UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

PARIMULO: REENGINEERING

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

CORRELATORI: Ing. Michele Bonazza e Ing. Paolo Bertasi

LAUREANDA: *Martina Muscarella*

Padova, 24 Ottobre 2011
A.A. 2010/11

To myself,

because I deserve it.

Acknowledgements

In the first place I want to thank my advisor, Prof. Enoch Peserico, for his support and encouragement, but also for giving me many opportunities, such as to be the Mulo Team Leader, allowing me to grow up as an engineer. I'd also like to thank my assistant supervisors, Michele Bonazza and Paolo Bertasi, who lead and supported me during these three years of work on PariPari, teaching me things that I could never have learnt at any university class (such as to `ping` and to use a screwdriver).

Then I am grateful to all the other PariPari Team Leaders and the past and present Mulo developers. In particular way Francesco Peruch for every Connectivity integrations - hoping NIO will be the last one - and of course for the record of 150 emails sent in a couple of midsummer nights. My gratitude also goes to Mattia Samory, because Mulo now has what it deserves: a fresh, pretty and green graphic interface. Obviously I must also cite Vincenzo Cappelleri who has been really helpful me dealing with the candyman. Finally I want to thank my best developers Francesco Mattia and Christian Piccolo for the memorable Kad integration (I am so happy we are all still alive and still have both hands) and the former Mulo Team Leader, Roberto Ampezzan, for every public variable in the code and the hints about Acn and Rly.

My gratitude also goes to my friends of the Advanced Computing Group, who are always happy to listen, tell and share all my crap. I especially thank Federica Bogo and Marco Bressan who supervised me while I was trying not to break the hardware components of the Eridano cluster.

I'd like to thank all my friends: Jari Gianesini, Ruggero Rappo, Alice Bordin, Giulio Drusi and Alice Dal Maso for all the time spent together (I know you deep down could not survive without me); especially my best friend Serena Crocetta, because my life without her would be as M&M's without colors and chocolate. Obviously I have also to thank my university fellows and friends: Dario Turchetto, Giacomo Portolan, Lorenzo De Stefani and Elisa Tosello (the order is deliberately not alphabetical). Of course I really thank everyone who ever gave me a lift.

Finally but not less importantly, I am grateful to my parents for believing in me and investing in my university career: I really hope they can be proud of me. I'd also like to thank my dear little brother, who is not really dear and surely not little, but I love him anyway.

Contents

Abstract

Nowadays broadband is very widespread around the world and computers with a considerable amount of computing power are available to all, both computer engineers and amateur users, at lower and lower prices. In this context an amazing growth of software applications that offer a wide variety of useful services over the Internet is taking place, such as distributed storage, peer-to-peer networks, instant messaging, web services, file sharing, cloud computing and many more.

PariPari is the largest open source project of software engineering in the University of Padova; it is currently being managed by many students of the Department of Information Engineering, and its goal is to create a easy-to-use peer-to-peer application exhibiting scalability, transparency, efficiency, security and anonymity. It is moreover designed as a multi-functional application, whose services range from file sharing and distributed storage to video communications and conferences. Its plug-in architecture allows and eases the introduction of new features of increasing variety and complexity at anytime.

Mulo is one of the modules that provides file sharing services: it is an eMule client and it allows users to join the eDonkey and Kad networks by sharing and downloading files. Mulo is at a mature stage of development, supporting almost all of the main features needed for an eMule client to work considerably well. Furthermore Mulo provides innovative yet-to-be-seen features that no other existing client has.

This thesis illustrates how Mulo has been reengineered and optimized through a first phase of refactoring and the following integration of two basic PariPari plug-ins, Connectivity NIO and the GUI. The former allows asynchronous communications across the network and its adoption has reduced the amount of threads needed for Mulo to run, thus improving its overall performance. The latter introduces a new user-friendly graphic interface that replaced the old text-based interface that's been used until some months ago.

Sommario

Al giorno d'oggi le connessioni a banda larga sono considerevolmente diffuse sia fra ingegneri che utenti amatoriali, grazie alla sempre maggiore accessibilità del loro costo. In questo contesto ha avuto luogo uno straordinario sviluppo di applicazioni software che offrono un'ampia varietà di servizi Internet, come storage distribuito, reti peer-to-peer, messaggistica istantanea, servizi web, condivisione dei file, cloud computing e molto altro ancora.

PariPari è il più grande progetto open source di software engineering dell'Università di Padova, attualmente gestito e sviluppato da un vasto gruppo di studenti del Dipartimento di Ingegneria dell'Informazione; il suo obiettivo è creare una applicazione pere-to-peer di facile utilizzo che richiede qualità come scalabilità, trasparenza, efficienza, sicurezza e anonimato. Essa è inoltre progettata per essere un'applicazione multi-funzionale, i cui servizi spaziano da file sharing e distributed storage fino a video comunicazioni e conferenze attraverso Internet, e la sua architettura a plug-in permette e facilita in qualsiasi momento l'aggiunta di nuove funzionalità di crescente varietà e complessità.

Mulo è uno dei moduli che forniscono servizi di file sharing: è un eMule client che permette agli utenti di partecipare alle reti eDonkey e Kad condividendo e scaricando file. Lo stato dell'arte di Mulo è ad un livello davvero avanzato: supporta tutte le principali funzionalità necessarie ad un client eMule per operare discretamente. Mulo dispone anche di funzionalità innovative, non ancora presenti in altri client.

Questa tesi illustra come Mulo sia stato reingegnerizzato e ottimizzato attraverso una prima fase di refactoring e una successiva integrazione di due basilari plug-in di PariPari, ossia Connectivity NIO e GUI. Il primo plug-in fornisce connessioni asincrone e la sua integrazione ha ridotto lo spreco di thread, migliorando quindi notevolmente le prestazioni di Mulo. L'altro invece ha introdotto una nuova interfaccia grafica user-friendly, in sostituzione della vecchia console, al fine di rendere l'intera applicazione di PariPari più appetibile agli occhi degli utenti.

Overview

This thesis discusses the eMule plug-in of PariPari, *Mulo*, with a special focus on its recent re-engineering. Technical details are thoroughly explained across on-line documentation and other about Mulo, references to which are suggested at the right time.

Chapter 1 provides a succinct presentation of the PariPari project and its plug-ins, focusing on Mulo and the plug-ins to which it is directly linked.

In Chapter 2 the eMule networks are introduced and their several protocols are accurately described, with special attention to all the mechanisms needed for an eMule client to work.

Chapter 3 lists all of the features supported by Mulo and those not supported yet. Moreover we describe all the new functionalities that have been introduced by Mulo developers that other clients don't have. In this chapter some important research projects concerning Mulo and file sharing are also presented.

We explain in details the complete reengineering process in Chapter 4 by focusing on the integration of two PariPari plug-ins - Connectivity NIO and the GUI - that allowed asynchronous connections over the network and introduced a new user-friendly graphic interface, respectively.

Finally Chapter 5 is about the project organization, describing the job the author had as Team and Tester Leader and listing the rules adopted in Mulo team to allow a peaceful teamwork. A full list of the tools that PariPari developers learn to exploit is also presented.

Chapter 1

The PariPari project

PariPari is the largest software engineering project at *DEI*[1], currently having more than sixty students work on it. A peculiarity of this project is that it is totally managed and developed by students, both Bachelors and Masters in Computer Science. It is written in *Java*, not only because it is by far the most widespread programming language among students at DEI, but also because it inherently supports multiple platforms. In fact PariPari is launched as a *JWS*[2] application that anybody can run just by clicking on a web page, requiring no installation no matter what OS is installed.

The final goal is to create a multi-functional *P2P*[3] network application. A P2P network is a distributed architecture that partitions tasks or workloads between nodes, said *peers*. Here peers are equally privileged and equipotent participants. PariPari is a collection of all the most well-known and useful services available on the Internet, so anyone will be able to use them by simply downloading the application.

1.1 The DHT structure

The network layout is based on a *DHT*[4], providing a high degree of scalability, decentralization and fault tolerance. In the classic mechanism each node is assigned a unique identification number nID in a d-bit address space. A resource is

[1] *Department of Information Engineering* of University of Padua

[2] JWS - **J**ava **W**eb **S**tart is a framework developed by Oracle that allows users to start Java application directly from the Internet using a web browser.

[3] P2P - **P**eer-**to**-**P**eer

[4] DHT - **D**istributed **H**ash **T**able

represented by a *key-value* (k, kID) pair, where k is the keyword associated with the resource itself and it is mapped into a unique identification number kID. The kID belongs to the same d-bit address space of the nodes ID and it is calculated by using a well defined hash function h, in such a way that $kID = h(k)$. Then the notion of *distance* is defined among IDs by adopting the XOR metric, described in section 2.3.1.

Resources are stored and retrieved by nodes thanks to the shared address space and in particular two fundamental primitives are provided: *store* and *search*. When a node stores a resource r, corresponding to the key-value (k, kID) pair, r is assigned to the node of the network whose ID is the closest to kID. This node is then contacted by everyone that searches for the resource r.

Thanks to the DHT structure, a node in the network can contact any other node in $O(logN)$ hops, where N is the number of active nodes. In order to achieve this, each node n keeps contacts with a small number m of nodes in the other half of the network (with respect to n's nID), m nodes in the other quarter, m in the other eighth and so on. The structure used to store the information about those contacts is usually called *routing table*.

PariPari DHT adopts a value of d equal to 256 and of m equal to 20; although usually m is equal to 1 a bigger value improves the robustness of the network. Nowadays DHTs are used in many application fields and in section 2.3 another well known DHT will be described in detail.

1.2 Plug-ins architecture

The project has a modular architecture and the *Core* is in a manner of speaking its heart, as shown in figure 1.1; every team in PariPari develops its own plug-in to offer some specific service.

There is a differentiation between inner-circle plug-ins and external ones: the former class of plug-ins supplies resources to the latter. The most important inner plug-ins are *Connectivity* and *Storage*, which manage network and disk storage respectively. Finally the user interacts with the external plug-ins, which offers all traditional P2P services (such as *file sharing*, *VOIP*[5], *distributed storage*, etc.)

[5]VOIP - **V**oice **O**ver **I**nternet **P**rotocol is a family of technologies, methodologies, communication protocols, and transmission techniques for the delivery of voice communications and

and server-based ones (such as *IM*[6], e-mail hosting, *NTP*[7], etc.).

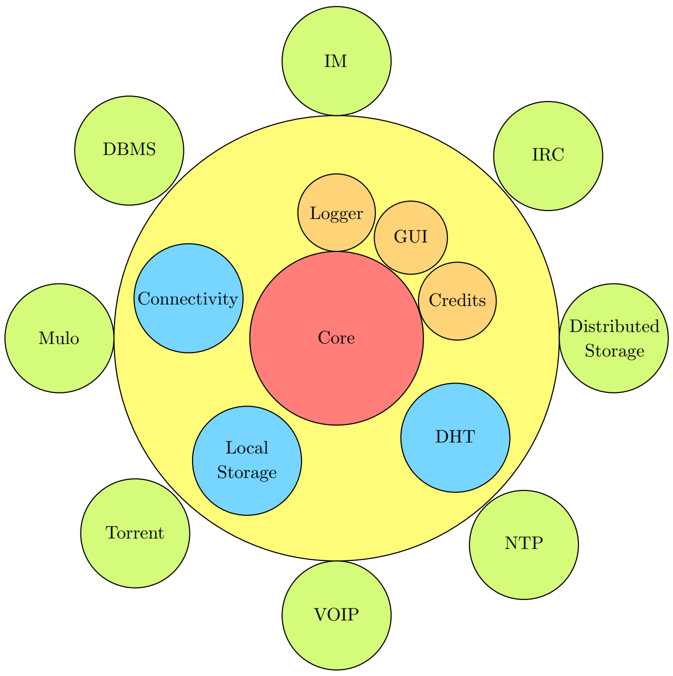


Figure 1.1: The PariPari plug-ins architecture.

1.2.1 The Core

The Core is not a real plug-in: it instead is the kernel of PariPari. Its current version is known as TALPA[8] and it was born at the beginning of 2009. For more details about the Core see reference [12].

Its main functions are managing plug-ins, routing their messages and protecting users and good plug-ins from malicious ones. In fact every request for a resource supplied by an inner-circle plug-in must pass through the Core. This means that malicious plug-ins are not allowed to write on disk or on a socket, unless users explicitly gives their authorization.

multimedia sessions over IP networks

[6]IM - **I**nstant **M**essaging is a form of real-time direct text-based communication between two or more people using personal computers or other devices, along with shared clients.

[7]NTP - **N**etwork **T**ime **P**rotocol synchronizes the clock of computers in a network.

[8]TALPA - **T**he **A**cronym for **L**ightweight **P**lug-ins **A**rchitecture

Figure 1.2: PariPari official logo.

Another service supplied by the Core is the *logger*, which lets log messages to be written on file. It is useful especially for developers, that can debug their plug-in behavior by logging error messages and using several levels of verbosity.

Lastly, the Core provides a basic and temporary *GUI*, written in Java *Swing*[9], but it is now being replaced by the official PariPari GUI plug-in and its integration will be discussed in Chapter 4.

1.2.2 Credits System

A dedicated module handles the *Credits System* of PariPari, which can be divided in two layers. The former deals with the communications between peers and it is used to encourage participation rather than parasitic behaviors. However this layer still in development pursues also the goal to create a scalable, transitive and cohesive economy among peers [7]. The latter instead is managed by the Core and it is achieved by the exchange of *tokens*. When a plug-in needs some kind of resource from another plug-in, it must pay a certain amount of tokens. In this way, plug-ins requiring the same resource compete and the system determines the price of the resource itself. So, analyzing such prices, plug-ins can choose a strategy that minimizes the expense of tokens.

[9]The primary Java GUI widget toolkit.

1.2.3 Connectivity and NIO

As written before Connectivity manages network activities, allowing plug-ins to write and read data on *TCP*[10] [11] and *UDP*[11] [10] sockets also supplying *HTTP*[12] connections. Here the recent migration to *NIO*[13] APIs - introduced in the version 1.4 of Java - has led to an important step forward, providing new features and improving performance in the areas of buffer management and network scaling.

The APIs defined by Connectivity are: `TCPNonBlockingSocketAPI`, `UDP-BlockingSocketAPI`, `TCPServerNonBlockingSocketAPI` and `URLConnectionAPI`. The first object initializes an outgoing TCP connection, the second is used to send UDP datagrams, the third kind of socket instead is a server socket that accepts incoming TCP connections, finally the last one sets up an HTTP connection. The operations of non blocking sockets are executed asynchronously with a notification being sent back upon completion. When an operation is requested, a `PluginNotification` object that can be used to hold all the necessary references to process the outcome of the operation must be provided. The notification are put in a `BlockingQueue` and it is possible to also define different queues for read, write or connect operations.

The integration of NIO will be discussed in Chapter 4, more detailed information about NIO are referenced in [9].

1.3 Mulo

Mulo belongs to the external plug-ins class and in particular to the file sharing module. In fact Mulo is an *eMule* client, whose main functions are to search, find, download and share files. This thesis is focused on the reengineering and refactoring of Mulo, also achieved by the NIO and GUI integration.

[10]TCP - **T**ransmission **C**ontrol **P**rotocol provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer.

[11]UDP -**U**ser **D**atagram **P**rotocol sends messages, in this case referred to as datagrams, to other hosts without requiring prior communications to set up special transmission channels or data paths.

[12]HTTP - **H**yper**T**ext **T**ransfer **P**rotocol is a networking protocol for distributed, collaborative, hypermedia information systems and it is the foundation of data communication for the web.

[13]NIO - **N**ew **I**nput/**O**utput

Mulo was born about four years ago, in 2007, and a lot of students have worked on it. At the moment of greatest activity the team counted twelve students (both Bachelors and Masters) and about 30.000 lines of source code have been written to develop Mulo.

Figure 1.3: Mulo official logo.

Mulo does not support all the features of the other clients and it does not copy eMule in every step. During its design students are always trying to improve the eMule Protocol and this approach is leading to some important results. Moreover some innovative functionalities, dealt in Chapter 3, are now matter of research, in fact they are not yet supported by any other eMule client.

Thanks to Mulo a PariPari user can participate to two notable P2P networks, described in the next chapter.

Chapter 2

The eMule network

eMule is the combination of two different networks and this chapter deals about them and their protocols.

The older of them is the original *eD2K*[1] network developed by a company, *MetaMachine*, which had possession of the servers managing the network and kept the source code private.

Figure 2.1: eMule official logo.

A lot of new clients have been then developed, until the birth of the eMule [2] project in 2002, an open-source software which aim was to improve the original eD2K client. eMule team has also created an extension to the primary protocol to add new features and functionalities.

In 2005 eDonkey was discontinued due to legal issues with RIAA[2] , but the eD2K network survived as servers were brought up all around the world. In fact interconnections between servers and features added to eMule prevented the network from splitting in a multitude of smaller networks.

[1]eD2K - **e**lectronic **D**onkey **2000**

[2]RIAA - **R**ecording **I**ndustry **A**ssociation of **A**merica

Then eMule introduced a DHT as a second network, *Kad*, mainly to handle the constant tear down of several servers. There actually is a huge gap between this network and the original one: eD2K is a hybrid network, because some operations are server-based, while Kad is completely decentralized.

Nowadays file sharing applications are very popular and eMule is not the most used P2P application anymore. The *BitTorrent* [1] Protocol and its several clients have encountered a huge success, also because it is DHT-based and does not need servers. However eMule still works fine and in some cases it performs better than BitTorrent: a problem of this new network is that shared files have a short life and often a file is no longer available some weeks after it has been shared. Although many people think that eMule is by now dead, the project is still alive and the version (0.50a) is really recent, introducing some new features as well as a new more user-friendly interface.

In the following sections we briefly describe the two eMule networks and their protocols.

2.1 The protocols

All communications in eMule happen via TCP or UDP and a list of packets is defined in order to exchange messages between the peer and the server and among peers.

All these packets have the same header. In the TCP case the header is of 6 bytes and has three fields (see figure 2.2) to indicate protocol, packet length and packet type. The UDP packets header has no length field, so it is only of 2 bytes (as shown in figure 2.3). This means that in TCP transmissions the length field can be used to frame packets, while in the UDP ones the header must be used as a sentinel to frame the received packets through a stream. Once protocol and packet type are known, we have all we need to decode the packets body.

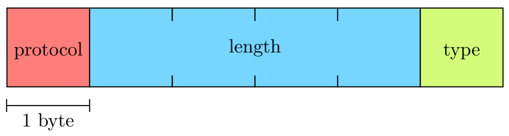


Figure 2.2: TCP packet header.

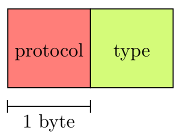


Figure 2.3: UDP packet header.

The *eD2k Protocol*, whose byte of identification is 0xE3, was the first to be created and it has two classes of packets: one for the exchange of messages between the peer and the server and another one for messages among peers. As stated before the *eMule Extension Protocol* (byte of identification 0xC5) was then created to extend the original one, all its packets concern only the exchange of information among peers. The most important packets are sent via TCP, but there are also some UDP packets. UDP is instead essential for the *Kad Protocol*, whose byte of identification is 0xE4, because all its packets are sent via UDP. The reference to a detailed specification of all the packets of those protocols is [19].

Those are the fundamental protocols that an eMule client must implement, even if not all the packets have to be supported to work fine. Beyond those there are other three protocols useful to send compressed data, to obfuscate communications and to support large files, but they will be described in Chapter 3.

Another important element defined in these protocols is the *tag*. A tag is useful to send additional information of different kinds, in fact it can contain a number (short, integer or long), a hash and even a string. Tags are often sent as a list that can change depending on the case.

2.2 The eD2K network

The eDonkey network is hybrid, containing two different entities: peers and servers. Because of the server-based operations it is not totally decentralized and every time a server is down a big portion of the network is lost. When a server is closed all its stored information about files and peers are missing. So the servers are fundamental to join the network, share and search for files and finally ask for sources from which files can be downloaded.

Once we know the file we want to download and the peers sharing it, all the following phases concern only the P2P structure and the server is not necessary anymore (except for callbacks, see section 2.2.4). In the following sections all the details about the eD2K network and its mechanisms will be presented.

2.2.1 Peers identification

Every peer has an identification number, the eD2K ID, which can be *low* or *high*. A low ID means that the peer does not accept incoming connections and it is not reachable from the outside. So while a peer with high ID can be contacted by anyone in the network, it is not the same for a peer with low ID.

In the low case the ID is a casual number lower than 2^{24}, otherwise the high ID is calculated from the *IP*[3] address: given the IP in the form a.b.c.d, where each letter stands for a byte, the ID will be computed using the following formula:

$$ID = a + b \cdot 2^8 + c \cdot 2^{16} + d \cdot 2^{24}$$

2.2.2 Files identification

Obviously also files must be univocally identified in the network in order to let peers share and search for them. Of course they can't be identified with the name, because the same file can be renamed in different ways by different users, sometimes also in a misleading way.

The files identification is achieved thanks to a *hash function*[4] and in the eD2K Protocol the *MD4*[5] [20] algorithm serves this purpose.

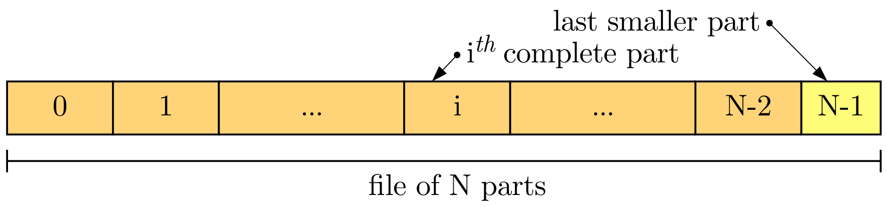


Figure 2.4: File partitioning.

[3]IP - **I**nternet **P**rotocol

[4]A hash function is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an identification number

[5]MD4 - The **M**essage **D**igest **4** algorithm takes as input a message of arbitrary length and produces as output a 128-bit hash.

A file is split into parts of 9.28 MB each, as in figure 2.4, except for the last one that could be smaller. Suppose the file has N parts: first of all the hashes of all its N parts must be calculated, then these hashes are concatenated and the MD4 function is applied again to the concatenation in order to compute the hash of the whole file.

As we will see later in section 3.1.1, these hashes are also used to manage and correct corruptions; this is not the only way to identify a file in the network, in fact a more advanced system has been developed to this purpose.

2.2.3 Server login

A peer must connect to a server to join the network. Several server lists are available on the web and they are kept updated. When a peer contacts a server for login, it receives the eD2k ID. To understand if a peer is high or low, the server, after the login request, acts as a peer and tries to perform an handshake with it. If the peer accepts incoming connections, it will answer, otherwise after a timeout the server assumes that the peer is not reachable from the outside. In this way the server can decide what kind of ID should be given to the peer through the login response.

The main difference between low and high ID is that each server maintains its own list of connected peers, so in two different servers there may exist peers with the same low ID, while, for obvious reasons, there can be just one peer with a fixed high ID for all the servers.

A list of tags is exchanged through the login, allowing the peer to know what features are supported or required by the server, such as compression.

2.2.4 Callback

Two peers with high ID can always connect to each other and a peer with low ID can always contact a peer with high ID. Obviously two peers both with low ID can't communicate, because none of them accepts incoming connections.

The *callback* mechanism allows a peer H with high ID to connect to a peer L with low ID (see figure 2.6), but with the restriction that the two peers must be connected to the same server, which acts as a relay. To contact L, the peer H sends to the server a callback request referred to L, then the server forwards

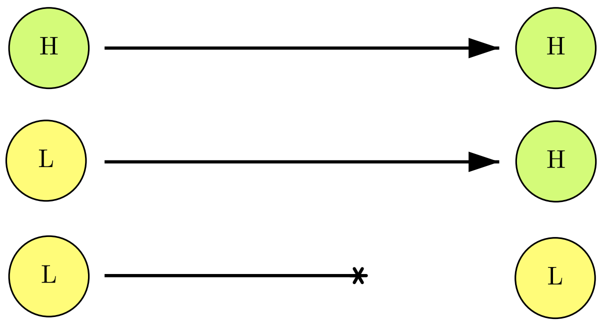


Figure 2.5: Connections between peers with high and/or low IDs.

a callback notify to L and at this point it is up to peer L to open the connection toward H.

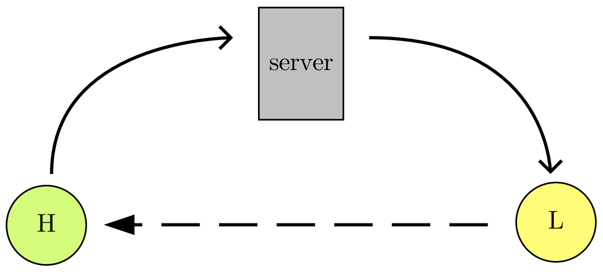


Figure 2.6: Callback mechanism.

2.2.5 Offer files

After the login, a peer must notify the server about its shared files, so that other peers can download those files from it. The offer files process takes place and the peer sends the list of its shared files in a packet with a list of tags for each one. In this list of tags there are some required details, such as name, size and MD4 hash of the file and some optional information such as the format and the type (video, audio, etc.).

The only restriction is that no more than 100 files can be shared in an offer files operation; if the files to be shared are more, then the list will be split into more offer files packets. Every time a peer updates its list of shared files, an offer files operation is necessary in order to inform the server about the changes. In fact the shared list does not contain only complete files, but also those still in download.

2.2.6 Files search

In the eD2k network it is absolutely necessary to contact a server to search for a file and there are two ways to do that: *local* and *global* search. Local search takes place via TCP and a peer must be logged in a server, before performing it. Before transmitting data on a TCP stream, a connection must be established and this is achieved through the login to the server. Global search is instead performed via UDP and it does not require a login, so a request can be sent to all the servers known by a peer. A totally different way to search for files is provided by Kad (see 2.3.8).

Usually a search request contains the *keyword* and the server response for that contains a list of files whose name is related with that keyword. Moreover the response contains a list of tags for each file with all the needed information.

An advanced mode also exists for sending more informations to the server about the searched files through a list of tags (such as minimum or maximum size, type, format, etc.), in this way the results list will be filtered by the server itself.

2.2.7 Sources search

When a peer wants to download a file, it must know who is owning and sharing this file, namely its *sources*. In the eD2k network the only way to learn these information is to ask the servers for them one more time. Sources search can be local or global as well and works exactly as the files search, only with a different kind of information being exchanged. Through local sources search the peer receives all the sources known by the server that is sharing the requested file, both high and low. But when a global search for sources is performed only the ones with high ID are communicated, because the low sources are not connected to the same server as the one to which the peer who sent the request is, so the callback mechanism can't take place.

In a sources search response two basic fields are inserted for each peers in the list: the eD2K ID and the TCP port. So if it is an high ID, the requesting peer gets the IP from it and directly contacts the new source; otherwise, the requesting peer sends a callback notify related to that ID to the common server.

Once the peer knows the list of sources who shared a file, it contacts them and everything that follows concerns only the P2P structure of the eD2K network.

2.2.8 Peers handshake

An eD2K handshake occurs before a peer can download a file from another one. It is achieved by exchanging a *hello* request and response. Through this handshake several basic information about peers are sent in a list of tags. First of all the details about what features they support and keep active are written in the two *miscellaneous options* tags, which will influence all the future phases, so it is of primary importance to decode them correctly. All those features will be described in Chapter 3. Other information included in hello packets are the nickname, the *user hash*[6], the UDP port, etc.

After the handshake there is one more optional phase (see section 3.1.2), then the file request takes place.

2.2.9 File request

The peer now asks for the file, first of all sending a *file request* to be sure that the contacted source is still sharing the file. If not, the source answers with a *file not found* response, otherwise the response contains some details, such as the name that these sources has given to the file. If the source has the file, the process will go on and a *file status request* is sent. The received response contains the *parts bitmap*, which allows to understand what the complete parts owned by the source are.

Some other phases occur depending on what features the two peers support. But before proceeding with the final steps, the peer will ask for the MD4 hashset, if it is not already known, that contains the MD4 hash of all parts. Knowing

[6]A casual MD4 hash that the peer automatically assigns to itself at the first run.

these hashes, the peer will be able to check if the received data for some file are correct.

At this point the peer sends the request to start the download: in the best case the download starts right after, otherwise the peer is put in a queue. The queue is managed by the source with a *FIFO*[7] policy, but a peer can also gain positions (see section 3.1.3). When the peer is put in queue, the connection is closed and it repeats the handshake and all the previously described phases after 5 minutes, in order to know its new position in the queue or to start the download if it is its turn. It may also send a queue ranking request via UDP, which does not require a connection.

2.2.10 Download

Finally the download starts and at least 20 minutes of download are guaranteed before the peer is put in queue again. Also, if the peer loses the connection, it will be able to restart, the download, if it reconnects within the first 20 minutes. The peer can request up to three different blocks of file at once. Data packets contain also the offsets of the sent block, so the peer can correctly save on disk.

When the source stops the download a cancellation message is sent. But the downloading peer can stop the data transfer as well, for example when it receives uncorrected data and decides to *ban* the source.

2.3 The Kad network

The development of Kad in Mulo started about a year ago and it is not yet completed, but all of its main functionalities work fine. For a full review about Kad and its implementation in Mulo the reader is referred to [8], here its overview will be brief.

As we said before, Kad is a DHT and it relies only on peers. Kad is based on the *Kademlia* [18] algorithm, with the only difference that the Kad implementation uses 128-bit long IDs, also said *Int128*, as opposed to the 160-bit ones used in the original Kademlia. This is due to the fact that to map resources - the shared file in this case - in the network the file MD4 hash is used. Because peers and resources must both be mapped on the same address space, an address

[7]FIFO - **F**irst **In** **F**irst **O**ut

space of 128-bit is also used for the peer ID. The peer ID is usually called *Kad ID* and it is chosen randomly by the peer itself at the first run. A 128-bit ID space can allocate up to 2^{128} different objects, a very large number compared to the number of peers in the network, so even choosing a random ID, the probability of ID collision is negligible.

2.3.1 Distance function

Kademlia's best intuition is to use a XOR metric to compute the distance function between two identifiers in the ID space. The distance between two identifiers ID_1 and ID_2 is defined as $\delta(ID_1, ID_2) = ID_2 \oplus ID_2$. The interesting features of this operation are:

- $\delta(ID, ID) = 0$,
- symmetry: $\delta(ID_1, ID_2) = \delta(ID_2, ID_1)$
- triangular property: $\delta(ID_1, ID_2) + \delta(ID_2, ID_3) \geq \delta(ID_1, ID_3)$.

2.3.2 Routing table

The routing table, stored by every peer, manages links between peers across the network, thus defining the network topology. It contains the set of contacts that the peer has knowledge of, maintaining it by adding contacts, deleting the stale ones and keeping a structure - a tree - that makes it efficient when used to access resources spread all over the network. Contacts are stored by their Kad ID and their reliability is evaluated on how long they have been known.

The routing table is made of different *routing zones*, which correspond to nodes in the tree; the tree leaves are also routing zones containing a *routing bin* each. A routing bin corresponds to a bucket, where contacts are stored ordered by the last time they were seen: older contacts are on the top, while newer ones or recently contacted ones are on the bottom. Contacts on the top are periodically contacted to determine if they are still active or they should be removed from the list. Contacts are inserted in the routing table according to their XOR distance from the node's own Kad ID.

2.3.3 Bootstrap

To join the network a peer has to retrieve in some way at least one another peer connected to Kad, thus a *bootstrap* mechanism serves this purpose. Bootstrapping implies to get a list of nodes from somewhere outside the Kad network (most preferably from some trusted web site) and to use this list to populate the routing table. A bootstrap file contains a list of approximately 500-1000 contacts, that are not directly inserted in the routing table; in fact not all the nodes are chosen for the bootstrap, but just the 50 closest to the peer's Kad ID.

Contacts chosen from the most reliable in the routing table are saved at the end of a session for later use in a local file.

2.3.4 Firewall check

Joining the network, a peer needs to find out if it's able to accept incoming TCP connections. In order to do that the peer sends to some known peers a firewall request and waits for the response. Contacted peers try to establish a connection with the requesting node: if the connection is successful, a firewall acknowledgement is sent, and after receiving two of these responses, the requesting node sets its state as not firewalled.

2.3.5 Find buddy

A firewalled peer is not able to accept incoming TCP connections and needs a way to be contacted by other peers. As there is no server in Kad, the callback mechanism relies on *buddies*, non-firewalled peers that act as relays, receiving the callback request for the firewalled peer and forwarding the callback notify to it. Obviously a firewalled peer must before find a buddy between its nearest peers and open a connection toward it.

2.3.6 Lookup

Lookup is a procedure that iteratively locates nodes closer to a target Kad ID: closer nodes have a better chance of being responsible for resources indexed by the target ID. Thus through the lookup we should get a list of nodes close to the resource we want to ask for, so that we may then perform search or publishing actions on those nodes. We may also just want to locate some nodes somewhere in the network, as done in routing table maintenance tasks: in that case only the lookup phase is performed and nodes are added to the routing table.

2.3.7 Publishing

In order to index resources for later retrieval, peers that want to share files and related resources must publish them. In Kad, as there is no server, the publishing scheme is a bit more sophisticated.

A peer must publish the keywords for its own files and publish itself as a source in two different processes. As if it was not enough, it should take care of grouping references to the same keyword to avoid sending many messages to the same contacts for the same keyword.

Publishing of resources does not last forever: contents need to be republished because publishing expires. The expiration time for keywords and notes runs out after 24 hours, while source publishing expires just after 5 hours. A client can index and store up to 50.000 keywords and for each file published a maximum of 1000 sources and 150 notes.

In each kind of publishing a lookup is performed and then, once a list of peers close to the target is known, requests will be sent. Peers receiving the requests will first check if they are entitled to be responsible for such resources, as only contacts in the *tolerance zone* can control a resource. If so, they check other constraints and will eventually index the requested content, replying with a message containing the *load* of the node for that kind of resource. The load is a value between 0 (empty) and 100 (full, that means the request is rejected).

The publishing process terminates when enough nodes have successfully stored the resource. The number of peers that must store the resource is 10; furthermore if the process takes too long, it will end after 140 seconds for keyword and sources publishing, after 100 seconds in the case of notes publishing.

2.3.8 Search

Thanks to Kad it is possible to locate different resources on the network using keywords, sources and *notes* search.

A keyword search is more exactly a search for files published on the network.

The longest word in the keyword is chosen as target by calculating its MD4 hash. Then a lookup phase takes place in order to reach the peers nearest to this target. When they are found, a search keyword request is sent and, if they are responsible for some content published by a third party, they will look for the keyword in its indexed contents and then reply with a list of contents. In keyword search a list of tags that need to match and other restrictions are usually sent, for advanced searches and results filtering.

In the sources search the target is the file ID, that is its MD4 hash. This kind of search works as the previous one, except for the received response: this time it contains the list of contacts sharing the requested file.

Finally as for sources search, notes search is targeted to a specific file, but this time the response contains a list of comments and ratings for the file.

A search terminates when enough results are retrieved (300 for keyword and files search, only 50 for notes one) or if it goes on for too long (the timeout is set to 45 seconds for all kinds of search).

Chapter 3

Features

It is not necessary that an eMule client supports all the eMule features to work fine, however quality and credibility of a client depend on how many more new features are implemented and closely they are adhering to the protocols. The more the functionalities that are supported, the easier the collaboration with other clients; some features are actually mandatory, because without implementing them the flow of messages dictated by the protocol could not proceed. Instead, in some cases it's up to the user to decide whether a feature should be used or not.

Mulo's stage of development is partway, but it is still uncompleted. Those few unsupported features are not indispensable and we intend to add them in the near future.

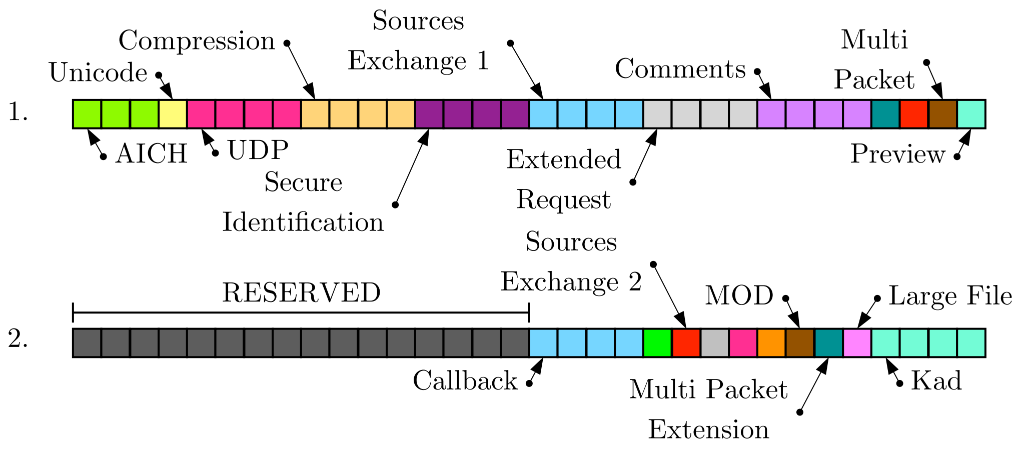


Figure 3.1: Structure of miscellaneous options tags.

A peer understands if another one supports a feature thanks to the miscellaneous options tags (shown in figure 3.1), exchanged in the initial handshake (see section 2.2.8).

3.1 Supported features

Here is a list of all features supported by Mulo. Although it is of vital importance that a peer adheres to the protocols, we sometimes do not strictly follow every part of it. That does not mean we change it, rather we use the exchanged information in a more efficient way, as we will see later.

3.1.1 Corruptions management

During a download, it may happen that incorrect data is received, this could be because of a transmission error on the channel or because the peer is malicious and sends wrong data. There are two ways for a peer to understand if what it is downloading is correct or not.

The first is *ICH*[1], a system that allows the peer to check if a complete downloaded part is correct by comparing the part's MD4 hash retrieved from sources with the one calculated from data saved on disk. Unfortunately this system is not very efficient, because for every little error the peer must download the whole part again.

So a second system was developed by the eMule team, *AICH*[2]. Here every part is split in *chunks* of 180 KB and for every one of them the *SHA1*[3] [17] hash is calculated. These hashes are managed with a tree and put in the leaves. The AICH root hash is the SHA1 hash contained in the root of the tree and recursively calculated starting from the leaves. In fact the hash of an intermediate node is calculated by applying the SHA1 function again to the concatenation of the hashes of its two children. Thus using the AICH root hash is another way to index files and probably this new system will replace the old one in a not so distant future. It is not currently possible to use only AICH, because not all of the eMule clients

[1]ICH - **I**intelligent **C**orruption **H**andling

[2]AICH - **A**dvanced ICH

[3]SHA1 - **S**ecure **H**ash **A**logorithm **1** produces a 160-bit message digest based on principles similar to those used in the design of the MD4 message digest algorithms.

support it.

Before using AICH the peer must retrieve the root hash from sources during the file request operations. A peer can then retrieve the *recovery hashset* of a part, that allows to understand what are the corrupted chunks in a part. In this way if the error is not severe, only few chunks will be downloaded again.

If interested in these algorithms, the reader can refer to [13], instead section 3.3.1 will illustrate how we have improved AICH in Mulo.

3.1.2 Secure identification

To ensure a proper identification on other clients the feature known as *secure identification* is used. It uses a *public-private key* encryption algorithm, that is *RSA*[4] [21]. The secure identification takes place after the initial handshake and it is optional, depending on whether peers support it and keep it active or not. But a peer can also decide to accept only secured peers, so if someone does not use secure identification, the connection will be closed.

Sending the request, peer A states whether it has the public key of peer B. If not, A receives the public key of B, which is then stored in a local file for future sessions, and a random value, usually called *challenge*. The challenge is not stored, because a different one is created for every session. If A wants to identify himself with B, it creates a digital *signature* and sends it to B. This signature is from its *private key*, the public key of B and the random challenge. After receiving the signature of A, B checks if it has been created from its public key and the correct challenge. If it also fits client A's public key, then it is correctly identified.

To better understand how this mechanism works, refer to [23], in the next section we will see the reason why it is so important.

3.1.3 Credits system

A huge problem in file sharing communities is the parasitic approach of some users who only download and don't share anything, so to scourge this behavior eMule has introduced a *credits system*.

[4]RSA - **R**ivest **S**hamir and **A**dleman (the names of who first publicly described it).

When a peer is put in queue, its position is established by its *rank*, as the queue is not managed with a simple FIFO policy. The rank is evaluated on the number of credits owned by the peer, calculated with the following formula

$$\texttt{credits} = min\left\{2 \cdot \frac{\texttt{downloaded\_data}}{\texttt{uploaded\_data}}, \sqrt{\texttt{uploaded\_data} + 2}\right\}$$

In this way the more a peer uploads to us, the more it gets ahead and goes up the queue. Obviously to use credits system peers must support secure identification, to store the amount of credits of other peers in a local file and retrieve it in every new session, avoiding in this way the problem of other peers cheating about their credits. Obviously a peer without secure identification is penalized: in fact all peers are rewarded by the credit system, whether they're supporting and using the credits system or not. eMule's credit system is described in [23] along with many other systems.

3.1.4 Intelligent autologin

A server list usually contains a large group of servers, but some of them are often unreachable or bad ones, thus eMule has implemented a system to automatically perform the login. Mulo sports an *intelligent autologin* system to automatically connect to the best possible server in the list and to prevent fake low IDs.

When a server list is downloaded from the web, usually it does not contain only the server's IP and TCP port, but also the number of users connected to the specific server and the number of files on this server. They are two approximations, however at every new login the server updates and sends this information to clients. So the rank of each server is based on these two values, rewarding good servers or conversely penalizing those that are bad or unreachable. The server list is sorted and Mulo automatically starts to perform login. Mulo attempts to login to some server for at most 3 times, stopping as soon as some login gives it a high ID. If the third login gives another low ID, Mulo assumes that the problem is due to the peer's network configuration and that it is not a fake low ID. More details about how this system is implemented in Mulo are in [5].

3.1.5 Comments and file rating

When users decide to download a file shared in the network, they are not sure of what the file will be until its finalization. Initially only the file name is known, but users can't really rely on it, because everyone could change the name and play the file off as something else.

Comments prevent the download of undesired contents. Every user that is downloading or has shared a file can leave a comment to give more information about it and its quality. For example comments on films often contain an evaluation to its video and audio quality; sometimes it is also useful to simply write that the file is a fake. Moreover a peer can also *rate* files. These informations are retrievable from both eD2k and Kad networks (in the last one comments are called notes as we have seen in section 2.3.8).

However malicious peers could also leave fake comments or give a low rate to a good file: comments and rating are not enough to ensure users about files and they must pay attention in believing or not in what is reported.

3.1.6 Preview

Preview is a more reliable system to ensure the good quality of a file; unfortunately it is not always applicable and surely not before a peer has already downloaded some data.

Mulo supports the preview of several formats such as *AVI*[5], *MP3*[6], *ZIP*, *RAR*[7] and *ISO*[8]. When a long enough block of a MP3 or AVI file is downloaded, a user can automatically start *VLC*[9] and play this portion of file: if it is a film or a song, users will be able to understand whether they downloaded the correct file by watching it and/or listening to it. In the case of archive files the data is compressed and unusable until the download is complete. Archive files also contain some additional information about the stored contents, so a user can know what files are contained in the archive, after downloading the blocks

[5]AVI - **A**udio **V**ideo **I**nterleave is a multimedia container format.

[6]MP3 - It refers to MPEG (**M**oving **P**icture **E**xperts **G**roup) and is a patented digital audio encoding format using a form of lossy data compression.

[7]RAR - **R**oshal **AR**chive

[8]ISO - An **I**nternational **O**rganization for **S**tandardization image is an archive file of an optical disc.

[9]VLC - **V**ideo**L**an **C**lient is a free and open source media player and multimedia framework.

with these information. ZIP, RAR and ISO use different ways to store different information inside the file and it was not easy to understand and implement their algorithms. For more details see their official specifications.

3.1.7 Compression

The *Compressed Protocol* has been introduced to allow the exchange of compressed data. Its byte of identification is 0xD3 for the compression of eD2K packets and 0xE5 for Kad ones.

Compression can be used by servers, when they response to a search request containing a long list of results. Some servers do not even allow the login of peers that do not support compression. Moreover compression is used to send compressed blocks of file during the download, if both the peers support it, reducing the amount of transferred bytes. To see how compression is implemented in Mulo refer to [24].

3.1.8 Large files management

Initially the file size was written as an integer field of 4 bytes: this means that only files smaller than about 4 GB could be shared and downloaded. The limit of 4 GB is too constraining, because nowadays files easily exceed that size. So the eMule team has also introduced the *64-bit Protocol*, in which packets have a file size field of 8 bytes, allowing to share large files of size up to 256 GB.

When the file size is instead inserted in a tag as an optional information there are two different solutions: to send a tag containing a long value of 8 bytes or two integer tags, where one contains the initial 4 bytes of the size and the second the last 4 bytes.

3.1.9 Sources Exchange

Sources exchange is another smart way to retrieve sources for a file in download, asking peers for them. The eMule team has even developed two versions for it, however supporting its first version is enough to know all we need.

3.2 Unsupported features

Here the most important features that Mulo does not support are described. Obviously it is also due to the fact we have not a good documentation about them: this is why there will be not external references in the following sections.

3.2.1 Obfuscation

Illegal actions, such as to share and download files covered by copyright, can be easily found out by simply analyzing the exchanged eD2K or Kad packets with appropriate tools. The *Obfuscation Protocol* has been created in order to avoid the discovery of what the user is downloading by a third party. It uses the *RC4*[10] algorithm to encrypt packets; in this way the transmitted data does not appear as eD2K or Kad packets, but as a pseudorandom stream of bits decodable only by the receiver. RSA is exploited one more time to exchange the keys used to encrypt and decrypt the stream.

Its support is of primary importance, because some servers and peers do not accept unobfuscated communications. Mulo can obfuscate only Kad packets, instead obfuscation in eD2k is more sophisticated and its development is still an open challenge.

3.2.2 Filters

Filters allow a client to check if a peer is malicious or not: peers are directly banned, without performing the connection and spending time trying to download from them. The most famous filter lists are supplied by *Peer Guardian 2* [4], which allows several levels of filtering. The *Level 1* filter is the most used and it should be set as the default option, the *Level 2* filter is instead optional. There also exist other filters but they are experimental or too restrictive, so the web site advises users against their utilization.

Actually we have already developed a good algorithm to use these filter lists, but it is in not integrated in Mulo yet.

[10]RC4 - **R**on's **C**ode **4**

3.2.3 Kad firewall, findbuddy and publishing

As we have said presenting Kad, it is not completed yet: not all of its functionalities are developed or perfectly work.

Firewall is almost done and it will be hopefully soon integrated with find buddy. The publishing system instead already exists and works fine, but it does not respect the protocol. It simply stores resources in some appropriate nodes, but it does not take into account the expiration of the publishing and other specifications. So a complete review must be done: it is vital to have a very good publishing system, because if we introduce a lot of bugged clients, the whole network will lay on the line.

3.3 What Mulo does but eMule can't

Developing and implementing eMule features, we pay attention to two fundamental aspects: are we using them in the right way and how can we improve them? Our team has reached some important results using this approach: in some cases Mulo works even better than eMule itself. Moreover we have introduced some features that other clients haven't.

3.3.1 Proactive corruptions management

As stated before, we have developed a new system to manage corruptions, *PACH*[11]. Here "*proactive*" means that it is used before the part's download is completed. In fact all the hashes of the chunks are requested during the initial phase of the file request. More precisely we ask for just one recovery hashset to every contacted peer. This lets check if the peer has a correct AICH hashset before starting the download: the tree is reconstructed by using the recovery hashset and its AICH root hash is compared with the one retrieved from the network. Moreover it allows to collect all the SHA1 hashes spreading requests to several peers.

The requested blocks have the size of a chunk: every time the download of a chunk is complete, its SHA1 hash is calculated from the data saved on the disk and it is compared with the known one. The strong point of this system is that a corrupted chunk is discovered in no time and it can be downloaded again

[11]PACH - **P**reactive **A**dvanced **C**orruption **H**andling

immediately. When the download is complete, the user does not wait for the check and recovery of corruptions, as in eMule, but the file is ready to be utilized.

3.3.2 Parallel hashing

To hash a file can take a lot of time, particularly if it is a large one: we have implemented a system to calculate hashes concurrently and in parallel. This means that two threads are used to hash every file not to freeze the console and other Mulo's operations and that the MD4 and SHA1 hashes can be calculated at the same time. One part at a time is loaded into memory and the two threads concurrently calculate its MD4 hash and the SHA1 hashes of its chunks and of the part itself.

In this way we cut by half the time spent hashing a file, furthermore we manage and control the loaded data, reducing accesses to the disk.

3.3.3 Intelligent banning

Clients download corrupted data for two reasons: it could be an transmission error due to the channel or data that is deliberately sent wrong by a malicious peer. It is not possible to understand the cause a priori: in the case of a brief error transmission due to the channel, the instantaneous ban of the peer is not the best decision to take. *Intelligent banning* allows to ban peers after they have sent the third corrupted chunk, ensuring not to confuse errors transmission with malicious peers.

3.3.4 Super sources search

In section 2.2.7 we have said that there are two ways to retrieve sources for a file. Obviously a local search gives more results than a global one: for this reason we developed the *super sources search.* It gradually and slowly performs a local search in all the servers in the list. Of course it must also login to every server before asking them for sources, then the connection to the server is kept alive for callback operations. Once Mulo has contacted all peers with low ID related to that server, it disconnects.

Thus Mulo contacts a lot of peers for each file, even more of these retrieved through Kad: it is not surprising that with few active downloads and after some

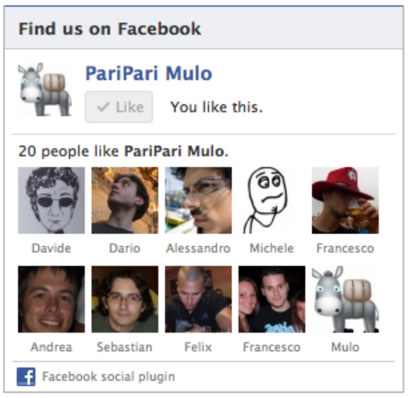


Figure 3.2: “I like!” button for Mulo’s Facebook page.

hours Mulo has contacted several thousands of peers.

3.3.5 Facebook: let the world know us!

Nowadays social networks are very popular and they are excellent way to advertise something: Mulo has a page on *Facebook*, the most popular social network. Thanks to its API [3] it is possible to write on the wall of users or ask them to press the *like button*. All this can be easily done registering as a Facebook developer on the web site and creating a Facebook application, whose assigned ID is requested to use the API.

At the first run of Mulo users are redirected to a web page where they can press the like button for the Facebook page of Mulo, as shown in figure 3.2. After that Mulo asks them if they want to advertise it by posting a note on their wall about them starting to use Mulo. Of course the user must perform the login and give its authorization before Mulo does anything.

Listing 3.1: Source code to post something on Facebook wall.

```
static void postOnWall(String message, String link, String picture) {
   String url = "/dialog/feed?app_id=" + APP_ID;
   url  =+ "&redirect_uri=" + FACEBOOK_PAGE;
   if (message == null && link == null && picture == null) {
      return;
   }
   if (message != null) {
      url += "&message=" + message;
   }
   if (link != null) {
      url += "&link=" + link;
   }
```

```
    if (picture != null) {
        url += "&picture=" + picture;
    }
    openWebPage(FACEBOOK_SITE, url);
}
```

Users might post on their Facebook wall when a download starts for the first time, but if the content is covered by copyright, it will be risky, especially if the wall is public: we suggest users to publish something when its total download or upload speed reaches some threshold.

3.3.6 Subtitles search

Recently an interesting feature has been added to Mulo that allows users to easily search for subtitles. The user can find the subtitles for a specific movie by searching for its title, while, in the case of TV series, it must also provide the season and episode number, of course the user had always to chose the language. Then Mulo searches in some databases available on the web and downloads the requested subtitles or gives the link from which the user can chose between several options.

3.3.7 Automatic management of dynamic server list

Several server lists are available on the web, but some are *dynamic*, that is their link changes every 12 minutes. So it is not possible to insert their links in the source code of Mulo, because they expire after few minutes. On the other hand every time users want to update a dynamic list, they must open the web page where they can find the actual link and manually download the list. To avoid all these operations, Mulo has a *parser* for the web page with the dynamic links and it automatically retrieves the related server lists.

3.4 Experimental features and research

We have introduced some innovative features to improve our application and to make it more suitable than other well-known ones.

3.4.1 Super file sharing

Our main challenge is definitely *super file sharing*, that is a system to combine several P2P networks. For now we are focusing on eMule and BitTorrent, but this system should be adaptable to any other network, already existing or not.

The idea is that, when users search for something, super file sharing automatically extends the search to all the known and available networks, then it arranges and merge the results, before showing them. The real issue is that in many different networks files are indexed in different ways. So it is not possible to know a priori if a result found in a network matches the result given by another one. However the problem can be simplified: results are grouped by file size and format, then the user chooses the file to download and super file sharing tries to retrieve blocks from all the networks by downloading blocks from all the files in the matching group. The downloaded data are then verified with the corruptions manager of the file really chosen by the user. If we retrieve corrupted data downloading blocks of a different file in the matching group, then this file will be discarded. Obviously the system must discover as soon as possible if a file in the groups really matches the chosen one or not.

In a utopian feature the system will show every matching group as a unique result, but for now only an experimental implementation of this system exists and it is described in [24]. Moreover we intend to create our protocol, the *PariPari Protocol*.

3.4.2 Assisted search

On the web there is a lot of information that can be exploited to develop an *assisted search*. For example, by keeping a list of the favorite musical artists, Mulo could automatically notify users when their new songs come out and suggest the download. In the same way there are services on the web that allow to make a adaptable calendar of TV series episodes. Knowing the users' favorite TV series, Mulo could notify them when a new episode comes out or it might automatically start the download and notify them only when the download is finalized. Otherwise Mulo could search the web for the name of old episodes and the user should only decide what episodes to download.

3.4.3 Fake files identification

When users downloads a file, it not always is what they expect: *fake* prevention is a delicate issue. Comment, ratings and previews are powerful tools, but they are not always reliable or available. A good idea is to use the file's *magic number*[12] to identify data it contains. This magic number is already detected and analyzed in MP3 files, but a lot of work is still to be done.

3.4.4 Facebook and friends

eMule allows to favor users just by declaring them as *friends*, so we are thinking about linking this feature to Facebook: knowing Facebook friends of the user, they could be automatically recognized.

Moreover we could notify users when their friends are downloading something (obviously not through the Facebook wall) and suggest those contents to them as well.

3.4.5 Mulo for embedded systems

Nowadays a number of embedded systems provide *SDK*[13] to develop applications for these platforms. Most notably, the *Android* platform can run software written in the Java language. The Android operating system runs on many smartphones and tablet computers with a wide variety of hardware specifications.

The point of export Mulo in such systems is that these devices are powerful enough to run the client and most of the time they are not just connected to the Internet through *3G*[14] networks, but also through more reliable *802.11*[15] networks.

Running a P2P file sharing application can be effective to download small files. Obviously also the rapid discharge of batteries issue must be taken into ac-

[12]It is a constant numerical or text value used to identify a file format or protocol. Detecting such constants in files is a simple and effective way of distinguishing between many file formats and can yield further run-time information.

[13]SDK - **S**oftware **D**evelopment **K**its

[14]3G - **3**rd **G**eneration Mobile Telecommunications is a generation of standards for mobile environment and mobile telecommunication services, including wide-area wireless voice telephone, mobile Internet access, video calls and mobile TV.

[15]The 802.11 family consists of a series of over-the-air modulation techniques that use the same basic protocol to create wireless networks.

count when dealing with these devices. Thanks to the new PariPari GUI plug-in, the interface should instead automatically adapt itself to the embedded system's hardware, so no more work will be necessary on it.

Chapter 4

Mulo reengineering

As we have seen in the previous chapter, after the first phase of *reverse engineering*, Mulo has reached an advanced stage of development and it could easily compete with other eMule clients but for two basic aspects: performances and appearances. We want Mulo to become the next best thing, so in this last period we have focused our attention to NIO and GUI integration, completely *reengineering* Mulo, as we will see in the course of this chapter.

4.1 Mulo before

Before reengineering, Mulo's performances were good, but not enough: its main issue was the waste of *threads*. As we can imagine a P2P application opens a lot of connections and their execution must be independent, concurrent and parallel: connections can't block each other and of course their processes can't be executed serially. Threads allow to write on a socket and wait for something to be read without freezeing what is happening in other connections or other active processes.

For this reason Mulo creates a thread for every new connection. If the entity to be contacted is a peer, its thread will last until the initial operations are performed and it will be put in queue or the download will start: after that peers are managed by their own download. Thus additional threads are requested that take care to download files from sources, one for every active download. If it is a server, the thread will instead stop only when the disconnection occurs. Then some other threads are used to manage the upload, send and receive data on the UDP socket and to accept incoming TCP connections. Finally a thread is started for every new search: this avoids freezing the whole plugin while it is searching, especially in Kad, in which searches may take several minutes. However for a full overview of how Mulo used to work see [19].

Thus the major issue is due to peers, the client usually is connected to one server at any given time, requiring only a thread. The worst case is when a *super sources search* occurs: in these cases a login for every server in the list is performed, however the server list is usually short. In the same way there is a limited number of active downloads at the same time. During a download a lot of peers are instead contacted many times; moreover, when a list of sources is retrieved from a search, all these sources are contacted one after another in few minutes. This means that a huge number of threads is needed and often a lot of them start almost simultaneously.

The introduction of Connectivity NIO allows to considerably reduce the amount of threads in use to a low and constant number as discussed in section 4.3.

The other problem that afflicts not only Mulo but the whole PariPari project was the lack of a nice, comfortable and user-friendly graphic interface. In fact before GUI plug-in's development and its integration, discussed in section 4.4, the only way to use Mulo was through a *CLI*[1].

4.2 Refactoring

A *refactoring* phase has preceded Mulo's reengineering, because the source code had some particular aspects. First of all there existed no subpackages, all the code instead was included in the main package `paripari.mulo`. There were more than one class in the same Java file, leading to files with more than 2000 lines of code. In some cases even a class could reach such a length, because of the presence of many long methods. All of that makes the code really complex in some points. Then almost all the variables had a *protected* visibility: every class could access and modify fields of other classes, also due to the fact that there was only one package. This kind of approach is not really *object-oriented* and it should be avoided. Finally there existed absolutely no interfaces, that would have allowed to encode similarities which the classes of various types share. On the other

[1]CLI - A **C**ommand **L**ine **I**nterface is a mechanism for interacting with a computer operating system or software by typing commands to perform specific tasks.

hand the source code already had a high level of readability and its *Javadoc* was complete and well written.

Package	Description
`connection`	Contains all the classes to create and manage connections over UDP and TCP and include the classes for NIO threads and the one that contacts the sources.
`crypto`	Contains classes that allow to use MD4, MD5 and SHA1 algorithms to calculate hashes, RC4 for obfuscation and RSA for secure identification.
`entities`	Its classes are used to create instances of peers and servers and other related objects.
`extra`	Contains classes to perform extra functionalities such as preview and subtitle search.
`files`	Its classes represent shared files and downloads.
`flow`	Contains classes to manage the communications with peers and servers in an asynchronous way by using a particular software design pattern, described in section 4.3.4.
`gui`	Here there are all the classes to interact with the old console and with the new GUI.
`kad`	Contains all the classes related to the Kad network.
`manager`	Contains classes to set up the threads and objects that allow to manage downloads and uploads.
`protocol`	Here there are classes that implement all the eMule protocols, defining a class for every type of packet and tag.
`search`	Contains all classes needed to perform every kind of search, including Kad ones.

Table 4.1: Mulo's packages organization.

The first thing to be refactored was the visibility of the variables: now they are all *private*, providing two methods to *get* and *set* a variable, that can have a different level of visibility, depending on the circumstances. Sometimes we want objects not to be modifiable, then the *get* method returns a clone. After that the second step consists in splitting files composed of more than one class in order to have a different file for every distinct class; the number of files is quadruplicated and they are now more than two hundreds. Finally, after reorganizing files in several package, the source code presents a more expressive and modular structure. The main package `paripari.mulo` contains all the classes that make up Mulo's core and several subpackages, described in table 4.1.

Finally the last refactoring phase concerned packet management. A new interface `IPacket` (shown in listing 4.1), must now be implemented by every packet. The new and most important method is the one that processes specific packets: when we must to process a packet, we simply have to insert a call to this method. Before reengineering a long *switch-case* statement was instead used everywhere received packets were managed. This solution also allows to write all the operations involving received packets only once and to remove a lot of duplicate code. This method will be moreover essential to implement asynchronous connections in a simple way. Since, when a packet is received, it is no longer necessary to know what kind of packet it is and thus to decide how to handle it. The only thing that remains is to choose what to do next, after receiving and processing a packet. This way of processing packets is even more interesting when the packet is a request: in this case the process ends sending the response to the other peer.

Listing 4.1: `IPacket` interface.

```java
/** Interface that all packets must implement. */
public interface IPacket {

  /**
   * Packet protocol.
   * @return The protocol as a <code>byte</code>.
   */
  byte getProtocol();

  /**
   * Total packet size (in bytes), *excluding* header.
   * @return The size.
   */
  int getSize();

  /**
   * Packet type.
   * @return The type as a <code>byte</code>.
   */
  byte getType();

  /**
   * Transform this packet in a form that can be transmitted over the network.
   * @return Packet content bytes as a <code>ByteBuffer</code>.
   */
  ByteBuffer toBytes();

  /**
   * Process the packet and the information that it contains.
   * @param source The peer or server that sends us this packet.
   * @param data Additional object such as the related <code>Download</code> or
   *       <code>Search</code> needed to process this packet.
   * @return If the process is successful or not.
   */
  boolean process(Object source, Object data);

}
```

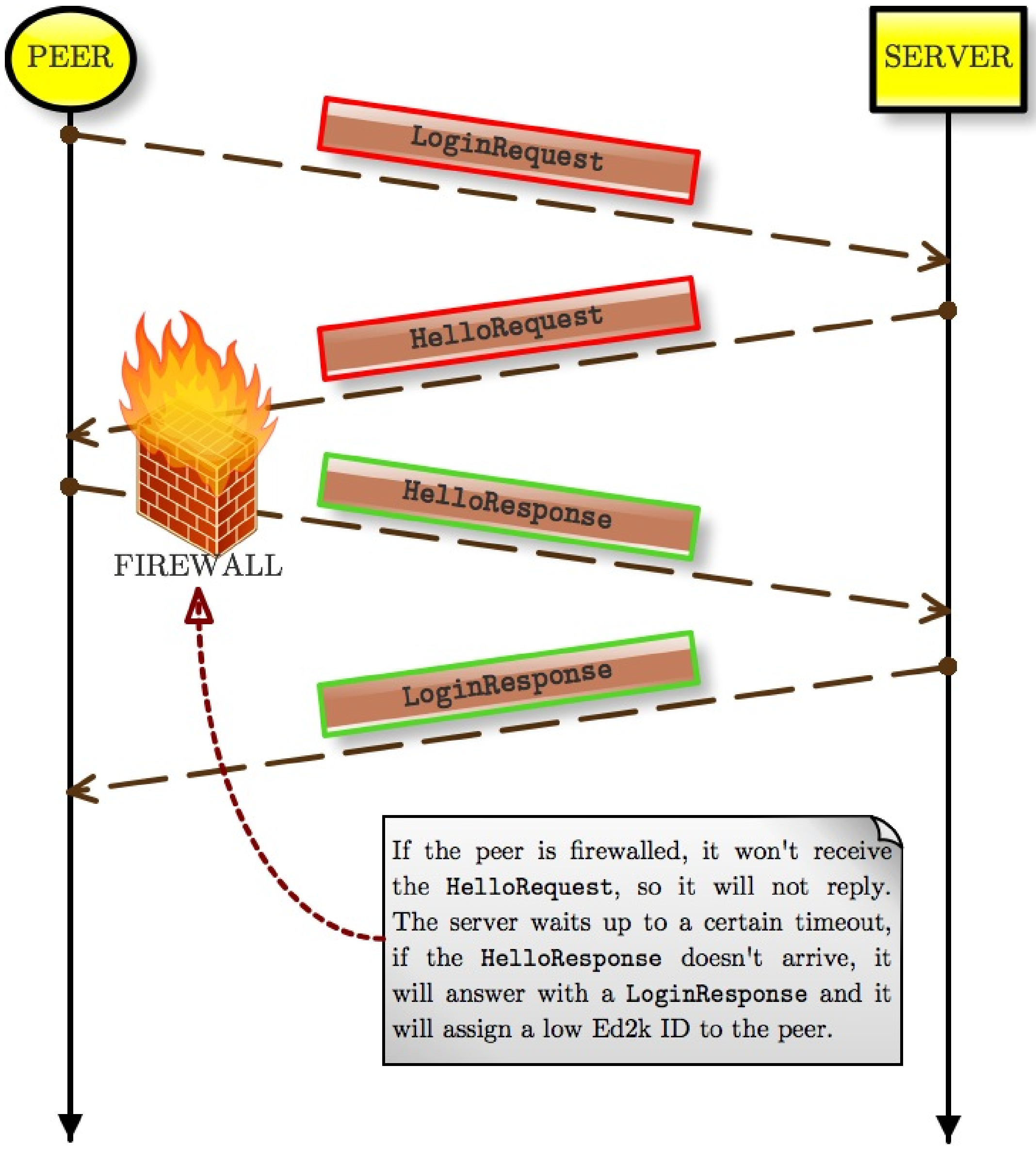
The initial refactoring was longer and more boring than hard. The only thing we had pay attention to is to choose the most appropriate visibility of methods, especially for *set* and *get*. Another important thing was to keep the right references between classes, while we move, rename and change them. However there are powerful tools that help developers useful not only to refactor the code, as we will see in Chapter 5

4.3 NIO integration

The need for reengineering was due to the emergency of NIO integration: the majority of changes in fact sources from the incompatibility between the structure of Mulo and the new Connectivity NIO. Here the heart of the issue was to find an efficient and non blocking way to use TCP sockets. Moreover we have paid attention to choose a clean and elegant solution through the use of a well-known design pattern.

4.3.1 Earlier design

Before we went through a phase of network communication analysis by studying atomic message sending operations.



If the peer is firewalled, it won't receive the `HelloRequest`, so it will not reply. The server waits up to a certain timeout, if the `HelloResponse` doesn't arrive, it will answer with a `LoginResponse` and it will assign a low Ed2k ID to the peer.

Figure 4.1: Login phase.

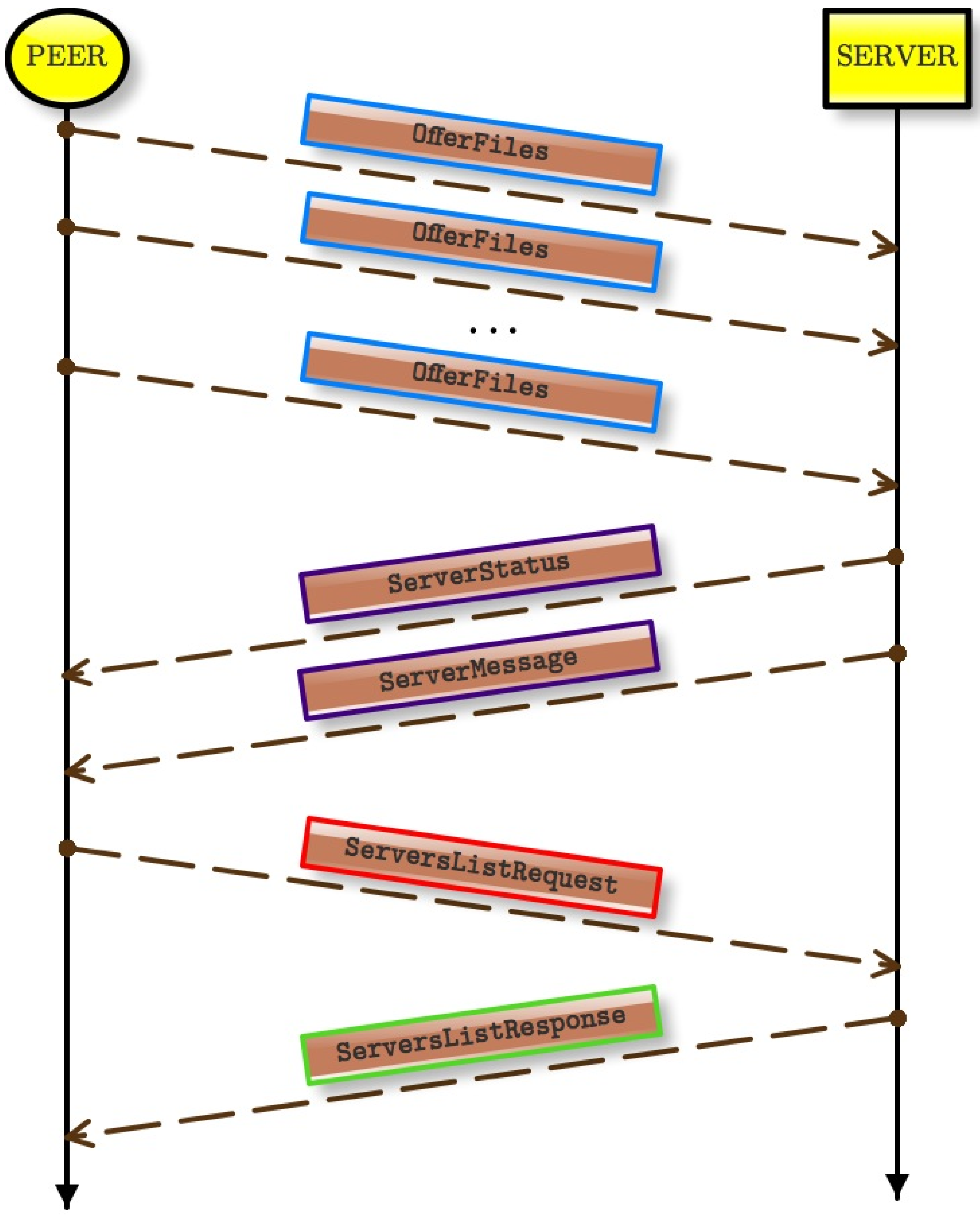


Figure 4.2: Offer files and server list request phases.

Starting with the exchange of packets with a server, we first have the login operations, shown in figure 4.1. A `PacketTCPLoginRequest` is sent and we wait for the respective `PacketTCPLoginResponse`. The check for the ID occurs between the exchange of these two packets: the server acts as a peer, sending us a `PacketTCPHelloRequest`: if we accept the incoming connection and thus we respond with a `PacketTCPHelloResponse`, the server will assign us a high ID.

Then the offer files phase takes place, potentially being executed many times depending on the lenght of list of shared files; every time some files are shared with the server, a `PacketTCPOfferFiles` is sent. In the final phase we ask the server for a list of other servers known by it, adding them in the server list. To do that the peer sends a `PacketTCPServersListRequest` and the server replies with a `PacketTCPServersListResponse`. Moreover, while connected to a server, peers receive periodical status messages from it. After that, several independent and random phases of search for files or sources can take place, until disconnection.

While we are connected to a server we can perform any kind of search. For a local files search over TCP a `PacketTCPSearchRequest` is sent and the results are contained in a number of `PacketTCPSearchResponse`. Instead for a local sources search over TCP we send a `PacketTCPSourcesRequest` and we wait for a `Pack-etTCPSourcesResponse`. If a global search via UDP is needed, the corresponding UDP packets will be used.

The sequence of phases is a little different when the user wants to perform an autologin. In this case there are several phases of login to different servers, until the best server is found; then all the following phases after the login take place, but only for this server. In the same way, when we ask for a super sources search, several phases of login, one for every server in the list, are directly followed by a local sources search, all the other phases are instead skipped.

The sequence of exchanged packets among peers is a bit more sophisticated, but its study has been also more useful. As we have seen in the previous chapter, there are a lot of features that are not always supported or required: thanks to this analysis we now have a clear head on whether a phase must occur and, depending on it, what to do next.

Starting from the handshake, which obviously always occurs, a `PacketTCPHel-loRequest` is sent, waiting for a `PacketTCPHelloResponse`. Then a phase related to secure identification take place, if both peers support it. Here a `PacketTCPSe-`

cureIdentificationRequest is sent and the other peer must answer with a PacketTCPPublicKey (if requested) and a PacketTCPSignature, see figure 5.1. Of course this procedure must be performed by both peers, in order that they can identify each other.

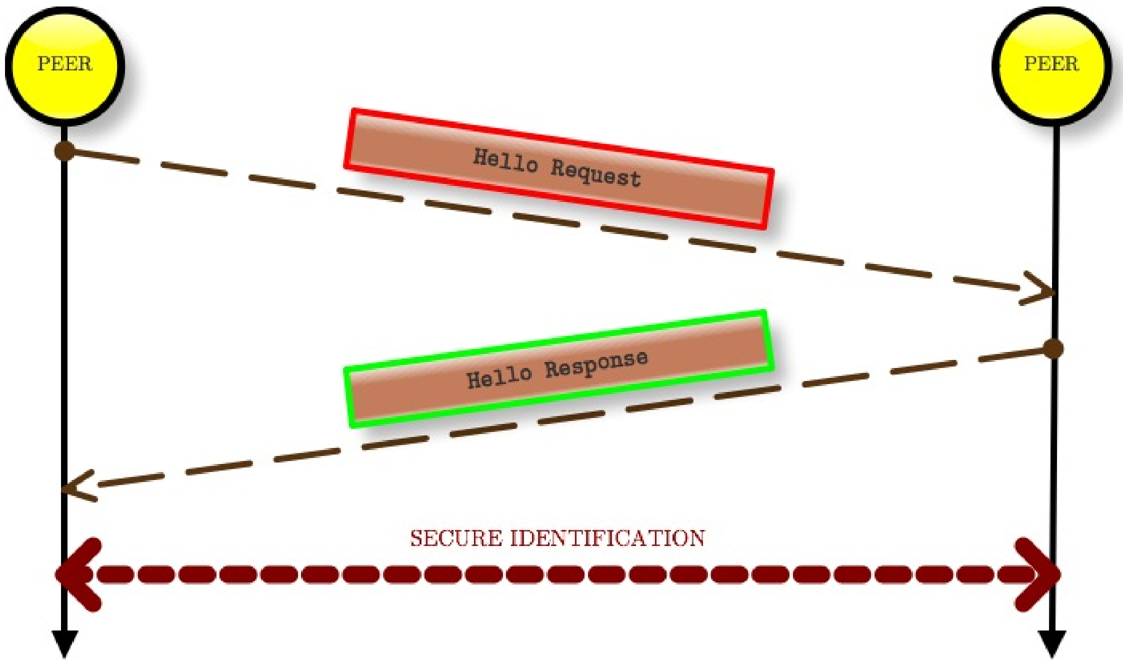


Figure 4.3: Handshake phase.

Then the file request can start and there are two ways to do that. One way is to send the PacketTCPFileRequest and the PacketTCPFileStatusRequest, whose responses are PacketTCPFileResponse and PacketTCPFileStatusResponse respectively. But, if both peers support *multiple packets*, only a single PacketTCP-MultiPacket will be sent that includes both the previous requests and some others. The other requests regard the sources exchange and the AICH root hash: this kind of information can be requested only through multiple packets and they are inserted only if both peers support them. Moreover there is an extended version of the multiple packet, that is the MultiPacketExt: before sending a multiple packet, we must also check if peers support the extended version or not, choosing the correct packet to send. The response is always a PacketTCPMulti-PacketResponse, containing all the responses except for the one related to the sources exchange, which is sent in a separate PacketTCPSourcesExchangeResponse, if and only if the peer knows other sources. Figures 4.5 and 4.6 show

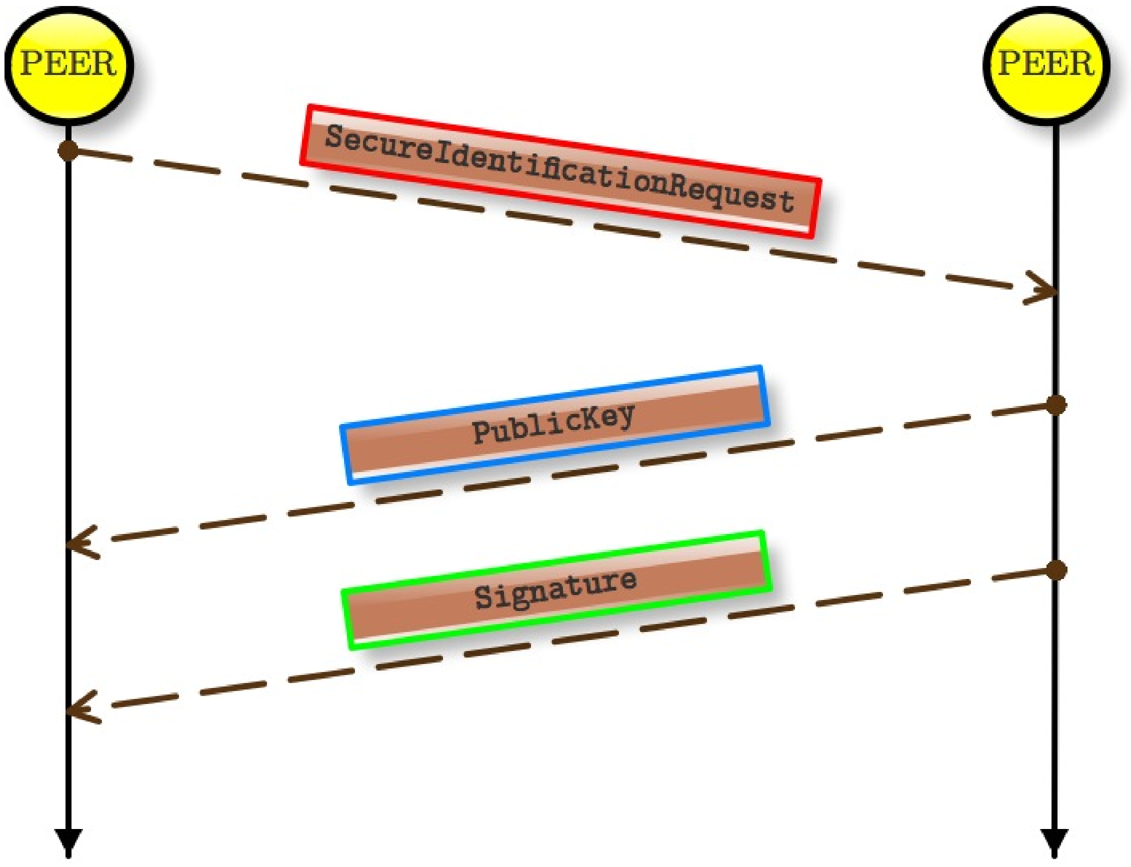


Figure 4.4: Secure identification phase.

the file request phase. If the peer has not the file or does not share it anymore, it will answer with a `PacketTCPFileNotFound`, closing the communication.

If both peers support multiple packets, allowing to exchange the AICH root hash, and they also support AICH, we now ask for the AICH recovery hashset of only one part through a `PacketTCPAICHHashsetRequest`, whose response is the `PacketTCPAICHHashsetResponse`. Then the hashset containing the MD4 hashes of all parts is requested: in this phase a `PacketTCPHashsetRequest` is sent and we wait for a `PacketTCPHashsetResponse` to retrieve the MD4 hashes.

Finally the request to start the download takes place, here we send a `Pack-etTCPStartUploadRequest` and the other peer can reply in two different ways: sending us a `PacketTCPQueueRanking`, that means we are put in queue, or a `PacketTCPUploadAccept`. In the latter case, we start to download asking blocks of the file. Every phase of download starts by sending a `PacketTCPFileDataRe-quest`, that could be also a `PacketTCPFileDataRequest64` if the file is large, end-

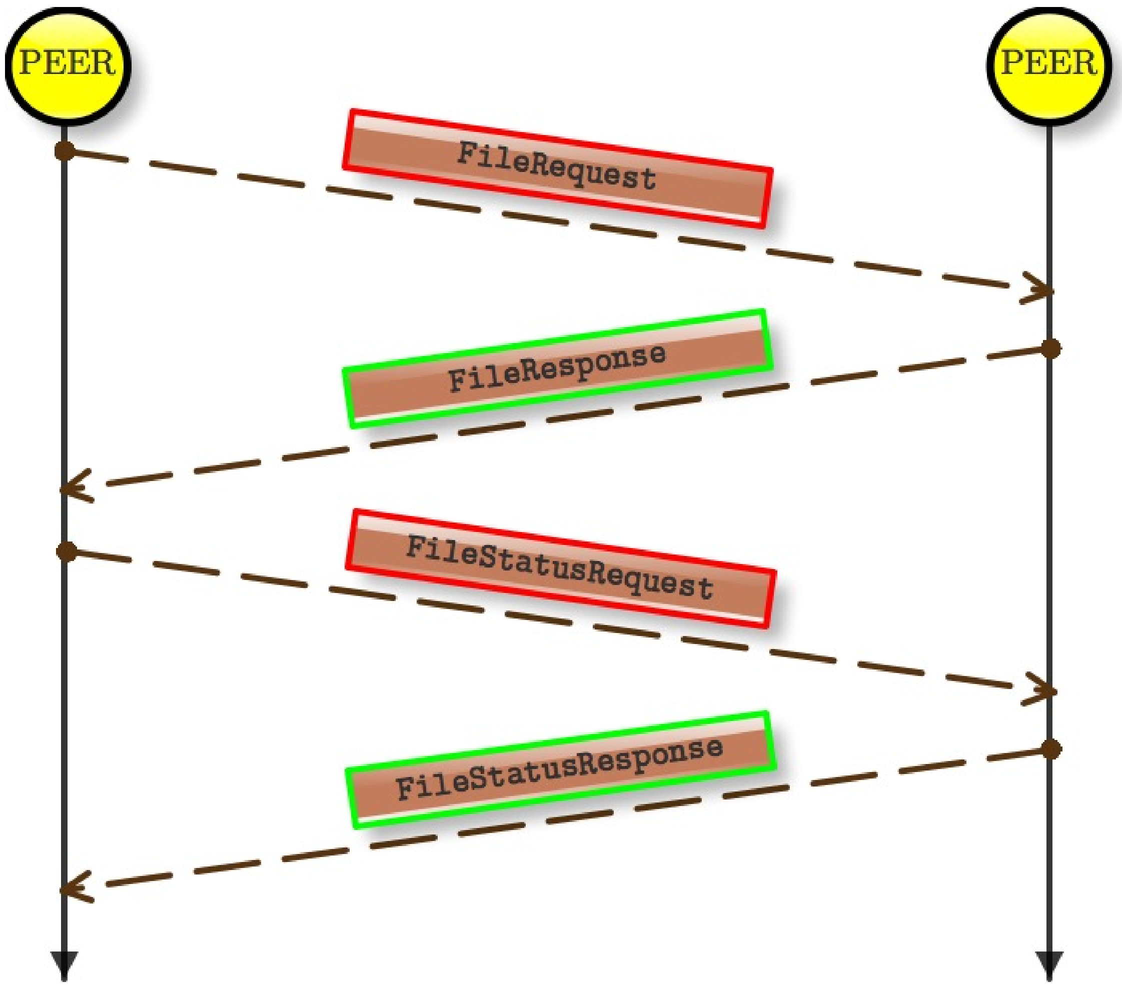


Figure 4.5: Simple file request phase.

ing when all the requested blocks. have been downloaded. Data is sent through several `PacketTCPFileData` or `PacketTCPFileDataCompressed`, if compression is supported, or their 64-bit version, namely `PacketTCPFileData64` and `PacketTCPFileDataCompressed64`.

4.3.2 Threads

While Mulo used to waste a lot of threads, they are now used very carefully, reducing them to a very low and constant number (11). Thus reengineering has produced a very notable improvement: all those threads will now be described in details.

First of all there are five threads needed to put in operation NIO. TCP allows four different kinds of action: *connection*, *write*, *read* and finally *accept* incom-

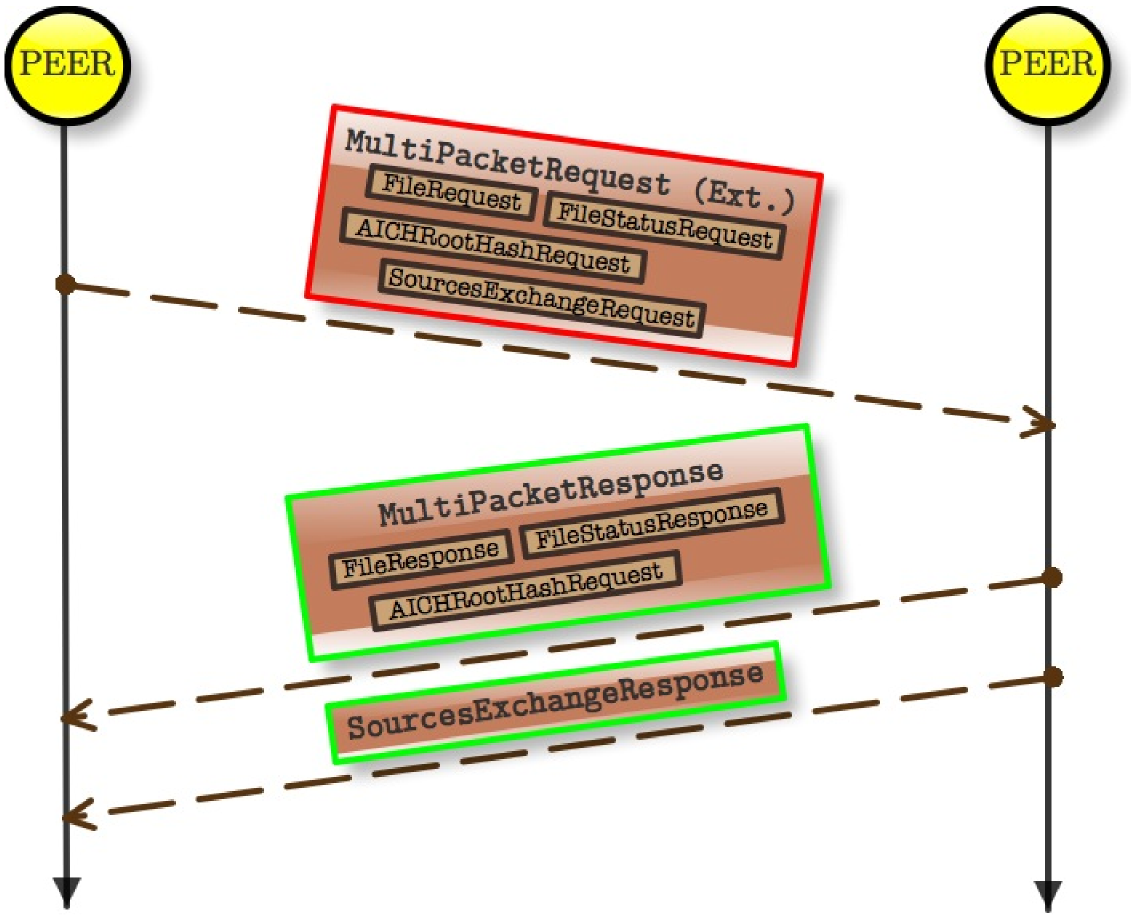


Figure 4.6: File request phase through multiple packet.

ing connection. When we ask Connectivity to perform one of those actions, we must provide a *notification*, which Connectivity NIO inserts in a queue when it executes the request. So we have a thread that removes notifications from the queue and processes them. Obviously if there are no notifications in the queue, the corresponding thread will wait for new ones: the `take()` method of the `BlockingQueue` is blocking and if it does not find notifications in the queue, it does not return until a notification arrives. Connectivity NIO allows us to define a different queue for every kind of action, thus we have four threads, one for each queue, that are `AcceptListener`, `ConnectListener`, `ReadListener` and `WriteListener`. Finally the `ServerListener` thread listens on the server socket for new incoming connections. Because the `accept(...)` method of `TCPServer-SocketNonBlokingAPI` is non-blocking, a lock on the server socket causes this thread to wait until we process an accept notification, avoiding an infinite loop. The process of an `AcceptNotification` invokes a notify on the server socket, unlocking this thread, which calls another accept. `ServerListener` class is shown in listing 4.2, the listing 4.3 instead shows the `go()` method of the other four threads.

Listing 4.2: `ServerListener` class.

```java
/**
 * This class defines the thread that creates a server socket on the configured
 * TCP port and starts a thread waiting on that port for connections.
 * For each accepted connections another thread is started to serve the peer.
 * Static methods to start and stop the listener are provided. The listener can
 * be started multiple times, provided it was previously stopped.
 */
public class ServerListener extends PariPariRunnable {

   /** Unique TCP listener thread, or <code>null</code> if not running. */
   private static PariPariThread thread = null;

   /** The server socket. */
   private static TCPServerNonBlockingSocketAPI serverSocket;

   /** Private constructor so it's not callable from outside. */
   private ServerListener() {
      // Nothing to do!
   }

   /**
    * The name of this thread.
    * @return The name.
    */
   private static String getListenerName() {
      return Mulo.MULO + "-ServerListener";
   }

   /**
    * Opens a server socket on <code>Config.tcpPort</code> port and starts
    * the TCP listener thread, which waits for incoming connection requests from
    * peers and opens new threads to serve each. Does nothing if the server
    * socket is already open or cannot be opened for whatever reason.
    * @return Whether the listener thread was started successfully.
    * @throws IllegalStateException If the listener thread is already running.
    */
   public static synchronized boolean start() throws IllegalStateException {
      if (thread != null) {
         throw new IllegalStateException(getListenerName() + " is already
 running");
      }
      try {
         serverSocket = Mulo.getTCPServerNonBlocking();
         serverSocket.setDefaultQueues(ReadListener.getQueue(),
            WriteListener.getQueue(), AcceptListener.getQueue());
         serverSocket.bind(new InetSocketAddress(Config.getTcpPort()));
    }
    catch (IOException e) {
        PPLog.printError("Exception trying to open TCP port", e);
        return false;
    }
    thread = MuloThread.startThread(new ServerListener(), getListenerName());
    return true;
}

/**
 * Stops the TCP listener thread, if it is running.
 * Note that it may take a few seconds before the thread is actually shut.
 */
public static void stop() {
    if (thread != null) {
        thread.interrupt();
        thread = null;
    }
    if (serverSocket != null) {
        try {
            serverSocket.close();
            serverSocket = null;
        }
        catch (IOException e) {
            PPLog.printError("Error closing " + getListenerName(), e);
        }
    }
}

/**
 * Checks that the listener is up and running.
 * @return Whether the thread is running and the TCP port is open.
 */
public static boolean isRunning() {
    return thread != null;
}

/**
 * This is actually not a public method, because <code>TCPListener</code> has
 * no public constructor. This method waits for new connection requests,
 * processes them and starts a new thread to serve each.
 * Terminates when either the thread is interrupted or the server socket is
 closed.
 */
@Override
@Deprecated
public void go() {
    try {
        while (!this.mustStop()) {
            synchronized (serverSocket) {
                if (serverSocket != null && !serverSocket.isClosed()) {
                    serverSocket.accept(new AcceptNotification());
```

```
               }
            }
            MuloThread.wait(serverSocket);
         }
         PPLog.printNotice(getListenerName() + " thread interrupted, quitting");
         if (thread != null) {
            thread.interrupt();
            thread = null;
         }
         MuloThread.endThread();
      }
      catch (Exception e) {
         PPLog.printError("Exception in " + getListenerName() + " thread,
 autorestarting it", e);
         thread = MuloThread.startThread(new ServerListener(), Mulo.MULO + "-
 TCPListener");
      }
   }

   protected static void notifyServerSocket() {
      MuloThread.notify(serverSocket);
   }

}
```

Listing 4.3: Listeners' `go()` method.

```
@Override
@Deprecated
public void go() {
   try {
      while (!this.mustStop()) {
         queue.take().process();
      }
      PPLog.printNotice(getListenerName() + " thread interrupted, quitting");
      if (thread != null) {
         thread.interrupt();
         thread = null;
      }
      MuloThread.endThread();
   }
   catch (InterruptedException e) {
      PPLog.printError("Interruption in " + getListenerName() + " thread,
 autorestarting it", e);
      thread = MuloThread.startThread(new ReadListener(), getListenerName());
   }
   catch (Exception e) {
      PPLog.printError("Exception in " + getListenerName() + " thread,
 autorestarting it", e);
      thread = MuloThread.startThread(new ReadListener(), getListenerName());
   }
}
```

`UDPListener` is another thread used by Mulo to communicate, waiting for data on the UDP socket and allowing to write datagrams on it. Finally there is the `HandshakeProcessor`: it takes care of contacting sources through a direct handshake, if they have a high ID, or otherwise requesting a callback to the common server. The peers to which we had to connect are managed with three `HashMap`. The former uses as keys the IP address, stored in an `InetAddress` object; it contains the connected peers, the other two instead contain the peer to which we are not yet connected. The not connected peers that have a high ID are stored in a `HashMap` in which the eD2K IDs of the peers are keys. Peers with low IDs are instead stored in a `HashMap` in which servers are the keys and every server is associated with another `HashMap` that contains the related low peers. In this way we can link a low peer with the relative server, allowing to perform the callback.

Other two threads are needed: `DownloadManager` manages the active downloads, `UploadManager` handles peers that are in our queue or to which we are uploading data. Then the `TaskManager` thread performs several periodic operations, such as to save the XML files of Mulo, to update speeds or to check the system status. Finally, the `Kad` thread performs its maintenance tasks.

Obviously spawning search-related threads cannot be avoided, because otherwise the whole plugin would be freezed every time a search takes place, as we said. However these threads live at most for a few seconds (or minutes in Kad); furthermore, users do not usually submit many search requests - not at the same time, at least - and if they do, they do it over time, making it less of an issue.

4.3.3 Notifications

We have four different implementations of the Connectivity NIO API `PluginNotification`, one for every different TCP action. The `ConnectNotification` is processed by the `ConnectListener` thread every time a new connection is opened. At the same way an `AcceptNotification` (processed by the `AcceptListener` thread) arrives when we accept an incoming connection. The incoming connections are detected by the `ServerListener` thread, which listens on the server socket for them. It creates an `AcceptNotification` to insert in the `accept(PluginNotification notification)` method every time it detects a new incoming connection. `ReadNotification` and `WriteNotification` are received every time Connectivity NIO performs a read or write action on a socket, respectively. All these objects implement the `PluginNotification` interface, allowing us to define our own implementation of the `process()` method adding whatever field we may need to the notification itself.

All the notifications (except for the accept one) need to know the related *entity* - that could be a peer or a server - to be correctly processed. We have thus created the `Entity` class, extended by `Peer` and `Server`. This class has a boolean variable that allows to understand if we are waiting for a write notification or not. When we ask for a write action, the `canWrite` variable of the entity is changed to `false` and, when we receive the notification, it is turned to `true`. Every time a write notification is received, we call the method `canWrite()` on the corresponding entity. If we are not waiting for another write notification and we must send some packets, they will be sent, otherwise they are stored in a list and they will be sent as soon as possible. In fact every time we receive a write notification, we call this method to send potentially waiting packets. The case of the read action is easier: the only thing to do is to always wait for new data. So every time we receive a read notification we ask to read again, calling the method `canRead()` of `Entity`. Of course these two methods that allow to read and eventually write on a socket are also called inside the process method of the `AcceptNotification` and `ConnectNotification` to start the exchange of data.

To be able to decide how to implement the `process()` method is of primary importance, allowing us to decode the received data as eMule packets and to decide what to do after NIO has processed a request. In the following listing 4.4 it is shown an example of how we have implemented this method in `ReadNotification`.

Listing 4.4: `ReadNotification`'s `process()` method.

```java
public boolean process(AsynchronousNotification<TCPNonBlockingSocketAPI>
    parameters) {
   if (parameters instanceof AsynchronousReadNotification){
      AsynchronousReadNotification<TCPNonBlockingSocketAPI> readParameters =
         ( (AsynchronousReadNotification<TCPNonBlockingSocketAPI>)parameters );
         if (readParameters.getException() != null) {
            Exception e = readParameters.getException();
            PPLog.printError("Exception in " + this.entity.toString(), e);
            this.entity.disconnect();
            return false;
         }
         if (readParameters.endOfStream() || !this.entity.isConnected() ) {
            return false;
         }
         this.entity.canRead();
         byte[] data = readParameters.getData();
         ByteBuffer cloneBuffer = null;
         ByteBuffer buffer = this.entity.getConnection().getBuffer();
         if (buffer != null) { // this is a clone!
            cloneBuffer = this.entity.getConnection().getBuffer().duplicate();
         }
         if (cloneBuffer == null) {
            cloneBuffer = ByteBuffer.allocate(data.length);
            cloneBuffer.order(ByteOrder.LITTLE_ENDIAN);
            cloneBuffer.put(data);
         }
         else {
            byte[] oldData = cloneBuffer.array();
            cloneBuffer = ByteBuffer.allocate(oldData.length + data.length);
            cloneBuffer.order(ByteOrder.LITTLE_ENDIAN);
            cloneBuffer.put(oldData).put(data);
         }
         if (Packet.isObfuscated(cloneBuffer.get(0))) {
            PPLog.printWarning("Mulo don't support Obfuscation, disconnecting
               from " + this.entity);
            this.entity.disconnect();
            return false;
         }
         while (cloneBuffer.capacity() >= PacketTCP.HEADER_SIZE) {
            int packetLength = PacketTCP.HEADER_SIZE -1 + cloneBuffer.getInt(1);
            if (cloneBuffer.capacity() >= packetLength) {
               byte[] bytesToDecode = Utils.arrayCopy(cloneBuffer.array(), 0,
                  packetLength);
               Peer src = this.entity instanceof Peer? (Peer)this.entity:null;
               PacketTCP packet = PacketTCP.decodePacket(bytesToDecode, src);
               if (packet != null) {
                  PPLog.printIncoming("Received " +
                     packet.getClass().getSimpleName() +
                     " (0x" + Utils.byteToHex(bytesToDecode[5]) + ")",
                     packet, bytesToDecode, this.entity.toString());
                  this.entity.getConnection().setInboundBytes(packetLength);
                  Mulo.setTcpBytesIn(packetLength);
                  Mulo.incrementTcpPacketsIn(1);
                  if (packet instanceof IPacket) {
                     this.entity.getContext().proceed((IPacket)packet);
                  }
               }
               int remainingBytes = cloneBuffer.capacity() - packetLength;
               if (remainingBytes > 0) {
                  ByteBuffer tempBuffer = ByteBuffer.allocate(remainingBytes);
                  tempBuffer.order(ByteOrder.LITTLE_ENDIAN);
                  tempBuffer.put(cloneBuffer.array(), packetLength,
                     remainingBytes);
                  cloneBuffer = tempBuffer;
```

```
                }
                else {
                    cloneBuffer= null;
                    break;
                }
            }
            else {
                break;
            }
        }
        this.entity.getConnection().setBuffer(cloneBuffer);
        return true;
    }
    PPLog.printError("Wrong read notification: " + this.entity);
    this.entity.canRead();
    return false;
}
```

4.3.4 States Pattern

Now that Mulo has asynchronous TCP connections, we need to know in what state an entity is when a packet is received. As we said, the only thing to do after receiving and processing a packet is to decide the following step. A very elegant way to do that is to use the *State Pattern*[2], that is a *behavioral*[3] software design pattern. This pattern is used in computer programming to represent the state of an object and it is a clean way for an object to change its state at runtime.

This pattern requires the definition of several different states linked among them. Every new state must extend the `State` class which implements the `IState` interface. When we have an `IState` object, we can call its method to process a received packet and to decide the next thing to do, changing the related entity's state. In fact every entity has a `StateContext` object, whose code is in listing 4.5. Its main variable, an `IState` object named exactly `state`, represents the current state of the corresponding entity. The `process(IPacket packet)` method of the `StateContext` object is called every time a packet is received, processing the `ReadNotification`. This method calls in turn the `process(IPacket packet)` method of `sState`. The class `State` has a general implementation of this method, but every object that extends this class can override, changing how the received

[2]It is also known as the *objects for states pattern*.

[3]Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns.

packets are managed, while the entity is in this particular condition. Listing 4.6 shows the implementation of this method. Every state stores the request to send and a list of answers to receive. Until the packet is not sent and the list of the corresponding answers is empty, the state cannot be changed. When the request is sent, the variable is set to `null`, an answer is removed from the list when it is received. Finally the state context keeps a list of packets to be sent, usually the answers to the received requests.

Listing 4.5: `StateContext` class.

```java
public class StateContext {

   /** The actual state. */
   private IState state;

   /** List of answers to send. */
   private final LinkedList<IPacket> answersToSend = new LinkedList<IPacket>();

   /**
    * Initialize the object, setting the state to <code>null</code>.
    */
   public StateContext() {
      this.state = null;
   }

   /**
    * When a packet is received, the state is processed.
    * @param The received packet.
    */
   public void process(IPacket packet) {
      this.getState().proceed(packet);
   }

   /**
    * Change the state.
    * @param state The next state.
    */
   public void switchContext(IState state) {
      this.state = state;
      if (this.state != null) {
         this.state.getEntity().canWrite();
      }
   }

   ...

}
```

Listing 4.6: `State` class.

```java
public class State implements IState {

   /**
    * The corresponding entity (could be a <code>Peer</code>
    * or a <code>Server</code>).
    */
   private Entity entity;

   /** The request to send. */
   private IPacket requestToSend;

   /** The list of the expected responses. */
   private final LinkedList<Class> expectedResponses = new LinkedList<Class>();

   /** If the state can be processed or not. */
   private boolean proceed = true;

   /**
    * Initialize the object.
    * @param entity The corresponding entity.
    */
   public State(Entity entity) {
      this.setEntity(entity);
   }

   /* (non-Javadoc)
   * @see paripari.mulo.flow.IState#proceed(paripari.mulo.protocol.IPacket)
   */
   public void process(IPacket packet) {
      boolean processed = false;
      if (packet != null && !packet.isMalformed()) {
         if (this.isWaitingForResponses()) {
            if (this.expectedResponses.contains(packet.getClass())) {
               this.expectedResponses.remove(packet.getClass());
               this.process(packet);
               processed = true;
            }
         }
         if (!processed) {
            packet.process(this.getEntity(), null);
         }
      }
      if (this.proceed) {
         if (!this.isWaitingForSend() && !this.isWaitingForResponses()) {
            this.setProceed(false);
            this.entity.getContext().switchContext(this.next());
         }
      }
   }

   /* (non-Javadoc)
```

```
    * @see paripari.mulo.flow.IState#proceed(paripari.mulo.protocol.IPacket)
    */
    public IState next() {
        return new IdleState(this.entity);
    }

    ...

}
```

Another important method of the `State` class is `next()`: it is also overridden and redefined in every object that extends this class. This method contains all the logic needed to decide what the next execution step will be or, in other words, what next state will be set. It is called when the state is processed, that is if the request has been sent and there are no more answers to wait for, allowing to easily switch the context. Every time the context is switched and every time a request is set or an answer is added to the list, the `canWrite()` method of `Entitiy` is called to try to send them.

Once the main elements are ready, the last thing to do is to define what states are needed and write code for them.: the earlier design phase gives us a clear idea on what are the states and how to implement them.

The first class of states concerns servers. Starting at the login operation we can define 3 different states. `LoginState` is the more simplest state, which performs a login to the associated server. n case the login operation succeeded, the workflow proceeds to an `OfferFilesState` or `ServersListState`, depending on whether users are sharing files or not. When these operations are completed, the context is switched to an `IdleState`, meaning that the connection is still open and that the server is waiting for search request. So if we must perform a files search, the context is switched to a `FilesSearchState`. After the search is finished, the context is switched to an `IdleState` again. If we must perform an autologin, we have the special `AutoLoginState` that performs an autologin: if its server results to be the best the process will continue with the `OfferFilesState`, otherwise there will be another `AutoLoginState` corresponding to the next best server in the user list. Finally if we have to do a super sources search a `Login-ForSourcesState` is initialized, that jumps directly to a `SourcesSearchState`. The complete server's flow of states is shown in figure 4.7.

The flow of states of a peer starts with the `HelloState` that performs the handshake. The next step is to perform the secure identification, but if it is not supported, the `SecureIdentificationState` will be skipped. Here we have two options: to send the file request and the file status request inside a multiple packet through a `FileInfoMPState` or to send them in two distinct phases through the `FileInfo1State` and `FileInfo2State`. If multiple packets are used, then the flow passes through the `AICHState`, if it is supported, otherwise it directly jumps to the `ICHState` to retrieve the MD4 hashes of all the parts. Finally we have the `StartUploadState`: if we are put in queue, the connection is closed and the peer will be contacted again some time later to get an update of our position in the queue or to start the download. Finally, when the download starts, we have a number of `DownloadState`s. The peer's flow of states is depicted in figure 4.8.

There is one more state to manage incoming connections, that is `Incoming-HandshakeState`. When Connectivity NIO notifies an accept and we process it, we create a new entity whose context is set to this state. It has no request to send, waiting for an hello request from the other peers. After receiveing the hello request we have all the information, such as IP and eD2K ID, to understand if it is the response to an our callback. In fact if we find this peer in our list of low peers and we have a pending callback for it, after the `SecureIdentificationState`, we can start to request the file with the usual process. If it is instead a high peer or a peer with low ID that we are not trying to contact, then it is the one that wants to download: after the `SecureIdentificationState`, this peer context is set to an `IdleState`, allowing to wait for requests, to process them and to send the corresponding answer. The `UploadManager` keeps the queue with these peers and decides when to put a peer in queue and when to let it download.

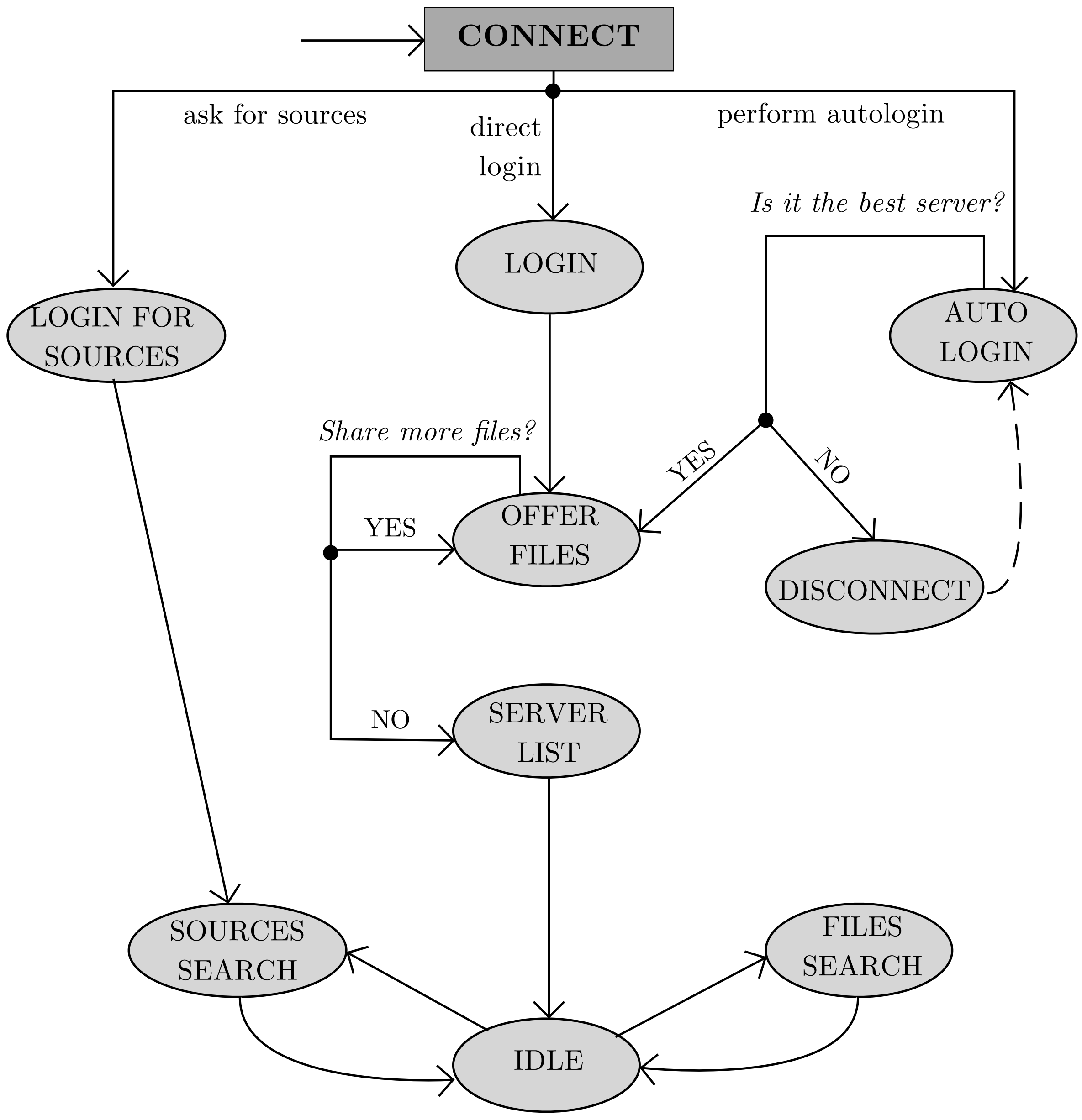


Figure 4.7: Server's flow of states.

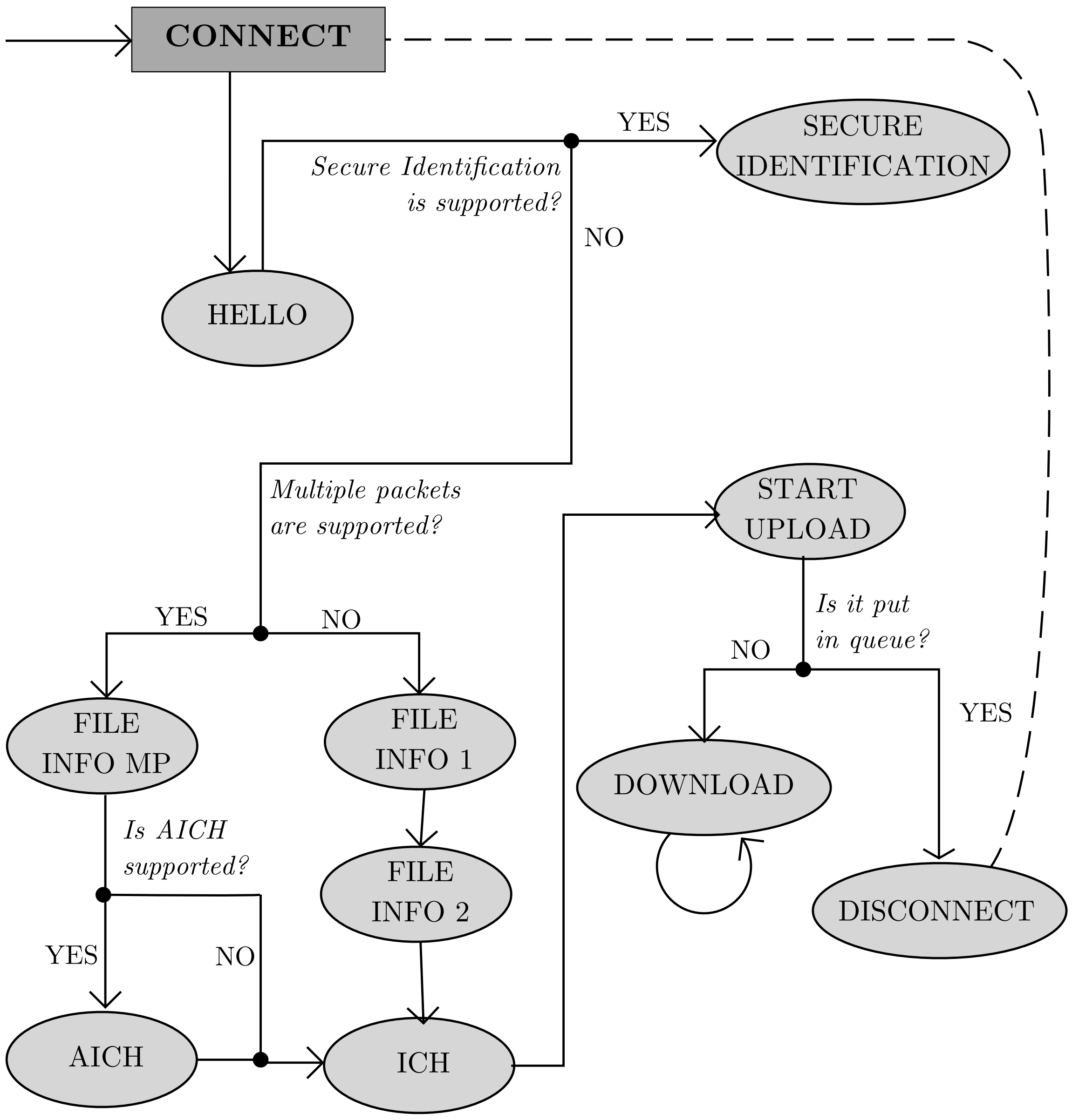


Figure 4.8: Peer's states flow.

4.3.5 Performances

Many tests have been done to show the final result and to prove the great improvement we have obtained reengineering Mulo: in this section two of them will be presented.

In the former we have downloaded a file of few MB: an MP3 audio file of about 7 MB, that is the average size of a song. It took Mulo just three minutes to download the whole file, during which Mulo contacted about 150 sources downloading from 15 of them. The average speed was about 40 MB/s, reaching the peak of 160 MB/s. During all the downloads the number of used threads was steady and the CPU usage, shown in figure 4.9, resulted absolutely negligible, except when PariPari is loaded and when the download is finalized.

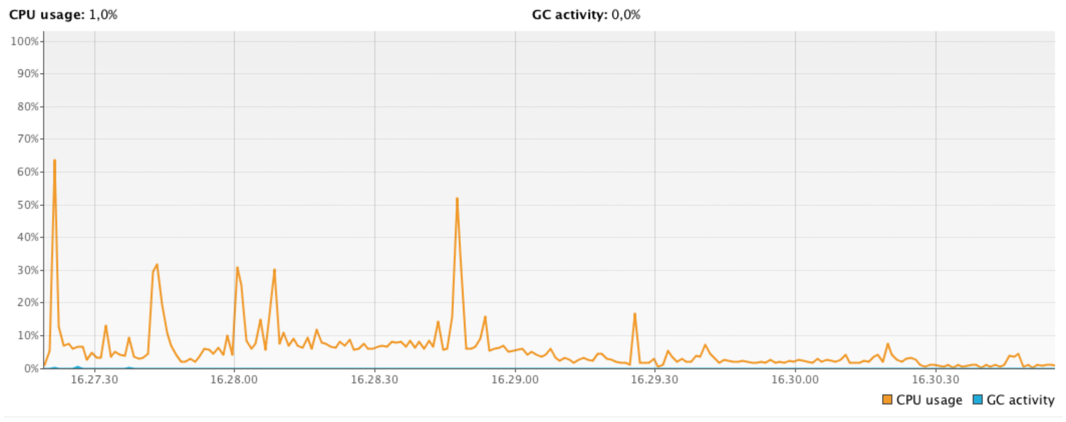


Figure 4.9: CPU usage in the first test.

The latter instead concerns the download a bigger file: an AVI video file, whose size is about 750 MB. Of course this time the download took a bit more, about 45 minutes, but during all this time Mulo has gone as far as retrieving and contacting more than a thousand of sources, downloading from about 25 different peers. The download of the whole file was very fast: the average speed was about 300 MB/s and, when all the 25 sources were active, it has reached a download bandwidth just shy of 500 MB/s.

Running the same tests on the old Mulo, we can see in figure 4.10 the excessive waste of threads. When Mulo is started the amount of alive threads is around 60. As the number of contacted peers increases, the alive threads grow up until Mulo reaches the threshold imposed by the Core of a maximum of 100 threads. At this point Mulo must wait that the number of threads decreases to start a new thread and try to connect to another peer. This means that now Mulo can start the download sooner, contacting peers more quickly.

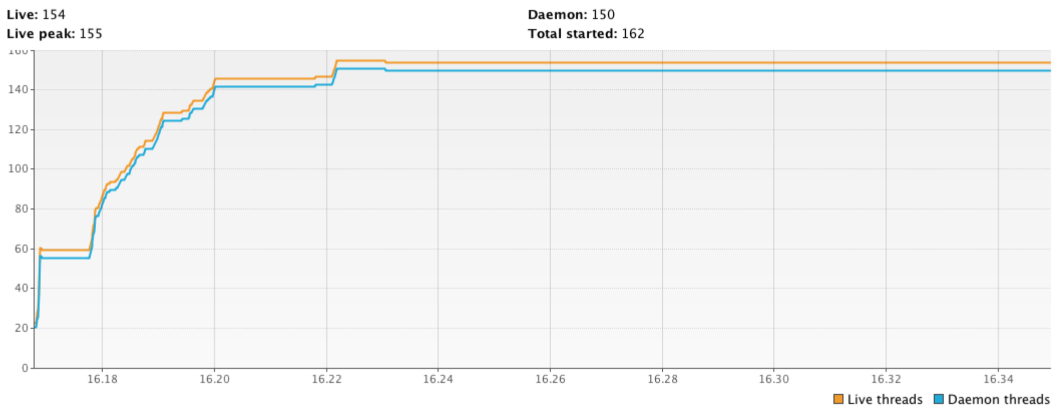


Figure 4.10: Waste of threads before reengineering.

Surely the CPU usage is also improved: before reengineering the average percentage of CPU usage was around 10%, while now it is a bit lower, as we see comparing figures 4.11 with 4.12. But in this test the Java heap memory usage is also interesting, it is absolutely reasonable considering the file size and the amount of peers, as we see in figure 4.13.

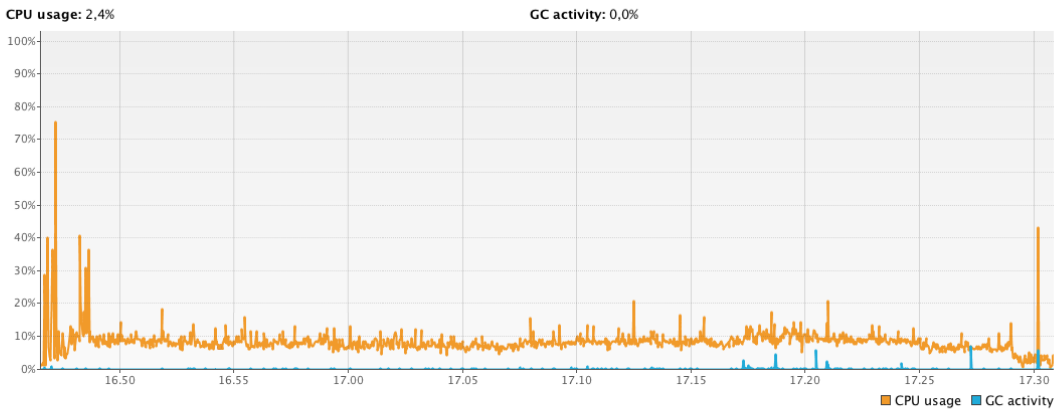


Figure 4.11: CPU usage before reengineering.

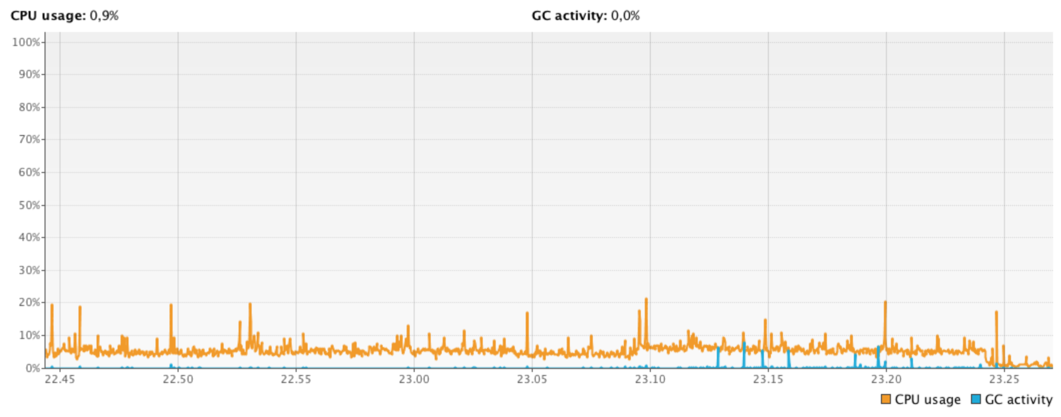


Figure 4.12: CPU usage in the second test.

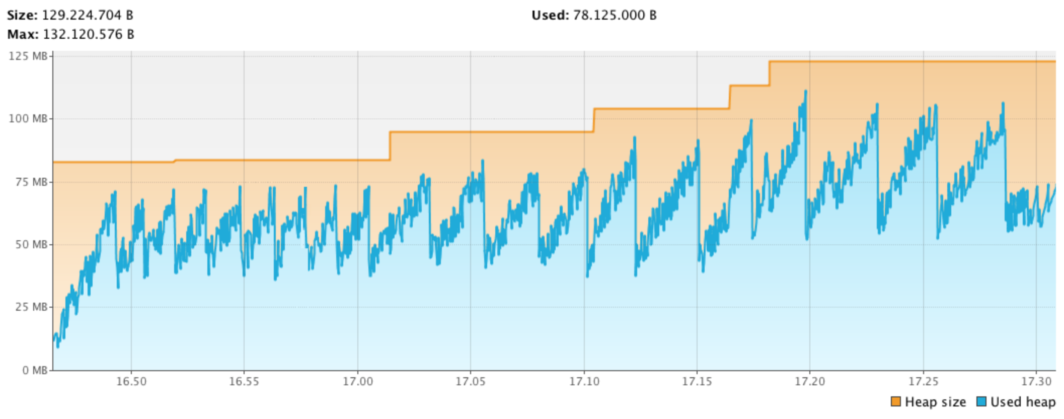


Figure 4.13: Java heap memory usage in the second test.

4.4 GUI integration

The PariPari GUI is developed in *Vaadin*, that is a Java framework for *RIA*[4], web based applications with the same potential of the desktop ones. It features a robust server-side architecture: the largest part of the application logic runs securely on the server. The code is written in Java, then *GWT*[5] automatically translates it in *AJAX*[6] to create the web page. Moreover it is *browser-independent*, preserving the portability of PariPari.

This plug-in is still a work in progress: Mulo is the first to adopt the new official GUI, thus its integration is a testing ground for this new interface. The GUI plug-in is dealt in [14] and [22], here only its integration will be described.

[4]RIA - **R**ich **I**nternet **A**pplication
[5]GWT - **G**oogle **W**eb **T**oolkit
[6]AJAX - **A**synchronous **J**avaScript **A**nd **X**ML)

Listing 4.7: Method to request a `GUIAPI` object.

```java
/**
* Requests and implementation of GUIAPI. The eventual retrieved GUIAPI
* implementation will be saved in <code>MuloGUI</code>.
* @return The GUI.
*/
public static GUIAPI getGUI() {
   try {
      IFeatureValue[] features = new IFeatureValue[] {new FeatureValue("time",
         Mulo.RESOURCE_TIMEOUT)};
      IRequest request = new Request(GUIAPI.class,
         new ConstructorParameters(features));
      IReply replay = Mulo.reference.askTheCore(request);
      PPLog.printError(replay.status().getClass().getSimpleName());
      if (replay != null && replay.status() == IMessage.Status.OK) {
         PPLog.printNotice("GUI started.");
         return (GUIAPI)replay.getAPIs()[0];
      }
      PPLog.printWarning("Could not obtain the GUI.");
      return null;
   }
   catch (FeaturesTooRestrictiveException e) {
      PPLog.printWarning("Could not obtain the GUI.", e);
      return null;
   }
   catch (NotEnoughCreditsException e) {
      PPLog.printWarning("Could not obtain the GUI.", e);
      return null;
   }
   catch (CannotFindRecipientException e) {
      PPLog.printWarning("Could not obtain the GUI.", e);
      return null;
   }
   catch (UnsatisfiedRequestException e) {
      PPLog.printWarning("Could not obtain the GUI.", e);
      return null;
   }
}
```

Mulo must ask the Core for an object `GUIAPI` to use the GUI , as shown in listing 4.7: the GUI is an inner plug-in that offers a graphic interface as service. Then we need a class, called `MuloGUI`, that implements some interfaces defined inside the GUI code: they are `FileManager`, `Searchable`, `Downloadable` and `Uploadable`. First of all we must subscribe Mulo for these actions, notifying the GUI that Mulo can search, download and upload, as we see in the method `start(Plugin)` of `MuloGUI`. The subscription of Mulo to the GUI is done using a thread which dies just after: in this way Mulo does not let the plug-in loading timeout expire while it is waiting for the GUI.

When users interact with PariPari GUI performing an action, such as a search for example, the GUI will call the method that is implemented in `MuloGUI`. In listing 4.8 the code to perform a search is also shown: as we see we must implement the method `search(String query)` of `Searchable`. The `query` is the keyword of the search: users cannot decide what kind of search to perform, thus the search is always local over TCP in the connected server and through Kad. For the time being it is not even possible to perform an advanced search.

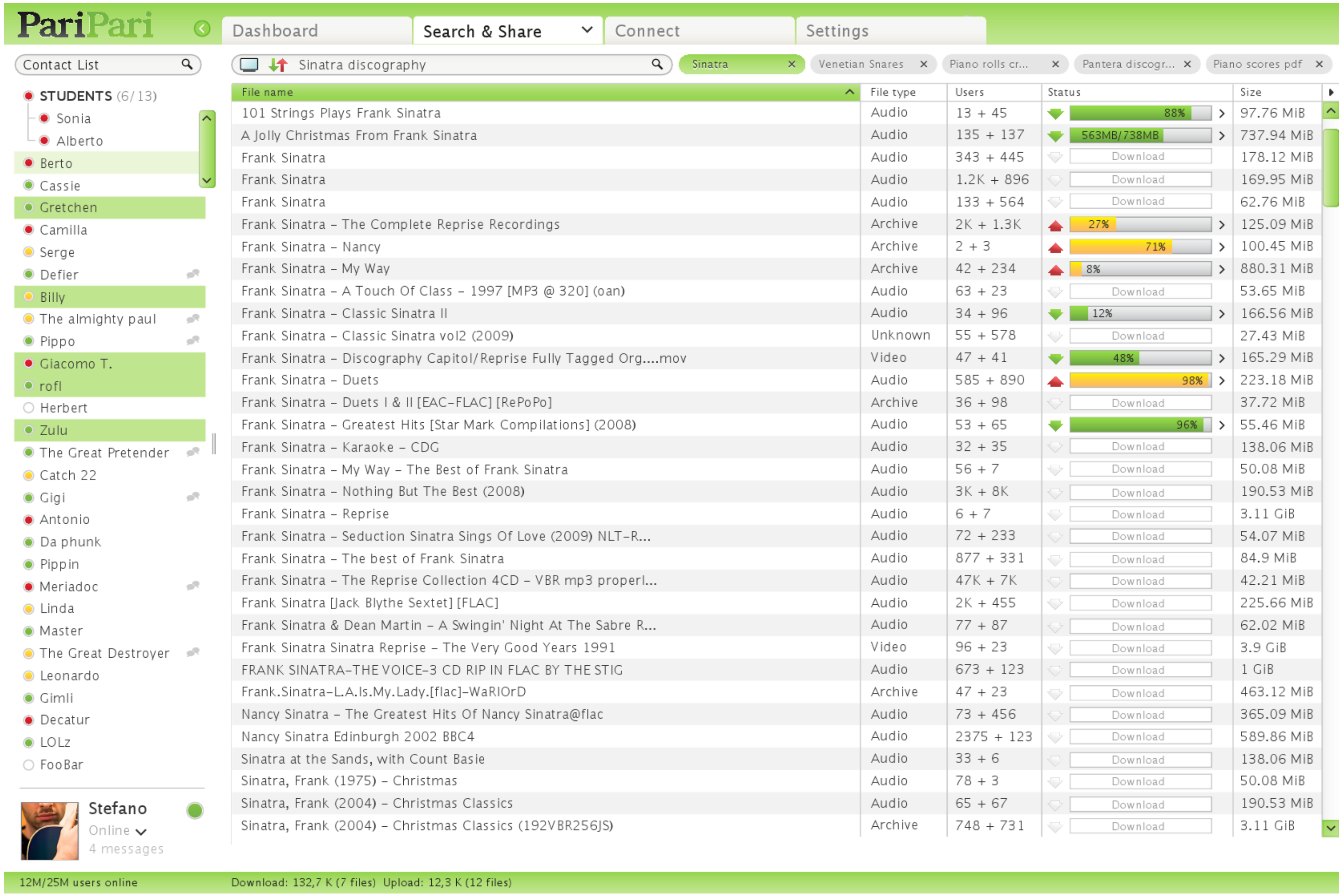
Methods of `GUIAPI` accept objects that implements GUI's interfaces, such as `IPGFileQuery` and `IPGFile`, thus they are implemented by the pre-existing `Search` and `SearchResult` classes of Mulo respectively. This allows us to use objects of Mulo instead of copying their details in a new GUI object that implements these interfaces. The GUI assigns to all these objects an `UUID`[7] we obviously must keep track of for later use. For example when a result have to be shown in a search tab, the GUI needs to know the identifier of the corresponding search. The method `addResults(UUID searchID, List<IPGFile> results)` is used inside the method that processes the search response packets to add files to the corresponding search tab in the GUI.

Listing 4.8: `MuloGUI` class.

```java
public class MuloGUI extends PariPariRunnable implements
   FileManager, Searchable, Downloadable, Uploadable {

   /** The GUI. */
   private static GUIAPI gui;

   /** The references to Mulo's <code>Plugin</code>. */
   private static Plugin father;

   /** If the GUI is active or not. */
   private static boolean active = false;

   /** Thread to request the GUI, it reads just after. */
   private static PariPariThread thread;

   /**
    * Initializes the GUI for Mulo.
    * @param mulo The references to Mulo plug-in.
    */
   private MuloGUI(Plugin mulo) throws IllegalArgumentException {
      father = mulo;
   }

```

[7]A class that represents an immutable universally unique identifier of 128-bit.

```java
/* (non-Javadoc)
 * @see paripari.gui.API.GraphicalPlugin#getName()
 */
@Override
public String getName() {
   return Mulo.MULO;
}

/* (non-Javadoc)
 * @see paripari.gui.API.GraphicalPlugin#getFather()
 */
public Plugin getFather() {
   return father;
}

/* (non-Javadoc)
 * @see paripari.gui.API.GraphicalPlugin#getPublicKey()
 */
public String getPublicKey() {
   return father.getPublicKey();
}

...

/* (non-Javadoc)
 * @see paripari.gui.API.GraphicalPlugin.Searchable#search(java.lang.String)
 */
public void search(String query) {
   Search search = new Search(Search.Type.ADVANCED, query.trim());
   UUID id = gui.addFileQuery(search);
   search.setId(id);
   search.start();
}

/**
 * Adds a list of results to a search in the GUI.
 * @param searchID The corresponding search identifier.
 * @param results The list of results.
 */
public static void addResults(UUID searchID, List<IPGFile> results) {
   List<UUID> uuids = gui.addFileQueryResultBlock(searchID, results);
   for ( IPGFile file : results ) {
      ( (SearchResult)file ).setId(uuids.remove(0));
   }
}

/**
 * Starts the thread to request the GUI.
 * @param mulo Mulo plugi-in reference.
 * @return If the thread is started or not.
 */
public static boolean start(Plugin mulo) {
        if (gui != null) {
            return false;
        }
        thread = MuloThread.startThread(new MuloGUI(mulo), "GUIThread");
        return true;
    }

    @Override
    public void go() throws InterruptedException {
        gui = Mulo.getGUI();
        if(gui.subscribe(this)) {
            active = true;
        }
        else {
            PPLog.printError("Can't start PariPari GUI!");
        }
    }

    /**
     * Says if the GUI is active.
     * @return If the GUI is active or not.
     */
    public boolean isActive() {
        return active;
    }

}
```



STUDENTS (6/13)
- Sonia
- Alberto

Berto, Cassie, Gretchen, Camilla, Serge, Defier, Billy, The almighty paul, Pippo, Giacomo T., rofl, Herbert, Zulu, The Great Pretender, Catch 22, Gigi, Antonio, Da phunk, Pippin, Meriadoc, Linda, Master, The Great Destroyer, Leonardo, Gimli, Decatur, LOLz, FooBar

Stefano
Online
4 messages

File name	File type	Users	Status	Size
101 Strings Plays Frank Sinatra	Audio	13 + 45	▼ 88%	97.76 MiB
A Jolly Christmas From Frank Sinatra	Audio	135 + 137	▼ 563MB/738MB	737.94 MiB
Frank Sinatra	Audio	343 + 445	Download	178.12 MiB
Frank Sinatra	Audio	1.2K + 896	Download	169.95 MiB
Frank Sinatra	Audio	133 + 564	Download	62.76 MiB
Frank Sinatra – The Complete Reprise Recordings	Archive	2K + 1.3K	▲ 27%	125.09 MiB
Frank Sinatra – Nancy	Archive	2 + 3	▲ 71%	100.45 MiB
Frank Sinatra – My Way	Archive	42 + 234	▲ 8%	880.31 MiB
Frank Sinatra – A Touch Of Class – 1997 [MP3 @ 320] (oan)	Audio	63 + 23	Download	53.65 MiB
Frank Sinatra – Classic Sinatra II	Audio	34 + 96	▼ 12%	166.56 MiB
Frank Sinatra – Classic Sinatra vol2 (2009)	Unknown	55 + 578	Download	27.43 MiB
Frank Sinatra – Discography Capitol/Reprise Fully Tagged Org....mov	Video	47 + 41	▼ 48%	165.29 MiB
Frank Sinatra – Duets	Audio	585 + 890	▲ 98%	223.18 MiB
Frank Sinatra – Duets I & II [EAC-FLAC] [RePoPo]	Archive	36 + 98	Download	37.72 MiB
Frank Sinatra – Greatest Hits [Star Mark Compilations] (2008)	Audio	53 + 65	▼ 96%	55.46 MiB
Frank Sinatra – Karaoke – CDG	Audio	32 + 35	Download	138.06 MiB
Frank Sinatra – My Way – The Best of Frank Sinatra	Audio	56 + 7	Download	50.08 MiB
Frank Sinatra – Nothing But The Best (2008)	Audio	3K + 8K	Download	190.53 MiB
Frank Sinatra – Reprise	Audio	6 + 7	Download	3.11 GiB
Frank Sinatra – Seduction Sinatra Sings Of Love (2009) NLT-R...	Audio	72 + 233	Download	54.07 MiB
Frank Sinatra – The best of Frank Sinatra	Audio	877 + 331	Download	84.9 MiB
Frank Sinatra – The Reprise Collection 4CD – VBR mp3 properl...	Audio	47K + 7K	Download	42.21 MiB
Frank Sinatra [Jack Blythe Sextet] [FLAC]	Audio	2K + 455	Download	225.66 MiB
Frank Sinatra & Dean Martin – A Swingin' Night At The Sabre R...	Audio	77 + 87	Download	62.02 MiB
Frank Sinatra Sinatra Reprise – The Very Good Years 1991	Video	96 + 23	Download	3.9 GiB
FRANK SINATRA-THE VOICE-3 CD RIP IN FLAC BY THE STIG	Audio	673 + 123	Download	1 GiB
Frank.Sinatra-L.A.Is.My.Lady.[flac]-WaRlOrD	Archive	47 + 23	Download	463.12 MiB
Nancy Sinatra – The Greatest Hits Of Nancy Sinatra@flac	Audio	73 + 456	Download	365.09 MiB
Nancy Sinatra Edinburgh 2002 BBC4	Audio	2375 + 123	Download	589.86 MiB
Sinatra at the Sands, with Count Basie	Audio	33 + 6	Download	138.06 MiB
Sinatra, Frank (1975) – Christmas	Audio	78 + 3	Download	50.08 MiB
Sinatra, Frank (2004) – Christmas Classics	Audio	65 + 67	Download	190.53 MiB
Sinatra, Frank (2004) – Christmas Classics (192VBR256JS)	Archive	748 + 731	Download	3.11 GiB

12M/25M users online

Download: 132,7 K (7 files) Upload: 12,3 K (12 files)

Figure 4.14: Preview of *Search & Share* tab realized by S. Calgaro.

Chapter 5

Programming and team management

PariPari is a modular project, whose plug-ins have dedicated teams of development and testing; it is moreover completely managed and developed by students, as we already said. Sometimes more plug-ins concerning the same field can be joined in a *confederation*, such as the file sharing one, that is constituted by Mulo and Torrent teams and it includes also the super file sharing project. Students are hierarchically organized inside the project: at the top there is a PhD. student, called the *architect*, that is the one in charge to manage the whole project and all its students.

In this chapter the project management and organization is presented. Here some development methodology and rules are also described, including the whole list of tools used by our developers and testers.

5.1 Tools

Nowadays a lot of tools exist to help the work of developers. It is sometimes not easy to use them, especially in the very beginning, but these tools have so many useful features, that when developers learn their capabilities, it is hard to do along without them. Working in PariPari, students get in touch with some well-known softwares to develop and manage the teamwork. The knowledge of these softwares will be useful in the near future, because they are also used by seasoned professionals.

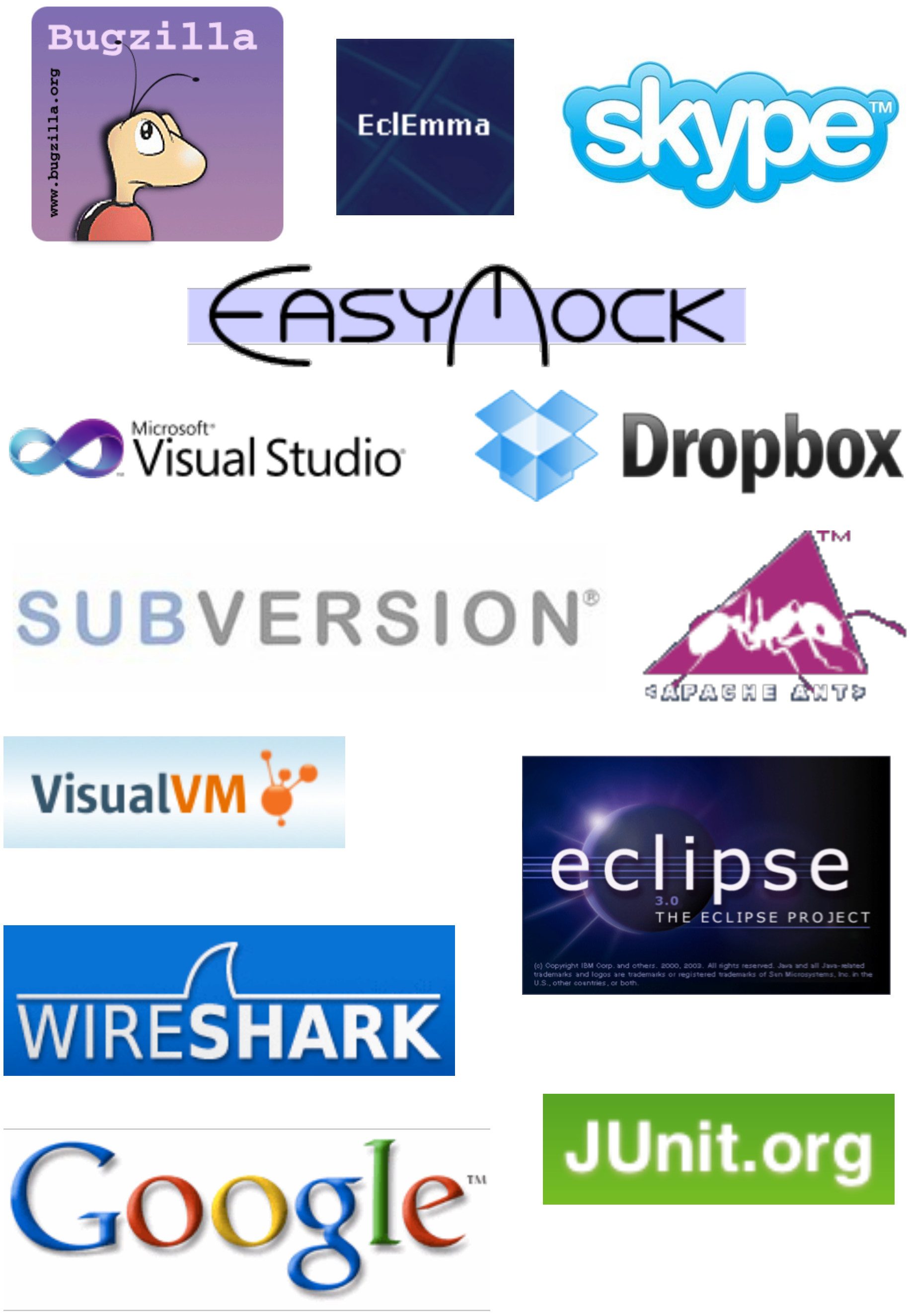


Figure 5.1: Official logos of some of the tools utilized in PariPari.

5.1.1 Developing tools

The absolutely most important software in use in PariPari is *Eclipse.* It is a project focused on building an extensible development platform comprised of extensible frameworks, runtimes and application frameworks for building, deploying and managing software across its entire lifecycle. It is an application that helps computer programmers for software development. Eclipse includes a source code editor, a compiler and interpreter, allowing advanced refactoring techniques, debugging and code analysis. Eclipse is released under the *EPL*[1] and its development is in full swing: every year there is a simultaneous release, the last one was *Juno.* One of its peculiarity is the extensible plug-ins mechanism to add extra features. People know it mainly as a Java *IDE*[2], but plug-ins for a multitude of other programming languages are available.

SVN[3] is another indispensable tool: it is a software versioning and a revision control system. Developers use it to maintain current and historical versions of files such as source code, web pages, and documentation and it semplifies a lot the teamwork. Fortunately there is a plug-in providing support for Subversion within the Eclipse IDE, that is *Subclipse.* It can also perform all the useful actions allowed by Subversion, that will be described in section 5.2.

We use *Apache Ant*, an automating software build processes, to build jars for every plugin. Ant exploits XML to describe the build process and its dependencies. It allows to clean up and compile the source code, to create and sign the jars, to write the Javadoc and finally to run the application with only one click. An integrated component allows us to use Ant in Eclipse.

Students that work in Mulo had to look through the eMule code, above all to do reverse engineering: it is the only real and reliable documentation, but this will be discussed in section 5.6. In many occasions it is even necessary to compile and run eMule in a debug modality. Fortunately eMule is an open source project and source code is available on the official web site, but it is written in C++ and it is set up to be compiled with *Visual Studio.* This is another powerful IDE: it is not free software, but students at DEI can get a free license. Moreover while

[1]EPL - **E**clipse **P**ublic **L**icense is a commercially friendly license that allows organizations to include Eclipse software in their commercial products, while at the same time asking those who create derivative works of EPL code to contribute back to the community.

[2]IDE - **I**ntegrated **D**evelopment **E**nvironment

[3]SVN - It is the abbreviation of *Apache Subversion*, come form the command name `svn`.

Eclipse supports multiple platforms, Visual Studio requires a Windows OS.

5.1.2 Testing tools

In PariPari we give relevance to testing and debugging. When new students are recruited, we even give some lessons about the testing suite. Source code for every plug-in must be tested using *JUnit*, whose library is integrated in Eclipse. It is a simple framework to write repeatable tests. Another useful library is *EasyMock* which provides mock objects for interfaces (and objects through the class extension) generating them on the fly.

There are Eclipse plug-ins that help the work of testers too, one of them is *EclEmma*, that is a free Java code coverage tool. It is also used to collect statistics about code, such as the number of tests, successful tests and failed ones: every month the coverage of the whole project is calculated and the team whose code has the highest coverage is symbolically rewarded with the *Talpa d'Oro*[4].

Eclipse also has a built-in debugging tool, useful to find the portion of code that is responsible for a bug. It allows to find the exact line of code that triggers a malicious behavior. Once the relevant line of code has been found, it also allows to know the context in which the error occurs and the associated values, variables, and methods. When a bug is found, it must be immediately notified on *Bugzilla*, a web-based general-purpose bugtracker.

All these tools combined with some clever software development methodology make the testing experience very useful and interesting.

5.1.3 Analyzing tools

Before developing anything in Mulo, there is always a reverse engineering phase due to the lack of documentation. There are two ways to achieve this: one is to consult eMule code, but sometimes it is useful to try to sniff eMule communications. *Wireshark* results perfect for this purpose, it is the world's foremost network protocol analyzer and it lets capture and interactively browse the traffic running on a computer network. Moreover it recognizes automatically the eMule packets, correctly decoding them in the majority of cases.

[4]In En. "Gold Mole", from the acronym TALPA.

Another useful tool is *VisualVM*, but it serves another purpose, that is to monitor and troubleshoot Java applications by imposing minimal overhead. Mainly it analyzes application performances, such as CPU usage, *GC*[5] activity, heap and permanent generation memory, number of loaded classes and running threads. It also allows to detect suspicious memory consumption and to invoke garbage collection in the application or to take a heap dump, browsing the contents of application heap.

5.1.4 Other tools

A lot of other tools are used by PariPari employees, many of which also in use among non programmers, and some new pieces of technology we are always glad to test as well. First of all we use a multitude of communication systems: *Skype* is the preferred way to communicate directly, allowing us to chat but also to use video calls and it is the best way to set up group conference calls. Then we also make frequent use of *Google Group* as mailing list: there is the general group of the PariPari project to which every PariPari student must subscribe and one that is private for each plug-in. But we also use or have been using *GTalk*, *Google Wave* and *MSN Messenger*; we have even a page and a group on Facebook. We also use tools that allow to share folders and files to concurrently work together on the same resources, such as *Google Docs* and *Dropbox*. Finally tools such as *Doodle* and *Google Calendar* are indispensable to organize meeting, especially when dealing with recruitment sessions or Team Leader Meetings, that usually have many participants.

5.2 Repository organization

The PariPari SVN *repository* is located on a server at DEI: it stores current and historical data of files. Its structure is quite simple and organized; only atomic operations are allowed on our server, to avoid ending in an incosistent state.

A *check-out* is the act of creating a *local working copy* from the repository. The *merge* operation is probably the most problematic: it occurs when two sets of changes are applied to a file or set of files. *Commits* allow to write and merge the local changes made to the working copy back to the repository. A *conflict*

[5] GC - **G**arbage **C**ollector

occurs when different parties make changes to the same document and the system is unable to reconcile changes. They must be *resolved* combining changes or selecting one change over the other. An *update* synchronizes and merges changes made in the repository by others into the local working copy.

There are three principal directories in the repository: one contains several *branches*, another one contains the *tags* and the *trunks* stay in the last one. The underlying botanic metaphor hints that code in trunks should be the - solid - base for unstable yet rapidly growing pieces of code added to implement some new feature or to try and fix some bug or glitch that has been found in the trunk. There is only one trunk for every plug-in storing code that is currently stable (in the computer scientific sense). This means that only bug fixing operations and minor changes can take place in the trunk. The number of branches for every plug-in can vary depending on how many developers work in the team and how many features they are developing. Every branch is a copy of the trunk that allows developers to work in parallel on different features without having one getting in the way of another. A new branch usually is created for every new feature to be implemented and during its lifecycle it must always be synchronized with the related trunk. When the development process is finalized and code is stable, the branch will be merged into the trunk, after that it dies. Finally tags store code for every release.

Every branch, trunk or tag is set up as a Java project in order to make the import from SVN to the Eclipse workspace automatic through a simple check-out. They also follow the same inner structure: source code, tests and libraries are divided into dedicated directories. It is important not to insert references to external files not stored in the project to avoid errors.

As we said, we use the Eclipse plug-in that provides support for Subversion, unfortunately it is not really efficient. Operations of merge are quite complex: every kind of conflicts is always notified, even useless ones. For example if space characters or brackets are differently located in the two versions of code, unnumbered conflicts will be reported by Eclipse. For this reason we usually format Mulo code in a pre-established manner, configuring the code template in Eclipse. Moreover, the larger the lifecycle of an unsynchronized branch is, more its code diverges from the trunk. Thus, when this branch will be merged with the trunk, every added, deleted or changed piece of code will be notified as a conflict, making the merge almost an impossible mission that requires and unnecessarily wastes a lot of time, energy and resources. In some cases new code could also be incompatible with code in the trunk, thwarting all the efforts to develop it.

Eclipse is indeed unable to make a clever search for conflicts, omitting the useless ones, or to recognize trivial ones, automatically resolving them. So Subversion is really useful and simplifies the teamwork, but if not properly used, many issues will be introduced and sometimes their resolution can be hard to achieve.

5.3 Extreme Programming

The *XP*[6] [6] is a agile software development technique, whose goal is to organize people to produce higher quality software more efficiently. XP also introduces a number of basic values, principles and practices that combine together make this methodology very successful.

In XP it is important to plan everything, releases should be small and frequent and *planning meetings* take place to schedule tasks. In fact communications among the entire team is crucial and even more between testers and developers. It is then useful to get a concrete *feedback* from users.

Simplicity must be adopted in design, then refactoring should be done whenever and wherever possible to keep the design simple as the project goes on and to avoid complexity. Integrations must also occur often, but one at the time.

A lot of effort is spent in testing, in fact all code must have unit tests and must pass all unit tests before it can be released. XP includes also *programming in pairs*, which means that two developers test their code each other. Another adopted technique is *TDD*[7] that relies on the repetition of a very short development cycle. First the developer writes a failing automated test case that defines a desired improvement or new function, then he produces code to pass that test and finally refactors the new code to acceptable standards.

[6]XP - **E**extreme **P**rogramming

[7]TDD - **Test D**riven **D**evelopment

5.4 Recruitments

We usually do not expect that recruited students are prepared to work in PariPari: in the best case they know the basics of Java and something about data algorithms, but we have also recruited some very good and experienced developers. In the very beginning it is not necessary that students have particular knowledge of tools and methodologies used in PariPari, but they must show at least good problem solving skills and an enthusiastic approach to learn, then in about a month they must be ready to develop and test.

Initially there was a recruitment at the beginning of every academic year and it didn't plan any kind of examination: moreover students were randomly assigned to plug-ins. This was not a very clever methodology and often results were not so good, in fact many students proved not to be very productive. Now something has changed and recruitments are handle in a more professional way: now an entrance exam (that could be a test or an homework) takes place. We do also interview both to understand the skill of students and to know their expectations and interest. When we recruit students, we do not only care to fill vacancies, but we also try to assign them to a plug-in they are interested in, in order to find *the right student for the right plug-in.*

An important and in some way surprising thing we learn from recruitments is that there is absolutely any correlation between the academic career of a student and its contribution to the project or its code and design skills. On the contrary we have good developers not only with a quite successful academic career but also with a very low grade average. Motivation in PariPari is another crucial point, in fact it is the main source of student productivity. We have often noticed that motivated and uneducated students perform better than capable but unmotivated ones. More statistical details about this phenomenon can be found in [16].

5.5 Team organization

All the teams follow the same internal organization: there are a *team leader* and a *tester leader*, but in some occasions these two jobs are assigned to the same person. Then every team has several students working on it, the amount of them depending on the size of the plug-in, but usually ranging from three to eight, although some trivial plug-in can be followed by only one person.

5.5.1 The team leader

The team leader is the most important figure inside a team, having several assignments and duties. Leader usually are Master students and of course senior developers, that have worked in PariPari for at least one year. They also are students that have risen above the group and that have kept up with the job and the concerning duties.

The leader must have a complete view and comprehension of the source code and he must know at what stage of development his plug-in is. The leader also has a full knowledge of the protocols. Obviously he is a good Java developer and, when he is the tester leader too, he must also know testing libraries and processes. The leader must be aware of what the plug-in issues are. He also is the one that merges work of other developers, managing the trunk into the SVN. Moreover the leader should know the plug-in checklist in order to plan the roadmap with the architect. The communications between distinct teams occur between theirs leaders and for this purpose there are frequent meetings to which also the architect takes part.

The leader obviously is the one in charge to follow the students assigned to his plug-in. He has in the first place the task to introduce new students inside the team, teaching them about the methodology and the tools in use. Moreover he decides what assignments to give, although the will of the student is also taken into account to come to an agreement. After assigning a task, he must monitor all the student activity during the whole development and testing process until the degree, also ensuring that all the rules are followed, including deadlines. The leader of course must motivate and give a boost to his team: sometimes rewarding students that achieve important goals can be a clever strategy. But he also must scourge and if possible prevent misbehaviors and inefficiency, taking appropriate measures.

Thus this is not an easy job, especially because the leader not only must have control over his students, but he also had to gain their respect.

5.5.2 The ideal Mulo developer

Since it was born, many students have worked on Mulo: its team can be considered among the most numerous and its activity is very sustained and dynamic.

Students had to observe some basic rules to allow everyone to hit it off.

First of all students had to cooperate and to keep others updated about any kind of issues or improvements, the team leader above all. They must respect the deadline, decided at the beginning of every assignment, reporting delay. It is moreover useful when students join forces and help one another, even if, when more students work together, it is hard to know if they all commit in the same manner. So the team leader should beware of group assignments.

Another very important thing is that every developer must commit frequently his work, to avoid lazy loss due to potential damage of local files. But also the source code update matters and it should be performed at least in the beginning of every new work session. Finally every branch must be always synchronized with the trunk. All these precautions are needed to avoid conflicts, in fact working on obsolete source code can compromise everything what has been done and even worse it always make future commit, update and synchronize more complicated.

Source code should be written with good style, following software engineering guideline and canons. Documentation is then vital in Mulo, so everyone have to report everything is discovered about the eMule protocols and also how features are developed. Javadoc of all code must also be maintained by everyone. In fact a real issue not only in Mulo but in the whole PariPari project is the fast turnover of employees, that usually work on a plugin for few months until the degree, unless, as it happens sometimes, they decide to continue as Master students. So an up-to-date and accurate documentation make the work of others easier, when something needs to be changed or improved in the future or to also introduce new developers in the team.

Furthermore developers must be sure that the goal is reached and that Mulo still works fine, thus a rigorous debugging takes place before integrating something into the trunk or before releasing a new version of Mulo. Finally of course anyone can avoid testing because it is sometimes boring: to avoid this behavior when a bug is introduced and not caught by the tester, we usually blame the tester and it will be the one which have to fix it. In fact it is impossible to write always correct source code, instead it is easy to introduce a bug and often developers have difficulty to judge their own code. Sometimes all these regulations seem extreme or heavy, but it is mainly in this way that Mulo has reached very significant results, becoming one of the most advanced and running plug-ins in PariPari.

5.6 Lack of documentation

Unfortunately, at the best of author's knowledge, an eMule official documentation does not exist, thus it is not an easy challenge to develop an eMule client. The only real information about eMule and its protocols can be found in [25] and [15], but these are incomplete, obsolete and furthermore we have found not only imprecisions but also some mistakes. Therefore in a manner of speaking the only reliable documentation is the source code of eMule, that is open source and can be downloaded from the eMule official web site. However it results often hard to understand, both because it is written in C++, while the majority of our students know only Java, with a low level of readability also caused by the absence of comments. Sometimes we try to make up for this problem by reverse engineering the protocol: exchanged packets with eMule clients are sniffed in order to understand what and how packets are sent in every different occasion. But in some cases this solution is also hard to use or even not applicable, so a lot of patience and creativity are needed to develop any of the eMule features.

As a consequence of the lack of documentation, a lot of dialects are born: every time developers are not able to understand and reproduce the eMule protocols, there is the risk to introduce malformed packets, as for example the VeryCD Mod does with the AICH packets. In other cases packets have some differences, but they are still utilizable, instead the worst situation is when totally new packets are created by other clients, changing the standard protocol. This means that when an eMule feature is implemented in a client, it is not enough to understand how eMule handles it, but we must also make out how other clients implement it. Moreover we must act accordingly to the client we are connected to, sending the correct packets that it expects to receive, otherwise the communication could be closed.

So it is not hard to believe that at the present time the best eMule documentations are the theses written by PariPari students that have worked on Mulo plug-in. This also explains why Mulo developers must carefully document everything, mainly what is not reported elsewhere.

Bibliography

[1] BitTorrent web site. `http://www.bittorrent.org`.

[2] eMule Project web site. `http://www.emule-project.net`.

[3] Facebook web site for developers. `http://developers.facebook.com`.

[4] Peer Guardian 2 documentation. `http://www.emuleitalia.net/fora/index.php/topic,18915.0.html`.

[5] Di Pieri A. *PariMulo: Autologin e Annotazioni degli utenti*. 2010.

[6] Wells D. *The Rules of Extreme Programming*. 1999.

[7] Peserico E. *P2P Economies*. 2006. In Proceedings of the ACM SIGCOMM.

[8] Mattia F. *PariMulo: Kad*. 2011.

[9] Peruch F. *PariPari: Connectivity Optimization*. 2011.

[10] Postel J. *User Datagram Protocol. RFC 768 (Standard)*. 1980.

[11] Postel J. *Transmission Control Protocol. RFC 793 (Standard)*. 1981.

[12] Bonazza M. *PariCore*. 2009.

[13] Muscarella M. *PariPari - Mulo: AICH*. 2009.

[14] Samory M. *PariGUI 2010*. 2010.

[15] Heckmann O. and Bock A. *The eDonkey 2000 Protocol*. 2002.

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

[17] Jones P. *US Secure Hash Algorithm 1. RFC 3174 (Standard).* 2001.

[18] Maymounkov P. and Mazíeres D. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* 2002.

[19] Ampezzan R. *PariMulo 2009.* 2009.

[20] Rivest R. *The MD4 Message-Digest Algorithm. RFC 1320 (Standard).* 1992.

[21] Shamir A. Rivest R. and Adleman L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* 1978. Communications of the ACM 21.

[22] Calgaro S. *PariGUI - Progettazione e realizzazione dell'interfaccia grafica per un software peer-to-peer.* 2010.

[23] Daberdaku S. *PariMulo: Credits.* 2010.

[24] Pelizzaro S. *PariMulo: Ottimizzazione del File Sharing.* 2011.

[25] Kulbak Y. and Bickson D. *The eMule Protocol Specification.* 2005.

List of Figures