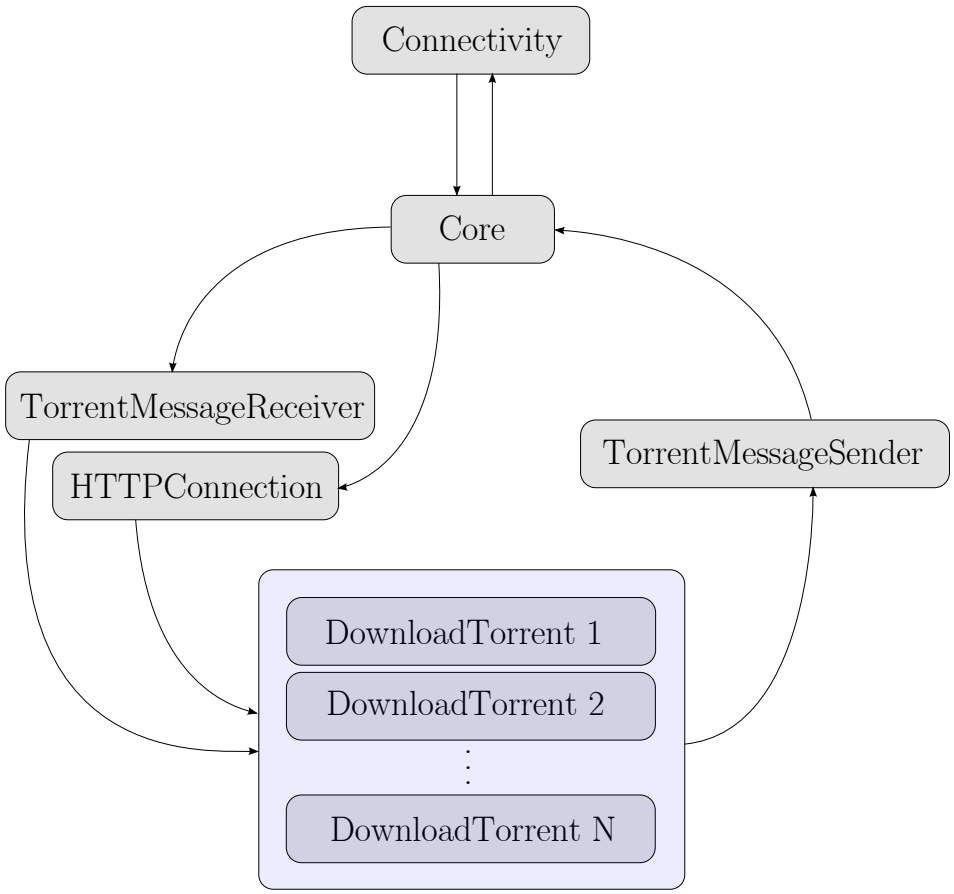

Figure 3.1: Interaction of the main Torrent threads for a download with Core and Connectivity.

name	contents
config	The classes that read from and write to an XML file the configurations for the plugin (fundamental variables such as the port number, the maximum number of concurrent downloads, etc.)
crypto	The classes that define the basics for the protocol encryption as defined by the standars we talked about in the previous chapter.
file	Class `TorrentFile` defines the structure and the contents of a `.torrent` file, while class `FileManager` deals with reading and writing data to and from the selected storage device.
ltep	This package contains the necessary classes for the management of the Peer Exchange mechanism defined by uTorrent and the libTorrent extensions (not Azureus/Vuze).
manager	The most important class here is `DownloadTorrent`, which manages the communication with all the connected peers and calls, whenever appropriate, public methods from other classes (such as saving of a received piece to disk or updating the peer list).
messages	This package contains the two main threads for the exchange of messages to and from the other peers. In particular, `TorrentMessageSender` deals with sending the messages and `TorrentMessageReceiver` deals with receiving the messages, while `TorrentMessageParser` parses (as the name says) the received messages, separating header and ID from the more interesting payload (if present).
peer.messages	Here are all the messages defined by the peer-wire protocol and the classes that generate them, `MessageFactory` in particular.
tracker	Class `HTTPConnection` contains all the methods necessary to establish an HTTP connection with the tracker, sending requests to it and receiving the responses. On the other hand, since the trackers' responses are lists of peers, class `PeerListUpdater` contains the methods that parse the received peer-list and store them in the peer database for future contact.

Table 3.1: The main Torrent packages and their contents.

The configuration file is very useful also when we want to change some default settings without having to type something into the console every time we start PariPari. It will also be useful for the user when he will be able to modify it through the GUI, changing settings at runtime.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE properties (View Source for full doctype...)>
- <properties version="1.0">
    <comment>PariPari Torrent configuration</comment>
    <entry key="tracker_interval_between_contact">150</entry>
    <entry key="completed_files_directory">Completed</entry>
    <entry key="LibTorrent_Extension_Protocol_support">true</entry>
    <entry key="max_connections_for_new_downloads">60</entry>
    <entry key="max_simultaneous_downloaders">5</entry>
    <entry key="tracker_port">7881</entry>
    <entry key="downloads_limit_enabled">true</entry>
    <entry key="connections_limit">100</entry>
    <entry key="outgoing_protocol_encryption">Disabled</entry>
    <entry key="bandwidth_for_tracker_connection">5000</entry>
    <entry key="incoming_files_directory">Incoming</entry>
    <entry key="min_port">7881</entry>
    <entry key="listening_port_for_remote_connections">51413</entry>
    <entry key="downloads_limit">3</entry>
    <entry key="experimental_unchoke">true</entry>
    <entry key="tracker_min_interval_between_contact">60</entry>
    <entry key="allow_incoming_legacy_BT_connections">true</entry>
    <entry key="time_between_unchoke">10000</entry>
    <entry key="seeds_limit">3</entry>
    <entry key="bandwidth_for_peer_connection">10000</entry>
    <entry key="Azureus_Messaging_Protocol_support">false</entry>
    <entry key="automatically_remove_completed_downloads">false</entry>
    <entry key="max_port">7889</entry>
    <entry key="seeds_limit_enabled">true</entry>
    <entry key="unfairness_threshold">3</entry>
    <entry key="seeding_quota">2.0</entry>
</properties>
```

Figure 3.2: The XML configuration file for Torrent at the present state of development.

3.3 Communication with inner-circle plugins

Torrent, being an outer-circle plugin, must obtain resources from the inner-circle plugins. In particular, it needs storage methods provided by LocalStorage, sockets provided by Connectivity and credits provided by Credits (even though this is still an experimental plugin). It does so by forwarding requests not directly to the Core but through `PluginSender`, a plugin which was purposely created for making the request procedures easier. Thus, PluginSender provides a wrapper for the Core methods needed to ask resources to the other plugins (every request must pass through the Core), sometimes being very handy because it automatically

manages things such as the Credits requests' renewal.

That's basically everything Torrent needs from the inner-circle plugins; some peculiar cases consist in saving to mass storage torrents with multiple files: in this case the requests to LocalStorage have to be properly handled every time Torrent needs to read from or write to disk in order to avoid getting exceptions on console.

3.4 Supported features

In this Section we will list all the major features supported by Torrent. We already talked in Chapter 2 of some of these, so we will briefly review them and describe how they are implemented in Torrent.

3.4.1 Multitracker support

This is a minor extension but it's very widespread and useful, since in the very early times of BitTorrent a `.torrent` file could contain a single tracker. Nowadays it is very common to insert more than a tracker in the `.torrent` file, so that if the first tracker doesn't work, the client can pass on to the following, try contacting it, and so on.

3.4.2 Extension protocol and Peer Exchange

Torrent supports the extension protocol and the peer exchange that is so important in contacting a large number of peers in order to get high download rates.

The classes that define the extension protocol messages can be found in the package `torrent.peer.messages.ltep`, while the classes that manage the Peer Exchange can be found in `torrent.ltep`, which contains the two classes `UtPeerExchangeManager` and `UtPeerExchangeSender` whose job is to keep track of the peers with which we are enabling the Peer Exchange and to divide the peers we get into the two lists of *added* and *dropped* peers (in fact, there are four lists: two for IPv4 and two for IPv6).

The class `UtPeerExchangeSender` is a class that associates itself with an instance of `DownloadTorrent` and sends a PEX message to all the due peers every 60 seconds (the standard time interval between two consecutive PEX messages).

3.4.3 Protocol Encryption

As previously stated said, protocol encryption is necessary to hide the payload or the header of the packets (or both) in order to avoid getting throttled by the ISPs that check the traffic travelling on their networks.

Protocol encryption can be either *disabled*, *enabled* or *forced*: when we enable the encryption protocol it means that we support it and we can initiate a crypted handshake if the other peers wants to do so; when we are forcing the encryption it means that we will *only* accept encrypted connections (outbound and inbound).

The main classes that manage the protocol encryption can be found in the package `torrent.crypto`.

3.4.4 Azureus Messaging Protocol

The developers of the Azureus/Vuze BitTorrent client have developed a particular set of messages which works only between two Azureus clients. In this way, whenever we receive a Azureus handshake message and we are able to parse it, we can begin communicating with that set of messages, and that only.

The classes that define the Azureus messages can be found in `torrent.peer.messages.azmp`.

Since the two extensions of the Azureus Messaging Protocol (AZMP) and the libTorrent Extension Protocol (LTEP) are mutually exclusive, we need to have a method for choosing between them. This is why the *Extension Negotiation Protocol* was developed.

3.4.5 Extension Negotiation Protocol

This simple extension lets a client choose between the two extension protocol of AZMP and LTEP. It works using two of the reserved bits of the handshake (bits 47 and 48, precisely), which can of course each be set to 0 or to 1. Depending on the combination of these two bits (00, 01, 10, 11) the client acts accordingly, enabling one extension or the other. In Torrent this all happens when we receive a handshake message, inside class `DownloadTorrent`.

3.4.6 File Preview

This is another simple feature that is implemented in Torrent: how many times were we downloading something fairly large (on the order of the hundreds of MegaBytes) that turned out not to be what we were expecting?

00	Force LTEP
01	Prefer LTEP
10	Prefer AZMP
11	Force AZMP

Table 3.2: The conventional table for the Extension Negotiation Protocol.

With file preview it is possible to check directly from within PariPari wether the file we are downloading interests us or not, since what the file preview does is to recognize the most common file types (audio or video, for example) and open them with the appropriate file player/viewer.

3.4.7 DHT

The Distributed Hash Table for Torrent is currently in development but it's not very far from being completed. It is a very important extension for Torrent since nowadays more and more torrents are becoming trackerless, and without a DHT it is impossible to download them, since we don't even have the `.torrent` file sometimes.

It's a very large and consistent piece of code that is being developed on a separate branch with its own threads; the very next step after completion is going to be the integration of the DHT in the present Torrent source code and the coordination of its threads with the standard downloads through `DownloadTorrent`.

3.4.8 Metadata Transfer

Metadata Transfer is a natural extension to the DHT, since otherwise we would not be able to retrieve the list of pieces' hash values that we need to possess in order to check if we are downloading good data or not. Metadata Transfer then mediates the transfer between two peers connected to the DHT of the *.torrent* file, without trackers. Metadata Transfer relies on *magnet URI*, which are particular links that let the client bootstrap into the DHT and begin retrieving the metadata.

All of this is almost completed in Torrent, the only thing missing is the DHT to which the Metadata Transfer code needs to be coupled: either they work together or they are both useless.

3.4.9 Multi-file torrents

Very frequently a torrent comes as a bundle of files, but it is not so obvious that we want to download all those files. It may be, in fact, that we only need one or two of the files, and it doesn't seem convenient to us that we need to wait for the whole download to complete in order to get the two files we need.

Here comes into play the ability to choose which files to download: since we know which pieces belong to which file, by selecting a file we mean that we will only request the pieces belonging to that file, so we will obviously be served only those pieces and not the others.

This feature is still in development in Torrent but we hope it will be completed soon, since it comes very handy for the final user.

4 Seeding with BitTorrent

In this chapter we will talk about seeding: what it is, how it works and how it can be improved. Seeding is crucial to the BitTorrent protocol since it's what actually enables the distribution of data, so it is important to come up with good seeding strategies.

4.1 What is seeding

Seeding is the word used in BitTorrent networks to indicate the uploading of pieces to other peers. There is a distiction to be made:

- **seeding** is the act of uploading pieces done by peers who have the complete data
- **uploading** is the act of giving pieces to other peers done by peers that are still downloading the data

Let's introduce some more terminology:

- **seeder**: a peer who has the complete data belonging to a certain torrent. This kind of peer can only upload pieces
- **leecher**: a peer who is downloading the data belonging to a certain torrent. This kind of peer can both upload and download pieces
- **free-rider**: a peer who is downloading the data belonging to a certain torrent but it only downloads pieces, it does not upload a single piece to other peers

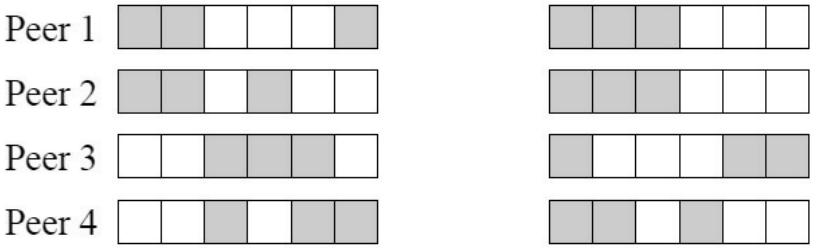

Figure 4.1: Example of piece overlap.

We can easily see that free-riders can be considered as malicious peers, since they do not contribute to the distribution of data in the network – what BitTorrent was specifically designed for.

Thus, what we - as peers in a BitTorrent network, downloading a torrent - want is to limit the impact of free-riders, in such a way that we do not waste upload bandwidth to peers which do not give back to us and concentrate on uploading to peers that can be useful to our download phase.

4.2 BitTorrent basic download strategy

As an introductory concept, we note that the distribution of pieces of a given torrent in the network contributes heavily on the download times of that torrent. In Figure 4.1 we can see that there may be more copies of some piece than of some other. This is what is called **piece overlap**. The higher the piece overlap, the longer the average download time for that torrent, because when a peer gets to the end of the download (i.e. it has to download the last pieces) it will, on average, have to wait some time before connecting to a peer that has them, since the pieces are not distributed equally. Vice versa, if the piece overlap is low, it means that the pieces are distributed fairly equally between the peers, and thus it won't take much longer to download some pieces rather than some others.

So what we want to look for when designing a protocol for distributing data between peers (such as BitTorrent) is to minimize the piece overlap. In the download phase, this is done by the code that regulates the download strategy of the peers. Usually BitTorrent clients operate (or at least *should* operate) based on what is known as the **Local Rarest First (LRF)** algorithm: based on the peers that a client see in the network, it can calculate the frequency of each piece (since every peer sends its bitfield immediately after the handshake) and begin asking the rarest pieces first. If every peer acts in this way, the piece overlap is

automatically minimized, since there won't be pieces that are *much more popular* than others.

This is a very basic way to achieve good (in the sense of *low*) piece overlap. As a note, it has been proved in [12] that if a peer asks pieces he needs in a random way, the resulting piece overlap will be very close to the one provided by the LRF algorithm.

4.3 BitTorrent basic upload strategy

It might be perceived as an overkill to devote a whole section to this, but it's better to be totally clear: as long as a peer is unchoken by an uploader, the uploader will give the peer the pieces it will ask. No headache on that.

Now, to elaborate a little bit: how does a peer get unchoken by an uploader? First of all we recall that, on average, a client has 5 upload slots: 4 of these are used for *rational unchoking*, which we described in Section 2.4.1 of Chapter 2, and the remaining one is used for *optimistic unchoking*. We distinguish now two cases:

1. *the uploader is a leecher*: In this case the 4 voluntarily unchoken peers are chosen based on which pieces they have: if they look interesting to the uploader than he will unchoke them in the hope that they will unchoke him, otherwise he discards them and chooses some more useful peers.

2. *the uploader is a seeder*: In this case the uplader will (on average) not follow a specific strategy for voluntarily unchoking some peers in favor of others, since he already has all the pieces he needs, therefore he cannot favor one peer over another based on what pieces they have

It is then fairly easy to see that the average seeder will unchoke a peer more or less randomly, upload to it for some time, then choke it, choose another peer to unchoke, and so forth. In this way all peers are equal in the eyes of the seeder, they have neither merits nor demerits.

4.4 Super-seeding

Let's suppose we are a seeder and we want to upload some data to the BitTorrent network for the first time (we are the creators of a new `.torrent` file). How do we do it?

The first part, creating the `.torrent`, is fairly easy, since almost every popular client provides an intuitive procedure that lets user create a `.torrent` file from scratch. Then we put the newly generated `.torrent` file to some websites (they may be trackers or indexes[1], or both) in order to let people download them. Suppose some peers are connecting to us and are asking us pieces of that data: since we have all the pieces it is natural that we give it to them, isn't it?

BitTorrent developers have thought of a better way. It has been experimentally shown that in this way some pieces have to be uploaded over and over again, thus wasting the seeder's bandwidth. This is why BitTorrent developers proposed a new algorithm, called **super-seeding**.

What super-seeding does is at the same time very simple and very effective. The algorithm works like this:

while there are pieces to be uploaded for the first time **do**
  $i \leftarrow$ the index of the first piece yet to be uploaded
  create a fake bitfield containing only i
  if peer A connects to us **then**
    send A the fake bitfield
    while we see a peer $B \neq A$ having piece i **do**
      do not inform A of other pieces
    end while
  end if
end while

So if a peer connects to us we present him the fake bitfield indicating that we possess only one piece, and that forces the peer to ask us that piece and we will not advertise other pieces until we see that another peer has got that same piece.

Super-seeding is a very effective seeding mode that preserves the precious seeder's bandwidth by reducing the uploaded data from the standard 150-200% of the total size of a torrent to the more "intuitive" 105%.

[1] A webserver that only lists the `.torrent` files it knows and lets users download them; it does not usually act as a tracker, too.

4.5 The problem of free-riders

It is well known that some peers, called *free-riders* as we said before, do not bother to upload a single piece to the network they are connected to (in the case of BitTorrent, the set of peers which are downloading the same `.torrent` file is called *swarm*), either because they set the upload bandwidth to zero or because they are programmed in a way that prevents uploading.

These peers are particularly dangerous because not only they do not contribute to the distribution of the data they are downloading, but because they waste precious uploaders' bandwidth, damaging other more honest peers. Sometimes, they even get faster downloads than honest peers.

A natural question comes to mind: is there a way to prevent (or at least limit) the free-riding phenomenon?

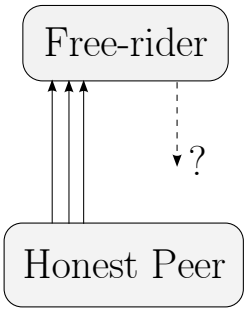

Figure 4.2: Free-riders do not upload pieces to other peers.

4.6 Damaging free-riders as leechers

Various researchers have proposed many methods to prevent the free-riding phenomen, some of which happens to be easily applicable and effective, while many others in our opinion do not provide significant improvements to the standard implementations ([3], [5] and [7]).

We reviewed some of such algorithms and techniques that claimed to improve download times for honest clients and to reduce the performance of bad clients (such as free-riders); after reading a decent amount of papers in the subject, we picked the ones that looked most promising in terms of real effectiveness concerning the damage to free-riders – without causing longer download times for standard peers.

4.6.1 Sorting-based unchoking algorithm

As described in [10], "Availability of seeding capacity can have a significant effect on BT, e.g., it can compensate for the asymmetric bandwidth scenarios in the Internet. At the same time, it can degrade the fairness and incentive properties of the system, as free-riders can finish their downloads with reasonable performance by relying on the seeds. (Not only do they not contribute to the systems' upload capacity, they also effectively reduce the performance gains that seeds provide.) Thus, intuitively, appropriate use of seeding capacity can have a significant effect on performance of both, contributing leechers as well as free-riders."

The authors compare in figure 4.3 the average download time with respect to the average seeding time of some data. It is evident that free-riders have average download times equal, if not better, to those of honest peers. This is enough to indicate clearly that free-riders exploit the seeders' upload bandwidth, damaging honest peers.

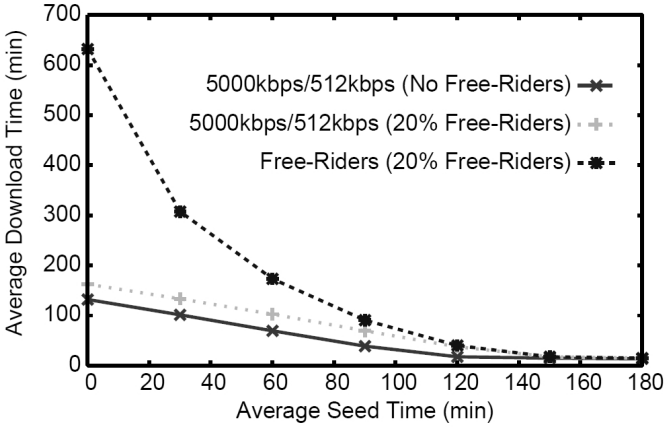

Figure 4.3: Average download time compared to average seeding time.

A solution to this is to "prioritize the use of seeding capacity to only certain portions of a file's downloading process". In figure 4.4 we can see that free-riders are damaged by the algorithm because they rely much more heavily on seeders than the honest peers.

There are many practical approaches to the implementation of this intuition, and the authors propose the following two:

- **Sort-based** (N): In this scheme peers are sorted based on the number of pieces they have, then the client unchokes N of them.
- **Threshold-based** (K,N): This is very similar to the previous scheme, but

here N peers are unchoked that have either $[0...\frac{K}{2}]\%$ or $[\left(100-\frac{K}{2}\right)...100]\%$ of the pieces composing the torrent

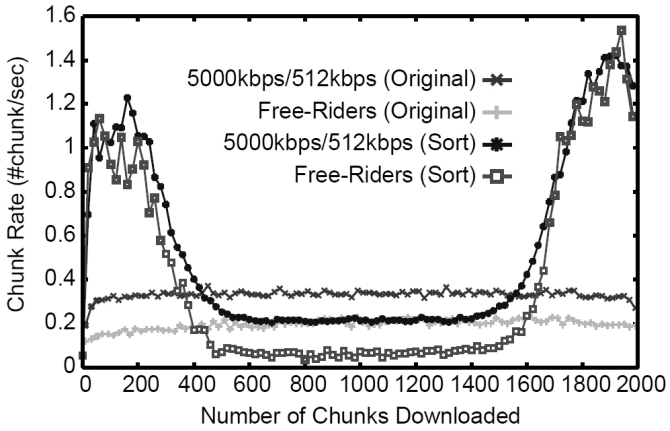

Figure 4.4: Effects of seeding prioritizing on free-riders and standard peers.

Some experiments were conducted on a PlanetLab[2] testbed to evaluate the proposed algorithms; the results are shown in figure 4.5. It can be easily seen that, on average, the algorithm that both damages free-riders without causing lower download rates to honest peers is the sort-based algorithm described above, so that's the one we decided to implement for Torrent and its implementation will be described in the following chapter.

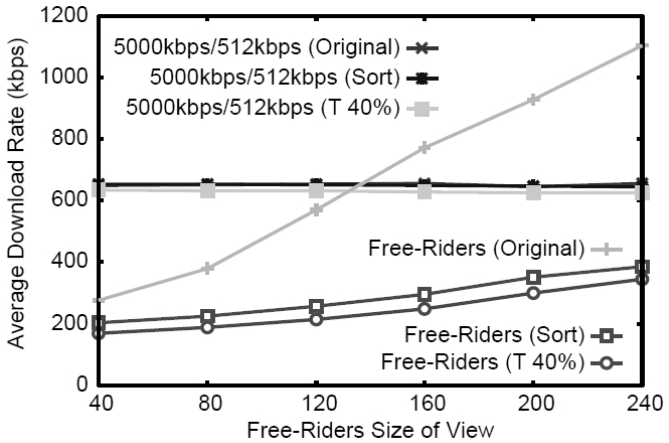

Figure 4.5: Average download rate of peer compared to the number of neighbors they have.

[2] `http://www.planet-lab.org/`

4.7 How to make seeders aware of free-riders

We have seen how leechers can damage free-riders by checking the respective number of uploaded pieces, but how can a seeder damage a free-rider? It is in the interest of honest peers making seeders aware of the fact that they're uploading pieces to a free-rider, since the free-rider is stealing precious bandwidth that could speed up the leechers' download, in some cases getting better average download rates than the honest peers downloading the same torrent (as demonstrated in [17]).

Literature focuses on the leecher's perspective, since it is indeed easier to identify a free-rider when it is possible to check the number of pieces it uploads (which, in the strict sense of free-rider, will be zero). Still, some attempts at detecting free-riders have been researched and implemented, as the *BarterCast protocol* described in [18], which is based on a reputation mechanism for peers and the *Ford-Fulkerson* algorithm for the computation of *maxflow* in graphs.

We decided to try a simpler approach, that doesn't require any modifications to the protocol itself, since it makes use of the already-widespread Peer Exchange mechanism as a transport layer.

First of all, we note that the majority of free-riders does not change the BitTorrent protocol to their advantage, but they simply refuse to upload pieces to other peers (setting the upload bandwidth to 0 KB/s, for example). This means that they keep sending usual messages such as `Choke` and `Unchoke`.

We then keep track of all the peers to which we send a `Request` message after they have unchoken us, saving also the time at which we sent the message. If, after a given time interval (that, through some experimentation, proved to be optimal at around 2 minutes) the peer has yet to send us the piece we requested, we label it as a free-rider and we save in the dropped list of the Peer Exchange message to be sent next. Then we will send, as usual, the PEX message to all the peers we are connected to that support the PEX extension (nowadays supported by the vast majority of clients); the peers that receive it will find the IP and port of the free-rider we found, so they'll save it in their database of bad peers and, hopefully, propagate the information by means of other PEX messages. In this way, part of the network will soon be flooded with PEX messages that classify the free-rider as a bad peer and it is likely that some seeders will get those PEX messages, so they can save the IP of the free-rider and refuse to connect to it when they find it.

If the supposed free-rider was instead a honest peer with a dynamic IP address

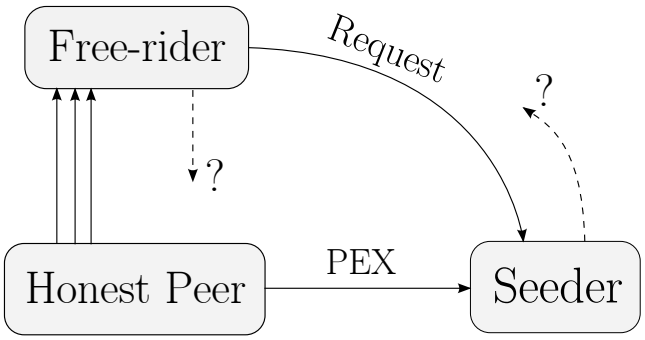

Figure 4.6: With our proposed algorithm a seeder will refuse to connect to a free-rider it knows through a PEX message it received.

that didn't send us the piece we requested because it disconnected from the network, we don't cause unjustified damage to it, since the next time it will reconnect to the network it will get, with very high probability, a different IP address. In the case of peers with a static IP address we could slightly modify the traditional Peer Exchange protocol by blocking the peer for a period of time and then sending a PEX message containing the IP address of the peer in the list of added peers, thus switching the peer's reputation back to good. If the peer is indeed a free-rider, the frequent switching from good to bad and viceversa would effectively lower its download rate.

Note that if the peer sends us bad pieces (that is, pieces that fail the hash check) twice, it will be inserted in the dropped list anyway, since Torrent already provides this feature (known as *Smart ban* algorithm).

4.7.1 Checking algorithm effectiveness

One thing that we want to avoid at all costs is to label as free-riders perfectly honest peers just because they didn't have enough time to send us some pieces. We then implemented, as a control mechanism in the testing phase of our procedure, a linked list to keep track of all the peers we classified as free-riders: every time we received a piece we would check if the linked list contained the peer that sent us the piece; if so, we would print a message informing us of the problem.

After some experiments with the time interval used as a timeout, we found out that 2 minutes is plenty of time for a peer to send us a piece if it's willing to do so, without the risk of getting erroneously labeled as free-rider (we observed that 1 minute was too little in some rare cases).

4.8 Improving protocol fairness

Another interesting problem concerns the protocol fairness, that in the case of peer-to-peer applications can be defined as *the number of pieces a peer downloads versus the number of pieces it uploads*. So, as we defined it, the fairness can be:

- < 1: In this case, a peer uploads more than it downloads
- $= 1$: The case of ideal fairness; a peer gets as much as it gives
- > 1: In this lucky case a peer downloads more than it uploads

Protocol fairness can be enforced by the rules that govern the protocol itself: if a protocol is designed in such a way that a peer can receive a piece if and only if it has uploaded a piece, then every peer is bound to be generous in uploading if it wants to achieve high download rates.

In [16] the authors conduct a series of experiments on the BitTorrent protocol; when discussing its fairness, they state that the BitTorrent protocol is far from being fair, as can be seen in figure 4.7

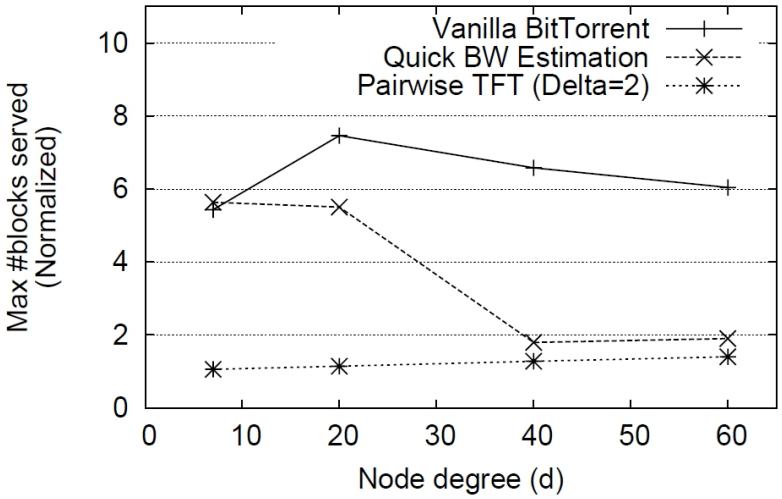

Figure 4.7: Number of pieces uploaded with various uploading algorithms.

The line of the plot labeled as *Vanilla BitTorrent* says that a peer has to upload up to 7 copies of a given piece to the network while receiving only one (the results of the plot are already normalized). A ratio of 7 to 1 is not what we think of when we want to define fairness for a peer-to-peer protocol.

The authors then propose the following modifications to the uploading algorithm in order to make the protocol more fair:

- **Quick Bandwidth Estimation**: Instead, if a node were able to quickly estimate the upload bandwidth for all its d peers, optimistic unchokes would not be needed. The node could simply unchoke the u peers out of a total of d that offer the highest upload bandwidth.
- **Pairwise Block-Level Tit-for-Tat**: The basic idea here is to enforce fairness directly in terms of blocks transferred rather than depending on rate-based TFT to match peers based on their upload rates.

We chose to implement the second option since it looked more consistent and effective, not requiring any modifications to the other peers, so we will now focus on that.

4.8.1 Pairwise Block-Level Tit-for-Tat

Here is the mechanism that regulates the experimental uploading algorithm: suppose that node A has uploaded U_{ab} blocks to node B and downloaded D_{ab} blocks from B. With pairwise block-level TFT, A allows a block to be uploaded to B if and only if $U_{ab} \leq D_{ab} + \delta$, where δ represents the unfairness threshold on this peer-to-peer connection. This ensures that the maximum number of extra blocks served by a node (in excess of what it has downloaded) is bounded by $d\delta$, where d is the size of its neighborhood.

Of course, one basic restriction implies that δ must be at least one, so that new nodes can start exchanging pieces (similarly to what the optimistic unchoke does). It is also easy to see that the pairwise algorithm is very strict in terms of how many pieces we can give to a peer without having anything in return, so as soon as the above mentioned condition is not satistified, we will stop uploading pieces to the peer, thus reducing on average our uplink utilization. We then come at a crossroads: choose fairness in favor of upload speed or vice-versa?

To complete the view provided by figure 4.7, we show in figure 4.8 the mean upload utilization for the various uploading methods. We can easily see that the lowest uplink utilization is provided by the pairwise block-level tit-for-tat, since it is a fundamental part of its design the strict enforcement of the "1 piece for 1 piece" rule that necessarily slows down the upload rate.

So we now have a dilemma: good fairness or good uplink utilization? We thought of a solution: we see from the plot in figure 4.7 that the standard upload algorithm causes some peers to upload from 6 to 8 times the data they receive (they are usually high-bandwidth peers), so we propose to use as a fairness threshold (the δ in the formula) a value between 1 and 7, in order to achieve

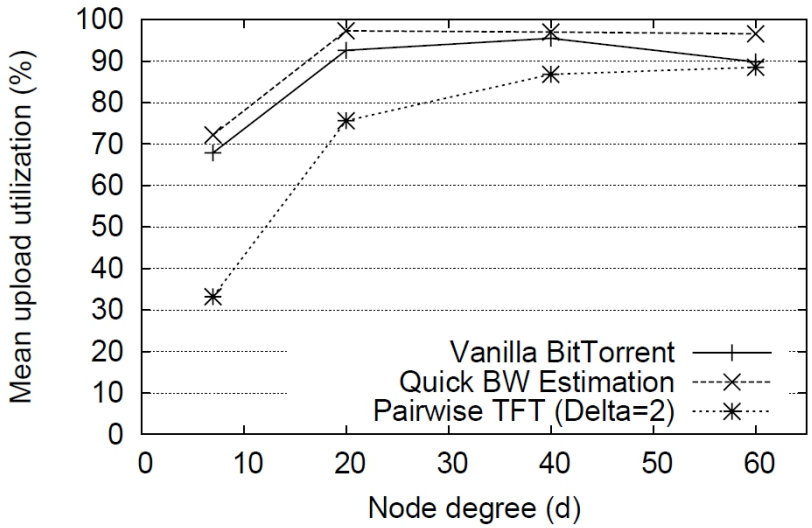

Figure 4.8: Mean upload utilization for the three different methods in analysis.

a trade-off between fairness and uplink utilization. 3 seems to be a good choice for δ, but of course this can be changed if a user feels more generous about its upload bandwidth.

Supporting our argument is again figure 4.8, which shows the uplink utilization for the pairwise algortihm when δ is set to 2. It seems to us still too inefficient though, when the number of peers we are connected to is low (5 to 20 peers); in fact, we usually reserve no more than 5 slots for uploading, so we have a very bad case when using the pairwise algorithm with low values of δ. Also, keep in mind that very rarely a node in the BitTorrent network uses all of its upload bandwidth – a much more common situation is to set the limit at a very small fraction of it. So when raising that threshold a little bit, setting it to 3 as we said, we get an uplink utilization that's fairly close to the one we would have when using the standard algorithm, but with a gain of about 100% in the protocol fairness.

5 Implementation

In this chapter we will talk about the implementation of the algorithms and techniques described in chapter 4 for limiting the free-riding phenomenon and improving the protocol fairness. We will also describe some other modifications we made to the Torrent code in order to make it more efficient.

5.1 Updating the peer list

This is a very trivial modification but it's what allowed us to contact up to 60-70 peers at a time and to reach download rates around 200-300 KB/s, whereas with the previous code the plugin could hardly be seen going over 100 KB/s.

In class `DownloadTorrent` there is a method called `updatePeerList()` that gets called when we receive the tracker's response containing the list of (at most) 50 peers sharing the torrent we need. This method contains the lines that effectively try to contact the peers we have in a data structure we call the **peer database**, defined in class `PeerDatabase`, inside the package `torrent.manager`. The problem with `updatePeerList()`, though, is that it is called *only* when we receive a tracker's response, which happens on average every two minutes; that's clearly a waste of time, especially if we have enabled extensions such as the Peer Exchange, which gives us new peers almost every 5 seconds.

What we have done is very simple: every time we receive a new PEX message, containing a list of good peers, we call the lines that update the peer database and try to contact the first peers, until we reach the maximum number of peers we can simultaneously be connected to (this value is set to 60 by default).

Even though this is a minor change, it allows Torrent to have a full connection list almost all the times, which translates into much higher download rates for a

given torrent.

5.2 Suggest Piece Sender

As we said in Section 2.5.3 of Chapter 2 when talking about Fast Extension, we said that the `Suggest Piece` message contains as its payload the index of a piece that it's *suggested* to a peer for download; usually this piece coincides with the rarest piece we have seen, in order to adhere to the LRF (*Local Rarest First*) algorithm.

Of course, this is a message that cannot be sent once in a lifetime, since our view of the rarest piece updates with every new peer we come in contact with; we figured that we needed some kind of timed operation that would calculate the rarest piece and send a `Suggest Piece` message every, say, 30 seconds. One lightweight option proved to be the `TimerTask` class provided by Java, which contains an internal timer that activates the assigned Task periodically (the period can of course be set by the programmer).

Thus we created the `SuggestPieceSender` class inside the `torrent.peer.messages.fastextensions` package, extending the `TimerTask` class. `SuggestPieceSender` contains a boolean variable `running` that is used to start or stop sending the messages to the peers. It also has a method, called `buildSuggestPieceMessage()`, that finds the rarest piece and invokes the `MessageSender` to send this message to every connected peer that supports fast Extension.

A `TreeMap` data structure called `peerSet` is used to keep track of all the peers that support Fast Extension, and we have provided `add()` and `remove()` metods for peers that connect or disconnect from us.

The computation of the rarest piece takes place inside `DownloadTorrent`, by means of the `calculateSuggestedPiece()` method, which very simply takes all the connected peers, checks which pieces they have and keeps the count for every piece in an array called `totalPieces`; after that, the array is scanned from beginning to end in order to find the index with the lowest number of occurrences.

The creation of the `SuggestPieceSender` instance associated with a given torrent happens inside `DownloadTorrent`, immediately after connecting to a peer that supports Fast Extension – if subsequent peers support Fast Extension they are only added to the `peerSet`. After that the `schedule` method of the `TimerTask` class is invoked.

We set the time interval for the `schedule()` method to 60 seconds (1 minute),

figuring that's enough for the calculated piece to be fairly accurate (based on the peers we have seen) and not too CPU-consuming.

All of this is done as an attempt to keep the piece overlap for a given torrent as low as possible.

5.3 uTorrent Peer Exchange Refactoring

We talked in Section 2.5.2 about the importance of the Peer Exchange procedure: it allows peers to send and receive information about peers they know and propagate this information to their neighbors, dividing the peers they have come in contact with in two categories, good or bad. In this way the peers that receive the PEX message will know which peers to contact first (the good peers) and which peers to avoid.

In the previous version of Torrent, there was a couple of classes, called `UtPeer ExchangeSender` and `UtPeerExchangeManager` associated with *a single peer*. Class `UtPeer ExchangeSender` used to extend the `PariPariRunnable` class, thus creating new threads. Since there was an instance of `UtPeerExchangeSender` for every peer we came in contact with that supported Peer Exchange, the number of threads was approximately linear with the number of peers, which is obviously a waste.

We decided to refactor the classes that manage the PEX procedure by associating the `UtPeerExchangeSender` and `UtPeerExchangeManager` classes not with a single peer but the `DownloadTorrent` instance associated with a given torrent, which seems more logical. Once we have done so, we needed to change the way in which the `UtPeerExchangeSender` class was sending messaging to the peers, since it was programmed to send the PEX messages only to the peer it was associated with. Thus we inserted a `for` loop in the class that iterates on the list of currently connected peers, for each it checks whether Peer Exchange is supported (which can be easily done by reading the relative byte in the `Peer` instance of the peer) and, if so, sends them the PEX message.

We also changed the class extended by `UtPeerExchangeManager` from `Pari PariRunnable` to `TimerTask`, as for the Suggest Piece Sender, since we don't really need a thread for sending PEX messages once every minute. After having implemented the `run()` method in `UtPeerExchangeSender`, it was sufficient to schedule the task in `DownloadTorrent` with the same `Timer` object used for `SuggestPieceSender`, since the two scheduled tasks work independently even if they are created on the same `Timer` instance.

5.4 Super-seeding

In order to implement the super-seeding algorithm in Torrent we had to modify some lines of class `DownloadTorrent`, since that's where the core of uploading happens.

First of all, we created a boolean variable called `superseeding`, initially set to `false`, that tells wether we are in super-seed more or not. Then we needed something to keep track of pieces we uploaded the first time and to which peer (meant as a *(IP, port)* pair): for this purpose we created a `TreeMap<Integer, IPeer>` object called `superseedSentPieces` which conveniently allows us to retrieve the peer associated with a given piece index.

We also created an integer counter, called `superseedCounter`, that serves as an incremental pointer to the index of the next piece we are considering for advertising in the bitfield. After that comes the necessary bitfield, as a byte array, called `superseedBitfield`.

When the constructor of `DownloadTorrent` gets called, all the variables and objects we created for the super-seeding algorithm are initialized. Then, inside the body of the `parseMessageReceived(Message m)` method, when we receive a bitfield from a peer and we are going to reply with our bitfield, we first check if we are in super-seed mode: if so, we send the peer our fake bitfield (containing, for the first iteration, only the first piece), otherwise we send the complete bitfield.

We then implemented the `checkCopiedPiece(IPeer p, int i)` method, whose job is to check, whenever we receive a bitfield or a `Have` message, the peers we have uploaded in the first place the pieces we see. Then we increment the `superseedCounter` by one and we inform them through a `Have` message that we possess another piece.

The super-seeding feature is something that we let the user start or stop at runtime; this is why we created the `superseeding` boolean variable (so that we can check every time if we have enabled or disabled the super-seeding feature) and we added the `superseed` command on console. The complete command defined in `TorrentConsole` is: `superseed [id]`, where `id` is the integer that identifies the file we want to seed through super-seeding mode. There is no `stop` command on console for the super-seed mode because the protocol states that we cannot send different bitfields to the same peer so, instead of flooding it with a lot of `Have` messages, it is advised that we close the client and reconnect to the network in normal seeding mode.

5.5 Sorting-based unchoking algorithm

We recall from Section 4.6.1 in Chapter 4 that the sorting-based unchoking algorithm works by unchoking the first N peers sorted based on the number of pieces they own; in particular, if a torrent has P pieces, we take the peers that are more distant from the arithmetic mean $\frac{P}{2}$.

The peer sorting algorithm that was used before was based on the download and upload rate of each peer. Inside the package `torrent.peer` there are a couple of classes, namely `DLRateComparator` and `ULRateComparator`, whose job is precisely to sort peers on the basis of the download and upload rate we have registered for them. Each of these two classes implements the Java `Comparator` interface, parametrized to the `IPeer` object. Every implementation of this interface must provide a `compare(Object a, Object b)` method between two objects, that can conveniently be called afterwords by the `sort()` static method of the `Collections` and `Arrays` classes.

In the same package of those two comparators we created a new class, called `PieceComparator`, which implements a `compare(IPeer a, IPeer b)` method that tells which one between A and B is the furthest away from the arithmetic mean of the number of pieces composing the torrent. In order to get the number of pieces A and B have, we implemented a `getNumPieces()` method in the `Peer` and `IPeer` class that returns the number of pieces a peer has – which we can see from the bitfield it sends us and from the subsequent `Have` messages that we read; every `Peer` object has a counter which we use to keep track of the pieces it has. We then proceeded to sort the array of peers for a given download in `DownloadTorrent`, inside the `unchokePeers` method.

What if a user doesn't like this experimental unchoking method and prefers to stick with the more conventional one? We added a switch in the code that, through the boolean variable `expUnchoke`, enables the experimental unchoking algorithm or the standard one; but we don't want a user to even *read* our code, do we? So we also added a line on the XML configuration file of Torrent that enables the activation or deactivation of the algorithm by simply setting to `true` or `false` the `experimental_unchoke` entry.

5.6 PeerChecking algorithm

As we described in Section 4.7 of Chapter 4, our proposed algorithm tries to identify free-riders through a timeout system and informs other peers of the discovered

free-riders by means of a Peer Exchange mechanism.

First of all, we created a boolean variable in the `Peer` class that tells whether the peer has sent us at least a piece or not. We then proceeded to insert into `DownloadTorrent` the data structure that would contain the discovered free-riders, in order to check that we weren't going to damage honest peers. For the task we used a `LinkedList<IPeer>` object called `freeriders`, since we just need to list peers without a specific order. The `PeerChecker` instance associated with a given download is created whenever the respective `DownloadTorrent` constructor is called, then it is added to a `Timer` object through the usual `schedule()` method. The task is called once every minute.

We then created the `PeerChecker` class inside package `torrent.manager`: this is the class that does the job of checking whether a peer is a free-rider or not and of putting it in the dropped list of the `UtPeerExchangeManager` associated with the same `DownloadTorrent` instance of the PeerChecker.

As we already did for other classes that need to perform an action once every given time interval (such as `SuggestPieceSender` and `UtPeerExchangeSender`, in the creation of `PeerChecker` we extended the `TimerTask` class provided by Java, in order to easily implement a scheduled action without using a separate thread.

`PeerChecker` also contains a `TreeMap<IPeer, Long>` object called `request Map`, which stores the peers to which we have sent a `Request` message and the time at which we sent it, so when the checking algorithm wakes up it will subtract the current system time from the time stored in the map for every entry: if the result is greater than 2 minutes, the respective peer is classified as free-rider and put into the `freeriders` linked list and the dropped list of the `UtPeerExchangeSender` instance associated with the same download. In the other case (we receive a piece before 2 minutes have passed) we remove the `(IPeer, Long)` pair from the map.

5.7 Pairwise Block-Level Tit-for-Tat

In Section 4.8.1 of Chapter 4 we described the proposed algorithm that tries to improve the fairness of the BitTorrent protocol. As the reader can realize by reading that Section, it is obvious that this algorithm is only applicable if we (as uploaders) do not possess the whole data composing the torrent – in other words we are leechers for that torrent. If it weren't so (that is, if we were seeders) how could we define the number of pieces we are downloading from a peer? It would of course be zero, since we do not need any pieces; therefore, we would not be

uploading pieces to any peer. That's obviously not how it is supposed to work.

In the `DownloadTorrent` class we created a new method, called `uploadProcedure(IPeer peer, int piece, int begin, int length)`, that contains an `if` statement in its body: if we are seeders, we are not concerned with the number of pieces we have downloaded from a peer and we will simply upload the requested piece to it; on the other hand, if we are leechers, we will begin looking at the number of pieces we have download or uploaded to a peer in a given session. This method is called whenever we receive a `Request` message from a peer, which is asking us a piece.

The `Peer` class already provided the `getDL()` and `getUL()` methods, which return the total amount of downloaded and uploaded bytes, respectively, for that peer. We divide such numbers by the length of a piece for the torrent we are considering (which can be done by accessing the `torrent.pieceLength` variable). After that, we simply check if the condition $U_{ab} \leq D_{ab} + \delta$ (which we previously defined) holds: in the positive case we upload the desired piece to the peer, otherwise we simply ignore its request.

What if a user wants to change the unfairness threshold we provided? It can simply access the Torrent XML configuration file inside the `PariPari/torrent/conf/` folder and change the integer value associated with the `unfairness_threshold` key.

6 Team management

In this chapter we will discuss the main topics that concern the management of a team working in a software engineering project like PariPari. We will talk about the main problems that arise in the development of a plugin and the tools and techniques that have been employed to deal with them.

6.1 Working as a team

The average student pursuing a Computer Science or Computer Engineering degree is not used to developing software in a group. Course projects and personal experiments accustom the student to solitary programming practices that do not work well when applied to a multi-person working environment.

One of the biggest problems we have faced is the very diversified habit of writing comments into the source code and of choosing variable names. Names should be expressive of what the variable is and/or what is does, what's its purpose, and names such as *temp, n, array* should be avoided like the plague. It may look easy but it requires some effort to get rid of such negative habits.

Another problem is posed by the comments to the source code: they should be inserted whenever the code is not straightforward to understand, they should be like guidelines to the programmer that faces those lines of code for the first time and is trying to figure out what is going on there. On the other hand, comments should be succint, not invading the code (we are still writing computer programs, not books) and as clear as possible, making it easy for the programmer reading the code for the first time to understand its inner workings.

When working with university students we come across other kinds of problems, too. One big issue is time management: students work at the project when they have time, between one exam and the other and between one party and the

next, so their contribution to the work is intrinsically discontinue. One way to solve this issue is to set mid-term goals and keep track of how the work advances more or less once a week, in order to see if there are some problems causing significant delay.

Like in every other cooperative endeavor, communication is of fundamental importance. The preferred media is private email or online groups like the ones provided by Google, while for more timely help instant messaging of any kind proved to be very handy. Once in a while, like once a month or when discussing important issues for the plugin's development, meetings are arranged in order to have a better view on the state of the work and also to keep a sense of belonging to the team.

6.2 Code versioning

When working for a software development project like PariPari it is fundamental to have a *version control system*, or *code version control*, that is a system for efficiently managing the changes to documents, programs, and other information stored as computer files. It is in fact common for the software developers to be working simultaneously on updates, which at the very least can lead to synchronization problems (programmers working on versions of the code which are not updated).

Version control provides a system for easily retrieving different versions of the code software in order to identify the sources of bugs and to prevent developers to step into each other's way when writing code. In PariPari the working copy of a plugin is stored in a folder called *trunk*, while the copies which are adding new features and need some testing before being accredited as finely working copies are stored in a folder called *branches*.

In order to begin working on the software developers need to *checkout* the desidered branch or trunk version of the plugin they need to modify. This creates a local copy of the software which they can modify as much as they like; when they're satisfied with their work (which doesn't need to be the total completion of their task) they can *commit* their changes in order to let everyone else see them if they want to and to prevent losing everything if something goes wrong on the machine they are working on.

In PariPari the versioning system of choice is **Subversion** (also known by its abbreviation **SVN**), which is very convenient because it comes with a plugin for Eclipse, the editor used by every PariPari developer. In this way students

involved in the project can easily perform checkouts and commits of their code, see changes from one version to another, revert to an older version if something has gone wrong, and many other useful things.

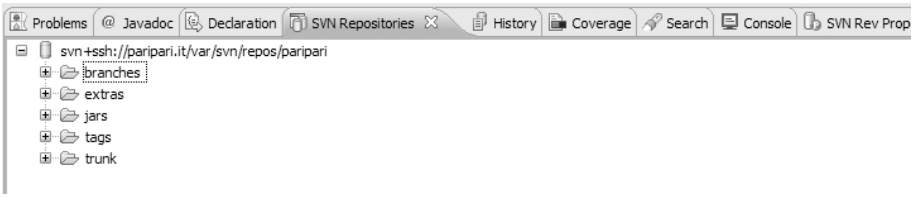

Figure 6.1: Main folder organization for the PariPari SVN repository.

6.3 Testing

Testing is of fundamental importance for every software project, since it is easy even for not-so-complex programs to behave in an unexpected way when provided with some input or after some time of execution. Every PariPari developer is then taught how to perform some automated tests, in order to catch and fix the most apparent bugs before releasing the code.

PariPari adopts the *eXtreme programming (XP)* software development methodology, which is a type of *agile software development*[1] developed by Kent Beck during his work on the *Chrysler Comprehensive Compensation System* payroll project. Some of the pillars of the XP methodology are *unit testing* and *pair programming*, and we tried to apply these techniques when working for Torrent.

Unit testing is a method by which individual units of code are tested to determine if they are fit for use. Since PariPari is written in Java, a class is considered a unit and the testing for that unit involves testing for all its methods, especially edge cases (such as invalid input, `null` parameters, and so on). In PariPari we use *JUnit*, written by Kent Beck himself, as the *unit testing framework* of choice.

Pair programming is a methodology in which two programmers work at the same workstation: while one developer is writing the code, the other reviews each line as it is typed in. For obvious organizational reasons (students having different timetables) it is hardly feasible to get two programmers working together at the same time (the only time we were able to do so was during the Software Engineering class in which students were working for the various PariPari plugins), so we adopted a slightly different strategy: one developed writes the code and another

[1] A group of software development methodologies based on iterative and incremental development.

one writes the tests for it, so the first developer is not tempted to write *ad-hoc* tests in order to show that his code works properly. The spirit remains the same, because writing tests requires some amount of knowledge of the code being tested, so the developer writing the tests will inevitably check the code sometimes. In this way we can reduce the number of bugs released with new versions of the plugin.

Conclusions

We have seen that free-riders do pose a problem to the BitTorrent protocol, since they do not upload pieces to the network and they consume precious uploaders' bandwidth, sometimes getting lower download times than the average honest peer. We have seen some ways in which the free-riding phenomenon can be constricted to some extent, while at the same time improving the protocol fairness (number of uploaded pieces versus number of downloaded pieces). In this way we were able to get upload rates comparable with the previous version of Torrent but with the added feature of at least not favoring free-riders.

We also dealt with how a simple change to the lines of code that manage the update of the peer list boosted the average download rate for Torrent by a factor of 2 to 3, moving the plugin performance closer to the widely popular BitTorrent clients such as uTorrent and Azureus/Vuze.

Much can still be done, though: important extensions such as the DHT and the Metadata Transfer have yet to be completed and integrated into the official code, download rates can certainly be improved and, maybe most importantly, Torrent still needs a fully functional GUI.

Nonetheless, we think that our work has led to some fair improvements in the plugin's performance (whose fundamental job is to download and upload data) and to advance, even if just a little, the state of the BitTorrent network by damaging selfish peers.

A BEncode

Bencode is the encoding used by the peer-to-peer file sharing system BitTorrent for storing and transmitting loosely structured data. It supports four different types of values:

- byte strings
- integers
- lists
- dictionaries

While less efficient than a pure binary encoding, bencoding is simple and (because numbers are encoded in decimal notation) is unaffected by endianness, which is important for a cross-platform application like BitTorrent. It is also fairly flexible, as long as applications ignore unexpected dictionary keys, so that new ones can be added without creating incompatibilities.

A.1 Integers

An integer is encoded as `i<number in base 10 notation>e`. Leading zeros are not allowed (although the number zero is still represented as `0`). Negative values are encoded by prefixing the number with a minus sign. The number 42 would thus be encoded as `i42e`, 0 as `i0e`, and -42 as `i-42e`. Negative zero is not permitted.

A.2 Byte strings

A byte string (a sequence of bytes, not necessarily characters) is encoded as `<length>:<contents>`. The length is encoded in base 10, like integers, but must be non-negative (zero is allowed); the contents are just the bytes that make up the string. The string `spam` would be encoded as `4:spam`. The specification does not deal with encoding of characters outside the ASCII set; to mitigate this, some BitTorrent applications explicitly communicate the encoding (most commonly UTF-8) in various non-standard ways.

A.3 Lists

A list of values is encoded as `l<contents>e`. The contents consist of the bencoded elements of the list, in order, concatenated. A list consisting of the string `spam` and the number 42 would be encoded as: `l4:spami42ee`. Note the absence of separators between elements.

A.4 Dictionaries

A dictionary is encoded as `d<contents>e`. The elements of the dictionary are encoded each key immediately followed by its value. All keys must be byte strings and must appear in lexicographical order. A dictionary that associates the values 42 and `spam` with the keys `foo` and `bar`, respectively, would be encoded as follows: `d3:bar4:spam3:fooi42ee`. (This might be easier to read by inserting some spaces: `d 3:bar 4:spam 3:foo i42e e`.)

There are no restrictions on what kind of values may be stored in lists and dictionaries; they may (and usually do) contain other lists and dictionaries. This allows for arbitrarily complex data structures to be encoded.

Bibliography

[1] *The BitTorrent Protocol Specification* `http://www.bittorrent.org/beps/bep_0003.html`.

[2] *Bittorrent Protocol Specification v1.0* `http://wiki.theory.org/BitTorrentSpecification`.

[3] M. Coates, R. Thommes, *BitTorrent Fairness: Analysis and Improvements*. In *WITSP '05*, December 2005.

[4] F. Peruch, *Paripari: Connectivity optimization*, 2011

[5] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, A. Venkataramani, *Do incentives build robustness in BitTorrent?*. In *Proc. 4th USENIX/ACM NSDI*, April 2007.

[6] M. Bonazza, *PariCore*, 2009

[7] Chi-Jen Wu, Cheng-Ying Li, Jan-Ming Ho, *Improving the Download Time of BitTorrent-like Systems*. In *Proc. of IEEE International Conference on Communications*, 2007.

[8] A. Gallo, *PariPari: Crittografia Torrent*, 2009

[9] A. Aldegheri, *Paritorrent: performance refactoring*, 2010

[10] A. Chow, L. Golubchik, V. Misra, *Improving BitTorrent: A Simple Approach*. In *Proc. of the IPTPS*, 2008.

[11] D. Turchetto, *PariPari Torrent: libTorrent and Fast Extension*, 2009

[12] A. Legout, G. Urvoy-Keller, and P. Michiardi, *Rarest First and Choke Algorithms are Enough*. In *Proc. of IMC*, 2006.

[13] R. Ampezzan, *PariMulo 2009*, 2009

[14] M. Muscarella, *PariMulo: Reengineering, 20011*

[15] F. Mattia, *PariMulo: Kad*, 2011

[16] A.R. Bharambe, C. Herley, *Analyzing and Improving BitTorrent Performance*. Microsoft Research Technical Report, February 2005.

[17] M. Sirivianos, J.H. Park, R. Chen, X. Yang, *Free-Riding in BitTorrent Networks with the Large View Exploit*. In IPTPS, 2007.

[18] M. Meulpolder, J.A. Pouwelse, D.H.J. Epema, H.J. Sips, *BarterCast: Fully Distributed Sharing-Ratio Enforcement in BitTorrent*. Delft University of Technology Technical Report, 2008.

List of Figures

List of Tables