



POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Corso di Laurea Magistrale in Ingegneria Informatica

**DEVELOPMENT OF A LOCAL CLOUD SYSTEM
BASED ON P2P FILE SHARING**

THE MYP2PSYNC FILE SYNCHRONIZATION SYSTEM

Supervisors

PROF. FULVIO RISSO

Politecnico di Torino

PROF. JORDI DOMINGO PASCUAL

Universitat Politècnica de Catalunya

Author

FRANCESCO LORENZO CASCIARO

ACADEMIC YEAR 2018/2019

*"We're just two lost souls
Swimming in a fish bowl
Year after year
Running over the same old ground
What have we found?
The same old fears
Wish you were here"*

Pink Floyd

Contents

Abstract	6
Keywords	8
1 Introduction	10
1.1 Project introduction: myP2PSync	10
1.2 Why an user should use myP2PSync	12
1.3 Terms of use	12
2 State of the art	13
2.1 File synchronization system	13
2.2 Distributed application models	16
2.2.1 Client-Server model: characteristics and drawbacks . . .	16
2.2.2 Peer-to-Peer networks: characteristics	18
2.2.3 Peer-to-Peer networks: classification	20
2.2.4 Peer-to-Peer networks: main applications and some drawbacks	23

2.3	BitTorrent	24
2.3.1	BitTorrent Protocol: introduction	25
2.3.2	BitTorrent Protocol: .torrent metafile	25
2.3.3	BitTorrent Protocol: tracker	26
2.3.4	BitTorrent Protocol: chunks exchange	28
3	Project planning	30
3.1	Overall view and stakeholders	30
3.2	Graphical modelization of the system	32
3.3	Requirements analysis	34
3.4	System functioning	36
3.5	Scheduling of the project development phases	37
4	Implementation tools	39
4.1	Desktop application vs Web application	39
4.2	Choice of the programming language	40
4.3	Choice of the framework for the GUI creation	41
5	Solution analysis	42
5.1	Architecture of the system and main characteristics	42
5.2	Code analysis: structure and modules	44
5.2.1	Tracker modules: overview	45
5.2.2	Tracker modules: myP2PSyncTracker.py	46

5.2.3	Tracker modules: reqHandlers.py	47
5.2.4	Tracker modules: group.py	48
5.2.5	Peer modules: overview	48
5.2.6	Peer modules: myP2PSyncClient.py	50
5.2.7	Peer modules: peerCore.py	54
5.2.8	Peer modules: fileManagement.py	55
5.2.9	Peer modules: fileSystem.py	58
5.2.10	Peer modules: peerServer.py	62
5.2.11	Peer modules: syncScheduler.py	62
5.2.12	Peer modules: fileSharing.py	65
5.3	File-sharing protocol	66
5.3.1	P2P Approach	66
5.3.2	Chunks size	67
5.3.3	File-sharing algorithm	68
5.3.4	Random discard approach	70
5.3.5	Synchronization stopped or failed	70
5.4	Devices communication	72
5.4.1	Choice of the communication protocol	73
5.4.2	Tracker side	74
5.4.3	Peer side	76
5.4.4	Messages exchange	78

5.4.5	Message format	80
5.5	Multi-threading architecture	81
6	Implementation choices	85
6.1	Data Structures	85
6.1.1	Synchronization on access	88
6.2	Server reachability	89
6.3	Previous session information	90
7	Main issues	93
7.1	NAT Traversal Problem	93
7.2	Synchronization problems	99
7.3	Path compatibility	102
7.4	Debugging	103
8	How to use myP2PSync	104
8.1	How to run a myP2PSync Tracker application	105
8.2	How to run a myP2PSync Client application	106
8.3	Usage constraints	107
9	Testing	109
9.1	Testing environment and tools	110
9.2	File-sharing protocol parameters optimization	112

9.2.1	MAX_CHUNKS evaluation	112
9.2.2	COMPLETION_RATE evaluation	114
9.3	Previous versions of the file-sharing protocol and their limitations	115
9.4	P2P vs CS performance	116
9.5	myP2PSync vs a similar product	118
9.6	Testing phase results	119
10	Conclusion and future improvements	120
	List of figures	123
	List of tables	124
	Appendix	125
	Bibliography	127
	Acknowledgements	129

Abstract

Nowadays, anyone dealing with computer systems has to work with a wide variety of different files, often large ones. It is therefore essential to have effective tools for sharing them. Another important aspect is that of file synchronization, i.e. making the same version of a file available on different devices and updating it in case of local changes on a single device. These changes must therefore also be reflected on all other devices.

This thesis project focuses on the development of a file synchronization system called myP2PSync, based on a distributed approach to file transmission in which different users share resources in order to synchronize files as quickly as possible, using an algorithm inspired by the famous BitTorrent protocol. It is well known that a Peer-to-Peer system is more effective than one with a normal Client-Server approach when you have to transmit large files and the number of users is significant. The researching area in which this work is carried out is therefore those of computer networks due to the fact that the system is a distributed system. In addition, the field of software engineering is also strongly present, since the application is a software product.

The main feature that distinguishes myP2PSync from other similar systems is the fact that it also makes available to the user the tracker application, i.e. an application that works as a coordinator for all users. Thanks to it, the user can install and run the system locally, without using devices such as servers provided by third parties. This improves privacy and data security. Users, once connected to a tracker, can access through an authentication system based on access tokens to different synchronization groups registered on that specific tracker. Within a group there may be several files subject to synchronization. These files are retrieved by the user as soon as possible in their latest version, if not already present locally. If a user modifies the file in the local area and decides to share his version with other users in the group, this file is updated as soon as possible on all other users' machines. Users can eventually add and remove files from a group using a privilege mechanism.

The following work starts with an introduction of the system that identifies the characteristics and its points of strength. It is followed by a more theoretical part that describes technologies used as a basis for the work, with a particular focus on existing systems for files synchronization, on the Peer-to-Peer model and on the characteristics of the BitTorrent protocol. The third chapter contains a formal description of the system, containing in particular a list of functional and non-functional requirements that the final application must meet. The following sections are much more related to the actual implementation of the system, starting from some initial decisions such as the technologies used for development and then going to describe the code. This description is not exhaustive but tends to focus on the architectural choices and some key features, including for example the file-sharing algorithm. In addition, there is a chapter describing the main difficulties encountered during development, such as the well known problem of NAT traversal. Then there is a part related to the functioning of myP2PSync, a sort of guide to its use. Finally, there is a chapter about the testing of the application, where some choices about the values of the parameters of the file-sharing algorithm are motivated. In this section there is also a description of other file-sharing algorithms previously implemented, then discarded in favor of the latest version of the algorithm that provides better performance. The testing chapter is followed by the conclusive section in which some possible improvements are listed.

In its current state, the system is fully working and stable. All functional and not functional requirements have been met. The performances are satisfactory, even if compared to those of pre-existing applications that are much older and more consolidated. However, the system can certainly be improved, both in the number of features available and in the performance of the file transmission.

The system is distributed as open source and can be found at the following URL:

`https://github.com/flcasciaro/myP2PSync`

Keywords

The following is a list of keywords, with associated explanations, that will often be used from now on:

- Peer: every user of the system.
- Synchronization group: is a group of users and it's associated with a set of shared files.
- Tracker: is the device that manages groups status and acts as coordinator for peers.
- myP2PSync client: is the graphic application that users can use to manage their files and synchronization groups.
- Join operation: when the user access a group for the first time by entering the right token.
- Restore operation: when a user already belongs to the group but switches from the inactive to the active state.
- Leave operation: when the user decides to leave a certain synchronization group and no longer wants to synchronize group files.
- Disconnect operation: when the user decides to switch from the active to the unactive state in a group and no longer wants to synchronize group files.
- Start operation: when the user launches the application client and starts using it.
- Exit operation: when the user closes the client and is disconnected from all active groups.
- Peer status: a peer can be ACTIVE or UNACTIVE in a joined group.

- Peer role: a peer can be MASTER, RW or RO in a group.
- Group status for a peer: can be ACTIVE (group joined and the peer is currently active), RESTORABLE (group joined but the peer is currently unactive) or OTHER (peer doesn't belong to the group, i.e. not joined group).
- Synchronization of a file: a user with sufficient privileges synchronizes his local version of a file or directories with that of the other peers members of his own group. If the user's version is more recent than that of the other peers, the latter will get its version by mean of a download operation. Conversely, if his version is older, he will get the new version, always by performing a download.
- Add file operation: a user with sufficient privileges adds a file to the synchronization group. All the other peers in the group are notified and they can start a synchronization for that file.
- Remove file operation: a user with sufficient privileges removes a file from the synchronization group. All the other peers in the group are notified and they can also removed the file.
- Update file operation: a user with sufficient privileges update the version of a file in the synchronization group. All the other peers in the group are notified and they can start a synchronization for that file.
- Chunk: is a piece of a file with a fixed size. A file can be composed by several chunks.
- Merge operation: is the process of rebuilding a file from its downloaded chunks.
- Seeder: it's a peer who has a complete copy a file, for example is the peer that has added a file to the group or has updated it with a new version.

Chapter 1

Introduction

This chapter contains the introduction to the project, outlining the objective, the main characteristics and the reasons behind the work done. This is described in the following paragraphs:

- 1.1 Project introduction: myP2PSync
- 1.2 Why an user should use myP2PSync
- 1.3 Terms of use

1.1 Project introduction: myP2PSync

The aim of the myP2PSync project is to develop a local system able to provide fast, reliable and secure files synchronization. It keeps updated and equal copies of any kinds of files between different devices belonging to the same synchronization group. The update operation is not real-time, but just when the user decides to synchronize his local version with all the other devices. The system is therefore oriented to all users who want to easily create backup copies of files or simply use the same files on different devices. Also in the company field the application can be useful, allowing to synchronize files used by employees for example. In addition, the system can also be

used for simple file sharing between different users. The end user benefits from the functionality offered by the system through a client application with a graphical interface, simple and easy to use. Groups are registered and managed by a server application, which is executed locally, i.e. it is not a global server used by all the myP2PSync users. So a user who wants to use myP2PSync must know the location of a server, in terms of IP address and port number. Access to a particular group is a function of a key that must be possessed. An user can belong to different groups, covering different roles, at the same time. The role of an user in a subscribed group can be:

- **Master:** it's usually the creator of the group and main maintainer. It has access to information like the list of peers of the group and it can modify the role of other peers, also electing another peer as Master. It can add, remove and update files in the group.
- **Reader&Writer:** users that can manage files in the group like the master, but they don't have privileges regarding the other peers.
- **ReadOnly:** users that cannot manage files in the group. They can only receive files that belong to a group, without the possibilities of adding, removing or updating files.

The role of a device is determined by the access token used during the group joining operation. Indeed, during the creation phase the user can specify two different tokens, one used to access the group as RW user and the other one for RO users.

File-sharing is performed using a Peer-to-Peer protocol inspired by the BitTorrent protocol. This allows the users network to decentralize traffic, avoiding the bottleneck effect on one or more central servers. In addition, a P2P protocol is much more stable and efficient as the number of users sharing the same file increases respect to a standard Client-Server approach. All traffic, both between devices and between server and single device, is transmitted in an encrypted and secure way. The system is cross-platform, so it can be used on Windows, macOS and Linux, but it is not available on mobile devices at the moment.

1.2 Why an user should use myP2PSync

The main problem with existing file synchronization systems is that they use a central server to manage users, groups and their files. Despite there are already systems that work on P2P networks to synchronize files, they still have central servers managed by the company or community used to run the application. This leads to a decrease in user privacy. That's why myP2PSync provides the end user with the server application as well as the client application, allowing him to be completely sure that only users or devices of his choice and who are aware of the presence of this server can access it. As there is no central server managed by a third party, myP2PSync requires a minimum of additional configuration, but all trivially executable by an intermediate user. However, once the initial configuration is done, using myP2PSync is simple and straightforward. One of the strengths of myP2PSync is the simplicity with which a device can join a synchronization group. All this is done by means of access tokens i.e. passwords, which also distinguish the role of the user. So a device just needs to know the server IP address and the group token. In other similar systems, access is managed by adding devices individually via their IP address or via a deviceID from a master account. The whole system is completely open-source and free, unlike many similar systems that require subscriptions or one-off payments in order to be exploited in their entirety of features, even based on the maximum capacity in terms of size of the synchronized files.

1.3 Terms of use

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details.

Chapter 2

State of the art

This chapter describes the main technologies used nowadays for the distribution of files, with a particular interest in the field of distributed solutions. It is divided into the following paragraphs:

- 2.1 File synchronization system
- 2.2 Distributed application models
- 2.3 BitTorrent

2.1 File synchronization system

A file synchronization system is an application that can keep equal copies of a set of files on different devices. This means that any change to a synchronized file on a single device must also be reported on all other devices that own that file. Modifying a file also means removing the file itself, or renaming it. A strict definition of file synchronization can be as follows [1]:

“File synchronization is a process of ensuring that files in two or more locations are updated via certain rules.”

These systems are mainly used for [2]:

- backups - making redundant copies of files so that they can be recovered in case of errors and deletions. For example, a user who wants to keep a backup copy of important files on a second machine that he may rarely use and is therefore more stable;
- data portability between different devices belonging to the same user or company. For example, a user who wants to access a certain set of files from both his laptop and his mobile device or a company that needs to quickly share changes to a certain file with all or part of its employees.

If we only consider the objective of backing up a set of files, we can talk about mirroring. Mirroring, also called one-way file synchronization, is the process of copying a set of files to single or multiple destination devices. Any changes to these files on one of the destinations do not affect all other devices. If we talk more generally about synchronization of files and data portability, we refer to two-way synchronization, where the changes made by a single device propagate to all other devices that have a copy of the file.

Nowadays, there are two approaches mainly used to achieve the files synchronization goal:

- file hosting and cloud storage systems, such as Google Drive and Drop-Box. They allow the user to store their files on a server device managed by a third party, usually the company that offers the service. Users do not use memory on their devices and can access their files at any time from any device, usually via a user identification mechanism. This approach is clearly based on a client-server model, thus suffering from all the advantages and disadvantages typical of this architecture. Moreover, being the storage space made available by the service provider, it is usually limited and subject to payment. Such payment can be in function of the space actually used or according to the models of subscription to the service.

- files synchronization systems, such as Resilio Sync, SyncThing e GoodSync. These systems, as well as myP2PSync, are based on sharing files directly between the devices concerned, while still using public server equipment for device registering. The method of file sharing used can be either client-server, where a device requires an entire file to another device, or peer-to-peer where files are shared thanks to the cooperation of all active nodes.

The rest of the paragraph will focus only on the latter, because the file hosting services are far from the goal of the project.

As mentioned above, some synchronization systems work using a client-server model. The devices are organized according to a star topology in which the central node works as a "hub" and all other devices as "spokes". The hub node is responsible for transmitting the entire file to all other connected devices. Other systems instead exploit a decentralized network called Peer-to-Peer network in which generally all devices play the same role and file sharing is carried out in a more equal way from the point of view of resources. The next paragraph will focus mainly on the differences between these two approaches, detailing in particular the P2P model, being the one used by myP2PSync.

Some features and functionality generally provided by a file sync system are as follows:

- encrypted file transmission, especially in the case of Internet connections, to increase synchronizations security;
- compression techniques used before data transmission, especially useful in the client-server model where entire files are sent and therefore the time required for the operation can be considerably long;
- ability to recognize any changes in a file, so that it can only be synchronized if necessary. Some applications are able to synchronize by sending only the modified parts of a file, using an approach similar to the one used by rsync, an UNIX file synchronization utility.

Some synchronization systems also offer the possibility to synchronize files at real-time, i.e. as soon as a change is detected it is propagated to other devices. Other systems require a manual request from the user.

2.2 Distributed application models

This paragraph focuses on the description of the two network models most used for the design of distributed applications, highlighting their advantages and disadvantages. Particular attention is given to the Peer-to-Peer model being the one used in myP2PSync.

2.2.1 Client-Server model: characteristics and drawbacks

Since the birth of the Internet, the Client-Server architecture, also called Server-based, has been the most widely used solution within computer networks to provide services across the network [3]. It is the basis of the operating principle of the World Wide Web and most web applications. It is also used for file-sharing, for example by the File Transfer Protocol i.e. FTP. This solution is characterized by a distributed model but very unbalanced in terms of resources and traffic. In fact, there is a strong distinction between client machines, which require a service, and server machines, which instead must offer a service in response to a request. The latter generally have to serve a large number of requests, even at the same time, so they have to be designed to support a high workload. They must have an adequate amount of both computational and network resources. Devices that act as clients, usually PCs, laptops and mobile devices, can have minimal resources.

The main advantages of a Client-Server approach are the followings.

- Centralization of control - access to resources and data is centralized, i.e. it is managed exclusively by the server. This makes it easier to guarantee integrity and consistency properties, especially through authentication and synchronization mechanisms. It is also easier to avoid damages or corruptions of data.

- Easy maintenance - this property is known as encapsulation. Client and server machines are separate entities with different roles and responsibilities. This makes it easier to make operations such as server-side changes, or even moving servers, without affecting clients.
- Well-known model. This network architecture has been in use for decades, so it is well designed and all its major problems have been solved. In addition, developers have more experience in developing CS applications.

Instead, these are the main disadvantages.

- The main problem of this paradigm is traffic congestion. In fact the traffic results really unbalanced, the servers tend to work n times more than a client, in a situation in which n clients use simultaneously the service. When the number of requests is higher than the maximum allowed the service can even be unavailable.
- The server can be considered as a weak point of the system. In the event that it does not work properly, for example as a result of an electrical problem or a computer attack like Denial of Service attack, all clients can not take advantage of the services offered. The server is in fact also called single point of failure of the system. A possible solution is to replicate the server in order to be able to switch to another one if the primary server is not working properly. However, this is a very expensive solution and is generally used only for critical applications, when you can not afford the absence of service e.g. healthy related application.
- The software and hardware of a server are usually very important to enable it to work properly in any incoming traffic situation. A regular computer may not be able to serve a high number of customers. Usually we need specific software and hardware on the server side in order to perform correctly all the work. Of course, this will increase the cost of the system.

If we apply this paradigm to our context of a system for files synchronization we can easily see the repercussions of advantages and disadvantages of

the model on the system. First of all, the system should have an always-connected node working as a server/hub. This node should also be equipped with considerable power and network capacity in order to manage the propagation of files to all other nodes in a fast and efficient way. In case of central node disconnection, a protocol should be used for the election of another device and for the notification of the decision to the other machines. The strength of the model lies in its simplicity. The central node can be considered as a coordinator, it deals with all possible situations such as the arrival of different changes from different files. The synchronization protocol would be much simpler.

2.2.2 Peer-to-Peer networks: characteristics

Peer-to-Peer networks, often abbreviated as P2P, have been designed as a solution to the main problems of the Client-Server architecture, namely the need for a node with access to a large number of resources and the possibility of no service in the event of failure of the central node [4]. Peer-to-Peer networks are distributed and decentralized networks. There is no more a distinction between client and server and all the nodes acts in the same way. Each node is called “peer” and can be both client and server of other peers at the same time. The nodes are called peer because are equally privileged, in other words they have the same behaviour and they act in the same way [5]. That is the general concept and it is called pure P2P. That are other types of P2P networks that differs in some way from this definition. Peers make part of their resources, such as their network bandwidth, disk storage or computational capacity, available to all other nodes on the same network. In pure P2P networks there aren’t servers or privileged nodes that play a different role from that of all other peers. Peers are both suppliers and consumers of resources. The following figure shows the different organizations of device in a Client-Server network and in a P2P network.

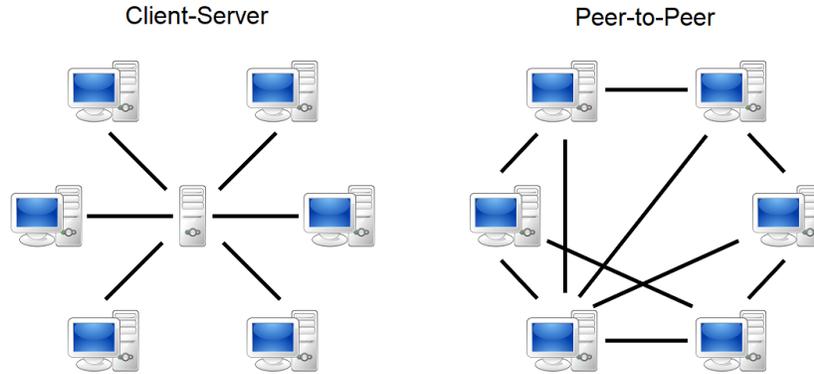


Figure 2.1: Comparison of Client Server and Peer-to-Peer model architectures.

As it is mentioned above in pure P2P networks there is no central node with which all other devices communicate. Instead, connections are made directly between the various members of the network. The P2P networks cornerstone is that workload and tasks are divided among peers. This usually avoids congestion on just a part of the network, like in the C/S model where the server can be a congestion point. In P2P networks each node can start and conclude a transaction that involves other nodes. The P2P networks are generally accessed and exploited through specific software applications e.g. eMule, uTorrent, Azureus.

Peers can have different local configuration in terms of bandwidth, memory available and processing power. Each node in the network serves a certain number of peers, with respect to the own bandwidth. In other words, the more bandwidth a node has, the bigger the number of peers that can serve. In an ideal peer-to-peer network, an algorithm in the communications protocol balances loads among peers, and even nodes with modest resources can help. Even peers who do not have a high amount of resources can participate in a P2P network, as their lacks can be balanced by other peers. Since peers are dynamic entities, because their availability and load capacity change over time, the protocol should also be able to dynamically redirect requests. It's possible to define P2P networks as overlay networks built on top of the physical network and independent from it. Communication is still over the

underlying TCP/IP or more rarely over UDP, but the routing mechanism and the peer discovery process are handled at the Application layer with respect to the overlay network, not to the physical network. The peer addressing is one of the most problematic aspect to solve, because peers are dynamic. They come and go, often changing IP address.

2.2.3 Peer-to-Peer networks: classification

There are different types of P2P networks, distinguished by the following criteria:

- the organization of the peers, that is how they are connected or how the discovery of the other nodes happens;
- how resources are allocated and searched.

The two categories into which we can divide P2P networks are structured and unstructured.

All those P2P networks that don't use any algorithm for the organization of peers in the network are defined as unstructured. Moreover, there is no mechanism for optimizing the network itself. There are three subtypes of this category:

- Pure P2P networks - is the classic peer-to-peer network, which respects all the characteristics described in the previous paragraph. The nodes establish the connections randomly, without any precise scheme. This allows the solution to be extremely scalable and easy to manage. The main disadvantage of this typology, deriving from the absence of a scheme and therefore from the lack of information about the totality of the peers, is the difficulty in finding the nodes and their resources. Requests must be flooded along the network until they reach the recipient, if present. This operation is extremely expensive in terms of time and load on the network. This model is used by the Gnutella file sharing system.

- Centralized P2P networks - in the networks that implement this model there are central nodes that contain information about the peers and their resources. This information makes peer discovery more efficient and allows a certain peer to find the desired resource quickly. The central node works in a similar way to a server but with the big difference that it does not directly manage resources, such as files in a file sharing system, so there is no problem of traffic congestion. The exchange of resources takes place directly between peers. This model is the one used by myP2PSync, as well as by the famous Napster, one of the first P2P applications.
- Hybrid P2P networks - this last sub-category is similar to the previous one but with the difference that the central nodes are not fixed but dynamic. They are periodically elected from among all the nodes and for a certain amount of time assume the role of super-node, being able to provide information about the peers and their resources. This method is used by the Kazaa file-sharing system.

All the P2P networks that instead use an algorithm for the organization and optimization of the network are called structured. They are supplied with appropriate protocols that allow an efficient discovery of peers and resources. The only disadvantage is the greater complexity of network management. The most used solutions are based on the use of a distributed data structure called Distributed Hash Table or DHT, which contains information about the peers and their resources. In other words, this structure is a distributed database and each peer has only a portion of the information. The search key inside the database is usually the string obtained by applying the SHA1 hash function to the name of the resource. The value associated with the key instead addresses the peers that manage that resource. Each peer therefore stores a set of key-value pairs. When a new resource is added to the network, a new entry is added to the DHT. This solution is very scalable because the task of maintaining the correct state of the database is distributed among the various nodes. The various entries are distributed according to an identifier assigned to each peer. Usually this identifier is the value obtained by applying the SHA1 hash function to the IP address of the peer. The resource key and the peer identifier therefore have the same format, usually a 128-bit string. The allocation method consists in assigning an entry to the peer with the same identifier as the key value or to the one with the smallest greater

identifier. For example, if we consider a dummy case with small integer values as results of the hash function:

- peerA has an ID equal to 3;
- peerB has an ID equal to 7;
- peerC has an ID equal to 10.

An entry with key equal to 5 will be assigned to peerB, as well as an entry with key equal to 7. Finally an entry with key equal to 11 will be assigned to peerA, being the DHT a cyclic structure.

The different peers must be connected in a consistent way with respect to the subdivision of the DHT. A widely used linking structure is the Chord, also called Circular DHT, where the peers are connected in a circular structure [6]. Each node is connected only to its predecessor and its successor and it only knows their address. Traversing the structure of the Chord the value of the identifiers will increase. To find a certain resource it is then calculated the hash of the name that corresponds to the entry's key and then the Chord is traversed until the peer that has the entry is reached . When a peer joins the network or leaves it, its entries are reassigned to neighboring peers and the protocol takes care of properly modifying the Chord links. The main problem of this structure is the inefficiency in retrieving the entry, in the worst case in fact all the nodes have to be crossed. The efficiency can be easily improved by inserting additional connections from a certain node to some more far nodes. These links are called shortcuts and make the search for a certain resource logarithmically complex in the number of nodes. The following figure represents a simple DHT with a chord structure improved by the use of some shortcuts. PeerID and entry's key are represented by a small integer value instead of the 128-bit string, just for sake of simplicity.

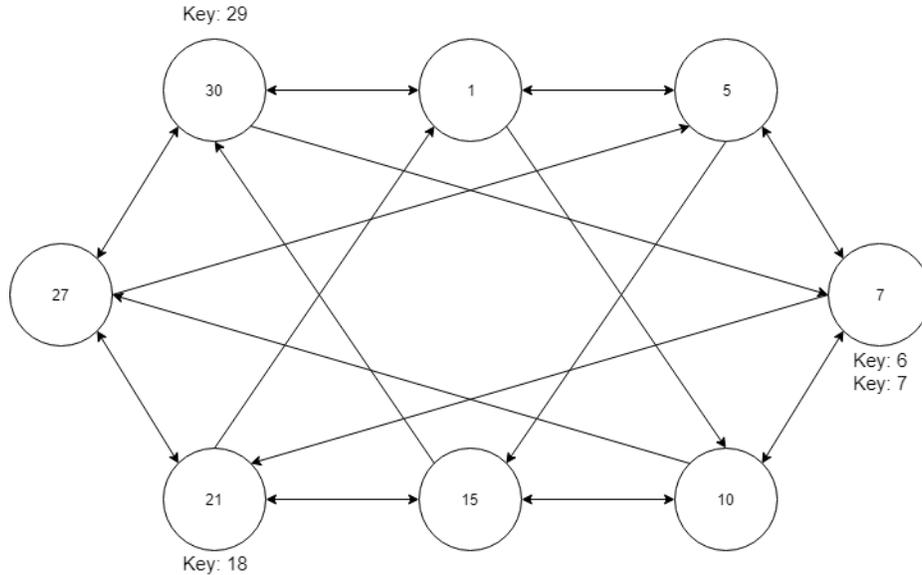


Figure 2.2: Example of DHT chord structure with use of shortcuts.

2.2.4 Peer-to-Peer networks: main applications and some drawbacks

The main context in which P2P networks are applied is file-sharing. In this case the resources represent the files that can be shared by the various users. The strength of P2P networks in the case of file sharing systems is that they are not negatively affected by a high number of users who want to get the same file, indeed in this situation sharing is even more efficient. In the case of the CS model, on the other hand, file sharing is strongly influenced in a negative way as the number of users requesting the file at the same time increases. However, file-sharing applications are not the only ones to benefit from the decentralized approach. Numerous streaming applications e.g. Spotify and Voice over IP applications e.g. Skype make use of this solution. In recent years, also data security technologies such as blockchains have been designed with the Peer-to-Peer model in mind.

However, decentralised networks also have disadvantages [7].

- Some malicious peers may falsify or corrupt resources or information about network routing. The lack of a centralized access mechanism leads to reduced system reliability.
- P2P network nodes are vulnerable because they act both as clients and servers, but generally do not have the security mechanisms that a real server has. For example, they can be the target of a DDoS attack aimed at shutting down the node.
- P2P traffic is generally not anonymous and safe. A possible solution can be Anonymous P2P: some clients like Vuze or FreeNet allow to run a P2P application on anonymous overlay network like Tor or I2P. The main idea is to reroute traffic through volunteer routers, using a multilayer cryptography. Paths are not stable, they are established just at the beginning of the session and they will be changed after a certain amount of time. The receiver will be not aware about the sender IP location or identity. So, peers don't know exactly with which other users they are talking, because they are using just pseudo-identity. Anonymous P2P is a controversial solution because it allows the sharing of illegal resources.

2.3 BitTorrent

BitTorrent is probably the most popular application of Peer-to-Peer networks, and the one that generates the most P2P traffic nowadays. It is a file-sharing application, the most used and widespread, and accounts for more than half of all file sharing traffic on the Internet. It is optimized especially for sharing large files, such as multimedia content. This peculiarity has often made it a victim of criticisms, motivated by the fact that BitTorrent is often the method of communication used by the computer pirates. The core protocol was designed by Bram Cohen in 2002 and the first BitTorrent client was written in Python. Nowadays there are hundreds of different clients, often available for every operating system.

2.3.1 BitTorrent Protocol: introduction

The protocol at the heart of BitTorrent is based on a centralized unstructured P2P network, where the central nodes are called trackers. Nowadays the protocol has also been redesigned to work with a DHT structure, but the first version is the most widely used and popular. It is also the one that has inspired myP2PSync, so the rest of the discussion will focus only on this version. A shared file is divided into smaller pieces of fixed size, usually 1 MByte, called chunks. The various users collaborate by exchanging chunks and the download is completed when all the chunks are retrieved and merged. The set of peers that collaborate to share a single file is called torrent. Peers in a torrent can be classified as:

- seeds or seeders - all the nodes that have completely downloaded the file and now are only working on the upload of chunks. The first seed of a torrent is the content releaser and it needs to stay in the torrent until at least a whole copy of the resource has been sent.
- peers - nodes which have not completed the download and are working both as client and server.
- leechers - all the peers with a bad share ratio. In other words, all the users that download more than upload, for example leaving the torrent as soon as the download is completed, not acting as seeder. This behaviour is strongly disapproved from the community because it can lead to the death of a torrent and usually the BitTorrent protocol tries to penalize this behavior, for example by slowing down the download process.

2.3.2 BitTorrent Protocol: .torrent metafile

The torrent access for a specific content is performed through a .torrent file, which is usually obtained using special online search engines. The .torrent file is a small file, usually some KBytes. It contains only information about the resource and the access to the sharing of the same, it is therefore a metafile

and contains metadata. This information is encoded using Bencode coding, similar to XML. The main information contained are the following [8]:

- announce i.e. tracker location - the IP address and port of the tracker server that manages the resource. It can also be a list of addresses in the case of multiple tracker servers.
- info dictionary - a Bencode dictionary with information about the content:
 - the name of the resource;
 - chunks size, usually 1 MByte or 256 KByte;
 - number of chunks in which the file has been divided;
 - hash of every single chunk - allows to perform an integrity check operation after a chunk reception.

If the resource corresponds to a single file, there is also a field containing the file size. Otherwise, if the content is a directory the info dictionary contains information about every single file, like its size and its path in the directory. Other information contained are a timestamp representing the torrent generation time, some comments and the description of the source.

2.3.3 BitTorrent Protocol: tracker

A torrent tracker is a server that manages the list of peers in a torrent. It is periodically called by a peer to find out which peers are active in order to communicate with for the file sharing. The list returned by the tracker can take into account certain policies such as the limiting of leechers. A peer with a bad sharing ratio may therefore not be listed in the reply and be partially excluded from sharing. To access a torrent, a peer must then make a request to the tracker, or to the trackers if there are more than one and if you want a larger list of peers. Any further requests periodically made by the peer are also used by the tracker to understand that the peer is still active. Communication between peers and trackers is done through the HTTP protocol. A peer sends a GET message specifying [8]:

- info hash - a string that identifies the torrent. It is the SHA1 hash function applied on the info dictionary found in .torrent file;
- peer ID - it identifies the peer on the tracker;
- port number - the port on which the peers is reachable from other peers;
- uploaded bytes - represents the number of bytes sent to other peers since the sharing beginning;
- downloaded bytes - represents the number of bytes received from other peers since the sharing beginning;
- left bytes - the number of remaining bytes in order to complete the download of the file;
- compact - is a Boolean value that will be used by the tracker to know how to encode the list in the response;
- event - represents the download status. Three possible values: “started” at the beginning of the session in order to ask the tracker to add the peer to its list, “stopped” in order to remove the peer from the list, “completed” when the download is done and the peer has become a seed.

The tracker replies with an HTTP message containing a Bencode dictionary with the following information [8]:

- failure reason (if any) - it’s an error message e.g. torrent not present;
- tracker id - it is the identificator of the tracker;
- complete - number of seeders connected;
- incomplete: number of peers connected;
- list of active peers and seeders selected - the format of this list is function of the compact boolean value sent by the peer. If compact is equal to true, the list is just a string, otherwise is a dictionary. In both case

a maximum of 50 peers is returned. For each peer is indicated the IP address, the port number and the peer ID. The latter is present just in the non compact case.

The tracker servers can be public, i.e. accessible by anyone, or private, so the peers must be registered or invited.

2.3.4 BitTorrent Protocol: chunks exchange

The exchange of chunks takes place between the various active peers and seeds in a torrent. The first ones work both in download and in upload, while the second ones altruistically only in upload. Once a peer has subscribed to the torrent through the tracker, it retrieves the list of active peers. This list is updated periodically because peers are dynamic entities and therefore the list changes over time. Until the file download is completed, the peer asks the other peers for the list of the chunks they have and, considering the lists received, it starts to ask for the chunks. The requests are made according to the Rarest-first approach, i.e. giving priority to the reception of the less spreaded chunks, in order to facilitate their diffusion in the torrent as soon as possible [9]. With regard to uploading, a peer does not serve all the other peers at the same time, but only responds to a limited number of peers. This approach is called choking. The selected and unchoked peers are generally 5 [10]:

- 4 are chosen according to the speed at which they are sending chunks, ranking and selecting the fastest ones. This approach severely limits the download speed for any leechers and is called tit-for-tat. The ranking is re-evaluated every 10 seconds.
- 1 peer is randomly selected from those in a state of choking, with an "optimistically unchoke". This technique is used to unlock any peers who can not be prolific in sending chunks just because they do not have the opportunity.

Each time a chunk is received, an integrity check is performed on it, calculating a hash value and comparing it with the one found in the .torrent

file. If the check is successful, the chunks can be added to the list of those already collected and can be sent back to any other peers. In addition, the chunk is added to the partial file that the peer is receiving and rebuilding. When the download is complete the peer can remain in the torrent as seeders or abandon it. In some BitTorrent clients it is possible to enable an option called “local peers”. This allows to exchange chunks only with peers on the same Local Area Network in order to make transmission faster.

Finally, some torrent allows the web seed option, that is a technique for downloading file chunks directly from an HTTP server in addition to the torrent peers. Webseeds are used to guarantee the availability of the download, even if no other P2P users are sharing it. This option could be useful in case of lack of enough seeders or good peers and guarantee a good velocity in the download of the resource. In fact, when the number of seeders is low a peer can use the web seeding and as soon as possible it can switch to the standard peers chunks exchanging.

Chapter 3

Project planning

This chapter contains all the considerations made before the system development phase, at the design level. Possible users of the service are taken into consideration, its functions and characteristics are strictly defined and the various phases of development are detailed. Furthermore, it contains also some considerations made at the end of the work, regarding the real time scheduling of the different steps of the work. The chapter is divided into the following paragraphs:

- 3.1 Overall view and stakeholders
- 3.2 Graphical modelization of the system
- 3.3 Requirements analysis
- 3.4 System functioning
- 3.5 Scheduling of the project development phases

3.1 Overall view and stakeholders

myP2PSync is a distributed file synchronization system and it has been conceived and designed with two application contexts in mind:

- the familiar or personal environment, where users want to have their files available on all their devices;
- the company or business environment, where can be useful to have a system for synchronization of files among workers machines.

The application is useful and efficient in contexts where the source of synchronization is unique or almost but the number of devices benefiting from this file is high. This is due to synchronization timing problems on which the description will focus in next chapters. In other words, the system works well when there are few users who edit files in synchronization, but many users who use files in read-only mode. Moreover, since the file-sharing protocol is based on a P2P technique, the various machines in synchronization must be active enough over time to ensure good files availabilities. It's also designed for those who don't want to take advantage of traditional synchronization clouds, where their data is in the hands of third-party companies. In fact, this is a local system, where all resources are made available by the user. No external servers are used, the whole system is in the hands of those who install and use it.

Taking into account this we can think of a user who wants to synchronize his files on a large number of personal devices or we can find various applications in the business field. For example the synchronization of files between different locations of the same company or the distribution of updates or patches for a software used by different machines. We can also look to the future and think about the world of IoT: the concept behind this application could be used and extended to easily share information between sensors or intelligent objects, such as sharing traffic information between different cars, always taking advantage of the strengths of Peer-to-Peer technologies.

The system requires a minimal initial configuration, so it's designed for environments where there's someone who meets some requirements. Those requirements are:

- basic knowledge of network configuration: IP addresses and ports;
- installation of Python modules;
- minimum knowledge of the command line environment.

3.2 Graphical modelization of the system

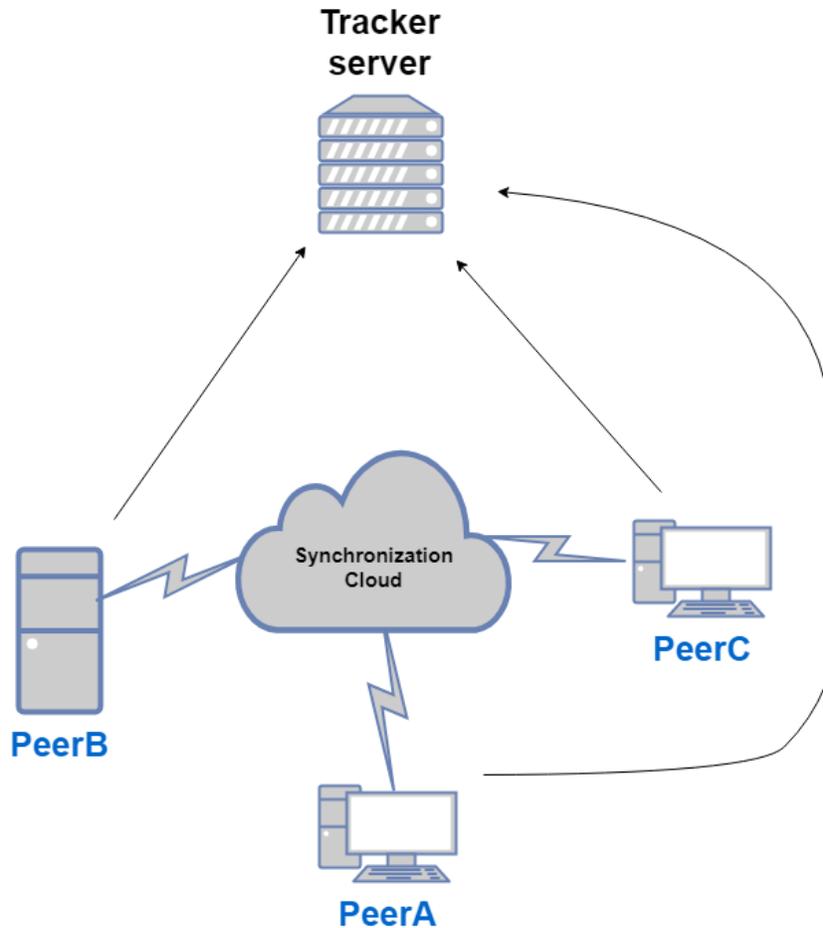


Figure 3.1: General architecture of the myP2PSync system.

In the figure above you can see a very abstract model of the system. The different peers, which use a client of the application are "connected to a synchronization cloud". In other words, peers belonging to the same group collaborate in the synchronization by exchanging messages, information and pieces of files. However, everything is coordinated by a small server appli-

cation that plays the role of tracker and with which peers communicate in order to manage their personal status and to obtain information, for example about the other active peers. The server manages the various synchronization groups and indexes the files of each group.

The following figures are examples of how a peer communicates with the tracker server.

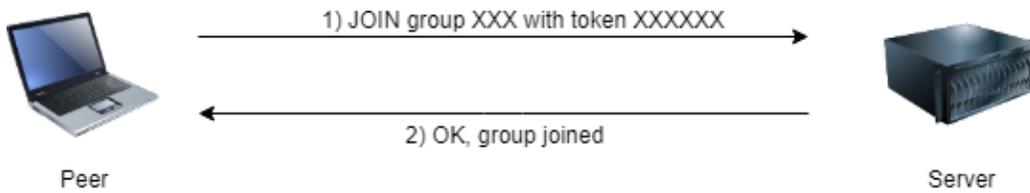


Figure 3.2: Messages exchanged during the join group operation.

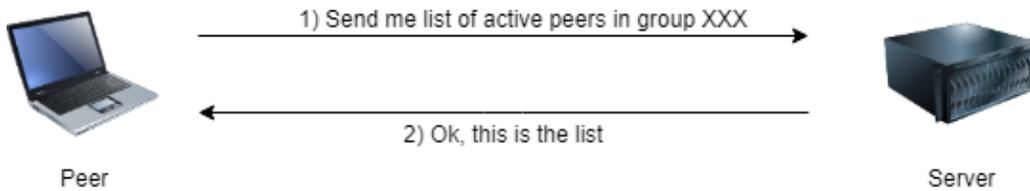


Figure 3.3: Messages exchanged in order to retrieve the list of peers of a group.

The following two figures show how file-sharing takes place between different peers in the same group.



Figure 3.4: Messages exchanged in order to retrieve the chunks list from another peer for a certain file.



Figure 3.5: Messages exchanged in order to retrieve file chunks.

File operations, such as adding, removing, or having a new version, generate a message exchange between both the peer that performs the action on the file and the server and between the peer and all other peers active in the same group at that time.

3.3 Requirements analysis

The system must be able to meet the following functional requirements:

- it must allow the user to create its own synchronization group;
- it must allow the user to join an existing synchronization group;
- it must allow the user to restore a synchronization group;
- it must allow the user to leave a synchronization group;
- it must allow the user to disconnect from a synchronization group;
- it must allow the user to manage their own synchronization group by managing user privileges;
- it must allow the user to also use several synchronization groups at the same time;
- it must allow the user to add files or directories in a certain group;
- it must allow the user to remove files or directories from a certain group;
- it must allow the user to synchronize files with other peers;

- it must stop the synchronization of a file if the file is removed from the group;
- it must stop the synchronization of a file if a new version of the file is present;
- it must ensure file transmission based on an efficient and reliable Peer-to-Peer protocol;
- it must ensure a correct reconstruction of files after a synchronization download process;
- it must allow users to resume any synchronization of files left in suspense in a previous session if the version is still valid;
- it must allow users to know where the synchronized file are stored in the machine;
- it must provide encrypted and secure transmission of files and messages.

In addition, the system must also meet the following non-functional requirements:

- it must be cross-platform and at least available for desktops and workstations (it's a desktop application);
- files must be transferred at an appropriate speed, also in relation to the quality and state of the network;
- the Graphical User Interface should be user-friendly and it must react quickly to user actions;
- all the peers must be able to communicate, also peers behind NAT devices;
- all the communications must be private.

It is important to note that when a user changes his local version of a file, the changes are not redistributed to other peers in real-time. The file must be saved and the user can decide to share its new version using the application.

3.4 System functioning

Here are listed and explained some significant functionalities and behaviours of the myP2PSync system:

- User starts the myP2PSync client application:
 1. the myP2PSync client reads from a configuration file how to reach the tracker device in terms of IP address and port number;
 2. if the reading is successful it tries to reach the device;
 3. if the reading of the file is not successful or the device is no longer reachable at those coordinates, the client asks the user to enter the tuple representing the new IP address and port number of the tracker.
- After the connection with the tracker has been established:
 1. user is proposed to do the restoring of all the groups in which he is already registered;
 2. for each restored group the client automatically resumes any synchronization left pending during the previous session. If a partial synchronization is related to a file which it has an updated version, the synchronization starts from scratch;
 3. at this point the user is free to manage his own groups and files until the closing of the application.
- When the user wants to synchronize with the other peers some files he has modified locally. There are 3 possibilities:
 1. can synchronize individual files;
 2. can synchronize an entire directory of files;
 3. can synchronize the entire collection of files in the group.

The application will recognize the files actually modified and synchronize only those.

- When a client tries to obtain a new version of a file or a new file:
 1. it obtains the list of group active peers from the tracker;
 2. if there aren't peers active it stops the synchronization and it will try again later;
 3. otherwise it starts the file-sharing process with other peers.
- When two users belonging to the same group work at the same time on the same file:
 1. peerA adds a file;
 2. peerB automatically acquires the file after the synchronization procedure;
 3. at this point peerB modifies the file and synchronizes it;
 4. peerA is resynchronized to the new version of the file. If he had modified his local file without synchronizing it, these changes would be lost.
- When the user leaves a group or disconnects from it:
 1. the client alerts the tracker about the operation;
 2. the client stops all synchronization processes related to files in that group.
- When the user closes the client application:
 1. communicates to the tracker his willingness to disconnect and can no longer be contacted by other peers;
 2. stops all active synchronizations;
 3. saves his current synchronization status in order to continue in the following session.

3.5 Scheduling of the project development phases

The application development was structured into the following steps:

1. definition of the problem to be solved and of the possible users;
2. definition of functional and non-functional requirements of the system;
3. definition of the architecture of the system;
4. selection of tools to be used to develop it;
5. code writing phase;
6. testing phase.

The following is a Gantt diagram showing the actual timeline of the project, which has been developed between February and July 2019.



Figure 3.6: Gantt chart representing the timeline of the project.

The first four steps were carried out in parallel, being closely related. The same has been done for the last two phases, in fact the testing phase has inevitably led to some changes in the previously written code, mainly for reasons of optimization of the application. Moreover, in order to have quickly a first minimal but working version of the application, a client version without a graphical interface has been developed first. Subsequently, the GUI was developed and added following a step-by-step integration with the existing core functions.

Chapter 4

Implementation tools

This chapter contains a description of the main technologies and solutions used in the development of myP2PSync, and the decision-making process that led to the choice of certain tools and not others. The following paragraphs are therefore:

- 4.1 Desktop application vs Web Application
- 4.2 Choice of the programming language
- 4.3 Choice of the framework for the GUI creation

Git is used as the version control system. Other implementation choices, closer to solution analysis and belonging to an higher level of detail, are treated in chapter 5 and 6.

4.1 Desktop application vs Web application

The first choice made during the implementation phase was between a Desktop application, i.e. an application that can be launched directly from the operating system, and a Web application, which runs within a browser. We opted for a desktop application because it is generally more performing and

independent from the quality of the network. Moreover, implementing a web application would have been more difficult to achieve the distinctive feature of myP2PSync, that is the use of a local server and not of a global one. It was also a matter of personal taste, simply by seeing more appropriate a desktop application for a file synchronization system.

4.2 Choice of the programming language

Nowadays there is a multitude of programming languages for the development of Desktop applications. The languages most taken into account during the decision-making phase have been:

- C++, a high-performance and very solid language, Object Oriented version of the historical C. A must in programming. Unfortunately not too suitable for the prototyping of applications because it is quite complex, being among the high-level languages one of the most "low-level";
- Java, another fundamental language. Strongly linked to the object paradigm, with a strong typing, it is very solid and not prone to errors. Despite being an interpreted language, it is still very performing thanks to the optimizations in the execution phase, including the use of a Just in Time Compiler.
- Python, the youngest of the three languages. It's not a strongly typed language, it's one of the highest level languages of the whole spectrum. It is interpreted but still guarantees excellent performance and is especially suitable for prototyping, i.e. the development of systems and applications in the short term.

The final choice went to the Python language because in recent years its diffusion is becoming more and more widespread. Being the language decidedly high level, it allows complex operations even with a few lines of code. It also has a wide variety of modules that give an excellent experience to the developer. The fact that it is an interpreted language can be considered a

small disadvantage for performance, but being myP2PSync an application that does not require excessive computational capacity to work well, the difference compared to a compiled language would have been minimal. The integrated development environment used was the pro version of Pycharm, which has an excellent debugger and a real-time code inspector.

4.3 Choice of the framework for the GUI creation

The last macro choice was about the framework or widget tools to use to write the graphical interface code. Python has more than a dozen GUI creation modules, but the focus has been on two solutions:

- Electron, which is a framework for creating desktop applications using web technologies such as HTML, CSS and Javascript. Nowadays it is the most used solution but it requires the knowledge of the Node.js run-time routine.
- PyQt version 5, which is instead a Python binding of the famous Qt framework. It's probably the solution that gave life to most desktop applications developed in Python. It is a free library if not used for commercial purposes, as in the case of myP2PSync which is completely open-source.

The final choice was PyQt5 because of the pervasive presence of documentation on the web. The documentation is still managed by Nokia and is really complete. PyQt5 also provides other useful tools for signal management and communication between threads. Electron has been discarded for the lack of programming experience with Node.js.

Chapter 5

Solution analysis

This chapter contains a fairly in-depth analysis of the solution, going to detail the architecture of the application, the various modules written, the file-sharing protocol, how communications between devices take place and the structure of the application at the thread level. The chapter is divided into the following paragraphs:

- 5.1 Architecture of the system and main characteristics
- 5.2 Code analysis: structure and modules
- 5.3 File-sharing protocol
- 5.4 Devices communication
- 5.5 Multi-threading architecture

5.1 Architecture of the system and main characteristics

The myP2PSync project is based on a distributed system essentially built around two components:

- a client component, i.e. the application that is used by users to take advantage of the functionality of the synchronization system. Every user running a myP2PSync client application is considered a peer of the system;
- a tracker server component, which plays the role of coordinator and is essential for the proper functioning of the entire system.

The following figure shows a general view about the architecture of the system:

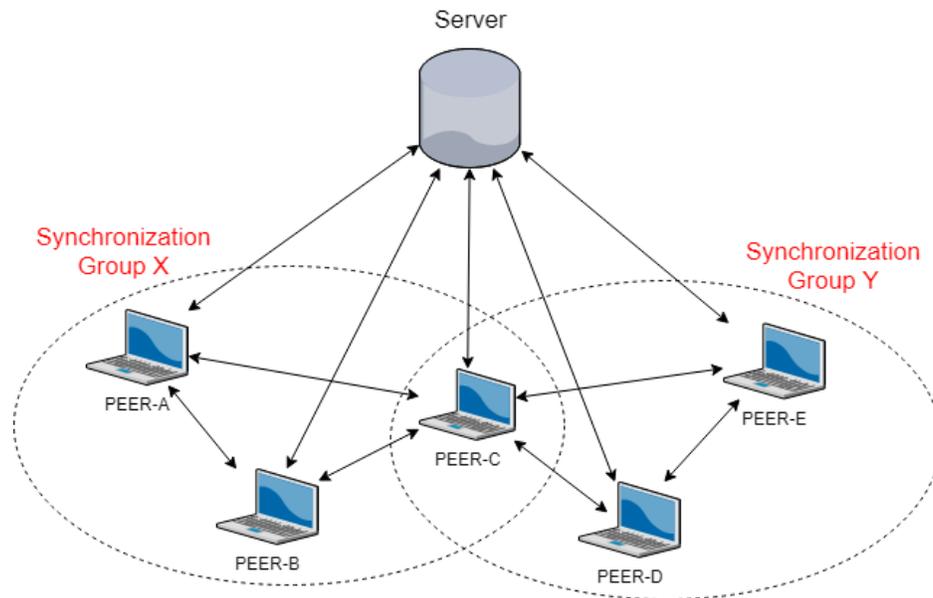


Figure 5.1: Overview of the myP2PSync system architecture.

The image shows five peers and the tracker server to which they are connected. Peers A,B,C are connected to the synchronization group X, while peers C,D,E are connected to the group Y. In the case in which all the peers in the figure are connected, peerC can interact with all the other peers. The system is coordinated by a tracker server that represents the central point where peers can retrieve all the information on files and groups. It provides useful information for the initial configuration of the individual peer and works as a tracker to discover and find other active users in a certain group.

However, the system focuses on communication between peers, not only for exchanging files, but also for information about their status. For example, when a peer adds a file to a group, it notifies both the server and all other peers active at that time of its action. The first one will register the action allowing even offline peers to learn about it once online. The latter instead will react real-time to the action, trying to get the file as soon as possible by means of a download operation. Next paragraphs explain in more detail the individual technological solutions used for the two different components and how they communicate.

5.2 Code analysis: structure and modules

The system is built around two software applications: the first runs the tracker server while the second runs the myP2PSync client of the individual peers. Both are written entirely in Python3 and use JSON files to keep track of information from previous sessions. They have only one module in common which is the one that manages the networking functions, like join and leave the network or send and receive data. An important difference between the two applications is that the client has a graphical interface while the tracker works on the command line. The following paragraphs contain a description of the various modules that make up the system, starting from those on the server side. However, the description will not be exhaustive, i.e. it will not be detailed all the code line by line, or even function by function, but only the salient and most interesting parts, trying to motivate why some solutions have been adopted instead of others.

In the creation of the two parts mentioned above, consistent design choices have been made, since the development has proceeded in parallel and incrementally. In other words, the tracker was designed to work perfectly with the client and vice versa. However, despite this desire to use coherent and similar solutions for the two components, in some cases I have chosen to use more specific approaches for reasons of efficiency and scalability, considering that the way clients and servers work is very different. These differences are especially relevant in the choice of data structures to use. No attempt was made to reuse existing code, even for individual data structures. All deci-

sions were made in order to create an ad-hoc solution to the problem that works well in terms of reliability and performance.

I would also like to make it clear that in the following paragraphs I will insert some parts of the code to make the whole report less abstract and easier to follow. This code is sometimes simplified or lacks some details that are not essential. Other times it is pseudo-code to make it even simpler, without extricating itself in useless syntactical difficulties. The original and working code can be viewed on my personal GitHub at the following link:

<https://github.com/flcasciaro/myP2PSync>

5.2.1 Tracker modules: overview

The tracker server structure is quite simple and consists of only three modules:

- the `myP2PSyncTracker` module that implements the main features of a multi-threaded server and it's the script that must be runned in order to have an active `myP2PSync` tracker;
- the `reqHandlers` module that contains the behavior of the tracker in response to different requests that may come from a peer;
- the `group` module that models and manages the individual synchronization groups.

In addition, the tracker server uses the networking module, also used by the peers. Its content is detailed in section 5.4.

The following paragraphs contain a detailed description of the individual modules and how and when they interact.

5.2.2 Tracker modules: `myP2PSyncTracker.py`

The `myP2PSync` tracker server simply waits for requests from peers and uses a multi-threading approach to serve them. Its main task is to manage synchronization groups, allowing operations such as joining or deleting from a group. Its role in keeping track of the files in a group is also fundamental, through special labels containing the main information such as filename, file size and timestamp (the latter serves as an indicator of the file version). Another function of the server is to keep track of the position of the peers in terms of IP address and port number. Each peer is identified by an ID, which is obtained from the MAC address of the machine on which it works, so it's only dependent on the machine on which `myP2PSync` is running. This identifier is used by the tracker server to recognize and distinguish the various peers making requests. It then associates each `peerID` with the current IP address and port number, which are dynamic as the peers can move to other networks. These information are stored in a dictionary where the key is the `peerID` and the value is a dictionary containing informations like the IP address and the port number where the peer is reachable.

Before the actual start of the tracker server, the application tries to retrieve information about a previous session. In fact, every time the server application is terminated, it saves the current status in JSON files before closing. The server status is composed of all the information about the synchronization groups, for example access tokens, peers registered to the single group and managed files. The server then retrieves this information and allocates and initializes the appropriate data structures. After that it starts effectively to act like a server, listening on a well-known port i.e. 45154. From that moment on, it waits for connection requests and it serves them, until it is terminated by an interrupt signal such as CTRL+C.

Each time a connection request is made, the main thread generates another thread that has as its unique purpose to serve all requests on that connection and then terminate following a connection end message, i.e. "BYE". Threads are configured in daemon mode and then end automatically, without the need for the main thread to recover their termination status. By making an analogy with C programming, they are placed in detach mode, and on termination they automatically release all owned resources. Each message

sent by a peer to the server is preceded by the peerID and contains a word that represents the type of request. It is followed by all the parameters that are necessary to serve the request, for example during the JOIN operation the request must specify the name of the group and the access token, encrypted using the md5 hash algorithm. The thread that receives the request first identifies the peer using its identifier and then, thanks to a construct switch, activates the appropriate handler.

Information about groups is stored in a dictionary data structure, where the key is the group name. Therefore, there cannot be homonymous groups on the same server. The value instead represents all the information about the group, such as files and peers. For more information please refer to paragraph 5.2.4.

5.2.3 Tracker modules: reqHandlers.py

The reqHandler.py script is nothing more than a set of functions that are called by the different threads to serve the requests. They are essentially functions for group manipulation, both at the peer level and at the file level operating on the server's data structures, using mutex-based synchronization mechanisms. It should be considered that being the tracker server multi-threaded, some threads may generate inconsistency in the data structures as a result of race conditions. For example, if two users were to log into the same group at the same time, there could be a situation in the update of the active peers counter where there is a single increment (+1) instead of a double increment (+2). In other words, there is the lost of an increment due to a race condition generating inconsistency in the data. Of course, these situations are extremely rare, but it is still important to manage them carefully using the synchronization tools provided by the language. For the complete list of requests and consequent actions of the tracker, please refer to appendix A.

5.2.4 Tracker modules: `group.py`

This last module contains the structure of a group with all its information and other accessory data structures useful for its manipulation. This script defines three classes:

- `Group`: describes the single group, has as properties the name, the number of total peers, the number of active peers, the number of files managed by the group and finally the two authentication tokens, naturally in an encrypted format. The methods offered by the `Group` class concern the manipulation of peers and files in terms of addition or removal or modification, or the retrieval of group information.
- `PeerInGroup`: describes the status of a single peer within a group through its `peerID`, its role (Master, RW or RO) and its being active or inactive.
- `FileInGroup`: describes a synchronized file within a single group using the file name, its size in terms of bytes and its timestamp as properties. This last parameter identifies the version of the file.

Each group keeps track of peers and files using dictionaries. In the first case the key is the `peerID` while the value is a `PeerInGroup` type object. In the second case the key is the file name while the value is a `FileInGroup` type object.

5.2.5 Peer modules: overview

The code structure used by the individual peers is organized into the following modules:

- `myP2PSyncClient` that generates and controls the graphical interface of the client and is the script that users need to run in order to use the application client;

- peerCore that contains all the code necessary to manage groups and files, communicating with the tracker server. It also performs some initialization functions;
- fileManagement that defines and manages the properties of synchronized files;
- fileSystem that contains the data structure used to store information about the files of the various groups;
- peerServer that allows the peer to work as a server in order to receive messages from other peers;
- syncScheduler that manages the various synchronization processes;
- fileSharing that contains all the operations for a file synchronization, from the protocol for files download to the reception and sending of chunks.

In addition, some of these modules make use of the networking module, also used by the server. Its content is detailed in section 5.4.

The following figure shows the organization of the various modules and their interactions:

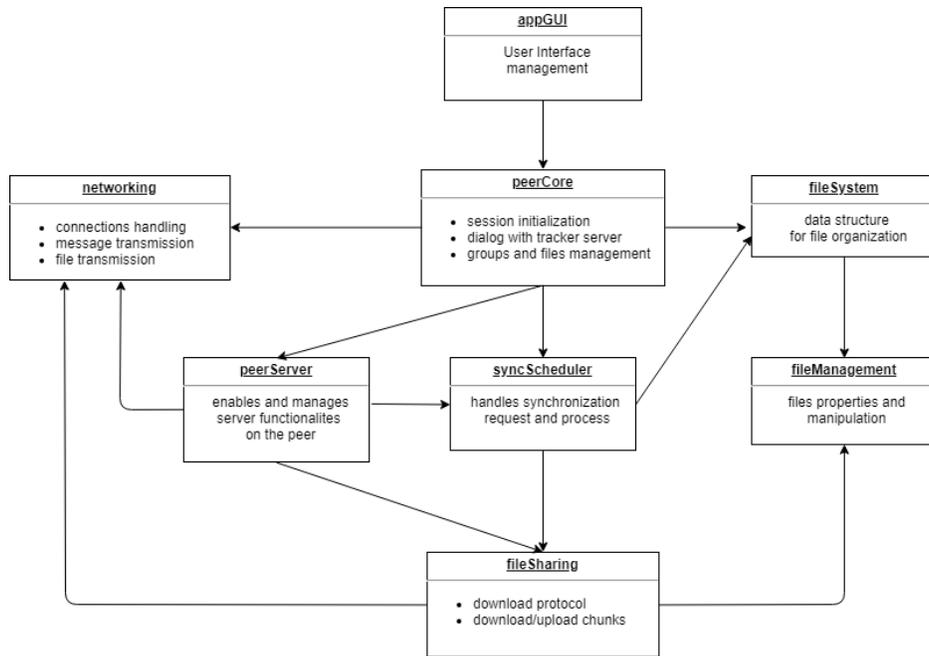


Figure 5.2: Organization of the client application modules.

5.2.6 Peer modules: myP2PSyncClient.py

This module creates and manages the graphical user interface of the myP2PSync client used by the user. Since the UI is written using PyQt5, which is a binding for the Python language of the famous cross-platform Qt framework, it is based on an events system. The code in the main of the myP2PSyncClient, which is actually the script to interpret to run the myP2PSync client, is as follows:

```

1  # declare the GUI application
2  app = QApplication([])
3
4  # create the window object
5  window = myP2PSync()
6
7  # start the application
8  sys.exit(app.exec_())

```

It first creates the application object, i.e. the event handler and the window object associated with it. Finally, it starts the main process of the application.

When the window is initialized, the main structural parameters, such as title, size, position and layout, are configured. All the various components and widgets that will be contained in the window, such as labels, items list and buttons, are also defined. Then the various widgets are placed in the window, using various nested layouts and a splitter, which divides the window into two sections. Immediately after initialization the user can see a screen like this:

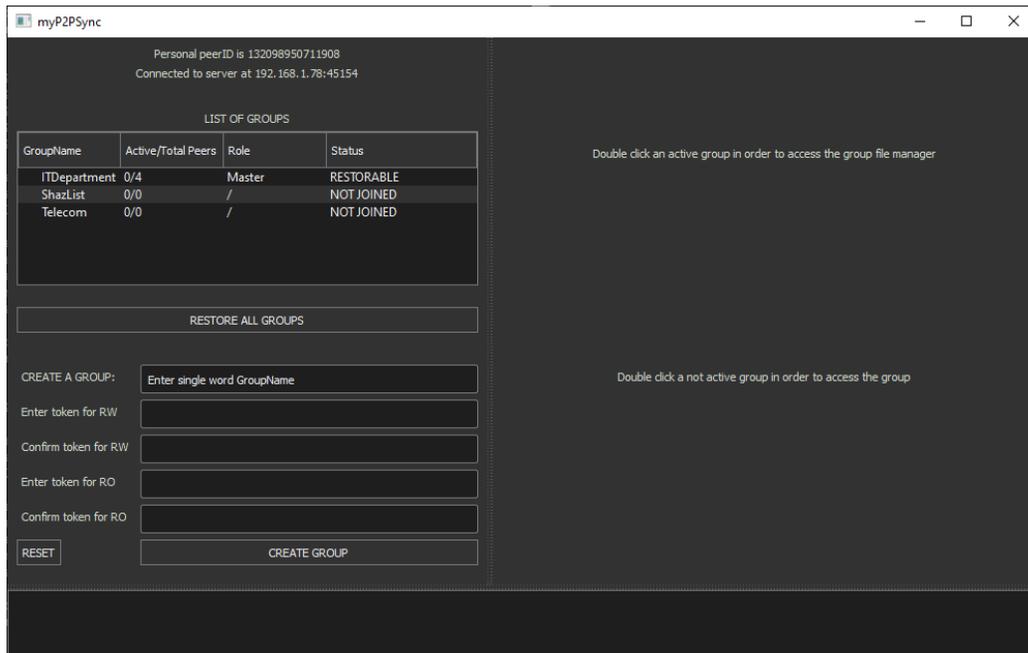


Figure 5.3: Example of client application window on startup.

The left section represents the GroupManager, and manages the operations of join, restore and creation of a group. The right section is called File-Manager and allows users to operate within a specific group, among those currently active. In the screenshot it is not active and contains only some info because no group is actually active. Finally below we have a section used to show to users messages about what happens on their group and files.

To access or restore a group users need to double click on the group name within the list. Access to a group will require the token via pop-up. After a group is reactivated or a new group is accessed, i.e. when a group with 'ACTIVE' status is showed in the group list, users can access the file manager of that group by simply double-clicking on the group name. At this point the FileManager will be active and will allow the peer different operations, depending on the role in the group:

- a ReadOnly (RO) peer will only be able to see the group files and will have access to the 'disconnect group' and 'leave group' buttons;
- a Read&Write (RW) peer can in addition add, remove and synchronize its files with the rest of the group;
- group master, in addition to RW privileges, has the possibility to change the role of other peers in the group, being also able to pass the role of master or to add another master.

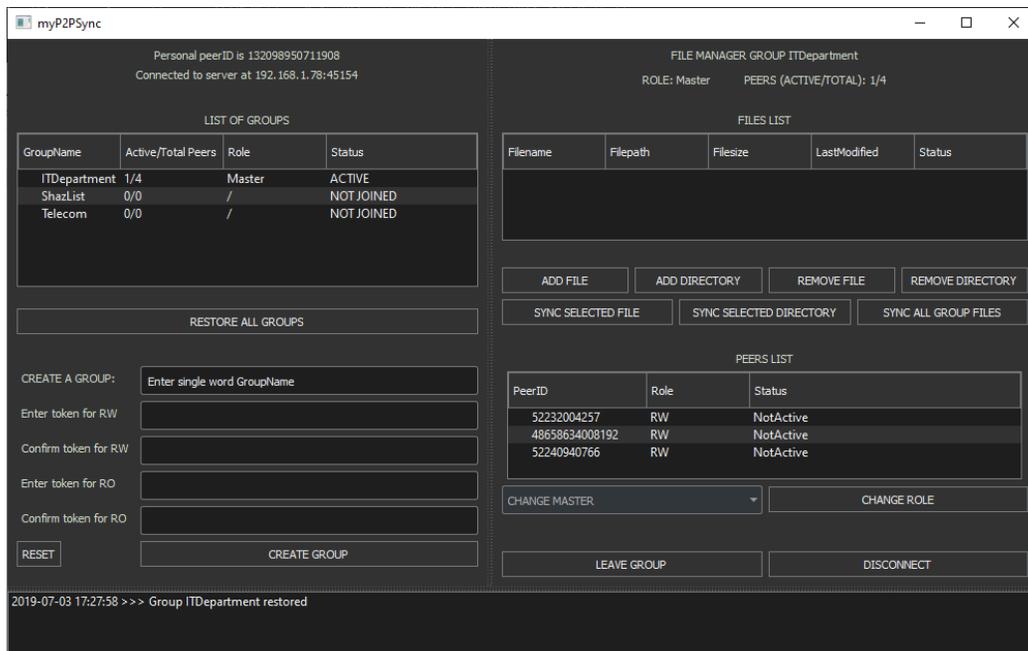


Figure 5.4: Example of client application window for a group master user.

Please refer to the functions `__init__` and `initUI` within the `myP2PSync` class for more details about widgets and layouts used.

As the system is based on event management, actions performed on a widget can generate such events. To allow the main thread to react to a specific event occurrence, it must assign an handler function called 'Slot' to the event signal. For example:

```
1 createGroupButton.clicked.connect(createGroupHandler)
```

This instruction associates the 'single click' event on the CREATE GROUP button, which generates a certain signal, to a `createGroupHandler` method. Each time the event is generated, this function will be executed.

The event mechanism can also be used for tasks not directly related to widgets. The fact that groups, peers and files are dynamic entities makes it necessary to periodically update the graphical interface by asking the server for updated information on the status of the system. In `myP2PSync` a refresh signal is used, emitted by a thread that runs in the background for as long as the user interface is active. It just sleeps for a certain number of seconds, in the current implementation 10, and then wakes up, emits this refresh signal and goes back to sleep. Actually it does not sleep for 10 seconds in a row, but it wakes up every second to check if the application is closed and to be able to finish releasing its resources correctly. The refresh signal is intercepted as any other event from the main thread of the application and is executed a handler function that requests updated information to the server about groups and peers, then updating the content of the GUI.

This script mainly interacts with the `peerCore` script in order to access the functions for the dialog with the server, to take advantage of the main data structures and parameters or even to configure some information, such as the IP address of the server. In fact, it is entered manually by the user through the GUI in case of failure or incorrect configuration of the file `configuration.json`, where this address should be written.

5.2.7 Peer modules: peerCore.py

This module contains all the functions necessary to interact with the tracker server. They are called by the handlers defined in the GUI. For example, the handler that manages the creation of a group is responsible for verifying the values entered by the user e.g. group name, tokens and then calls the function within peerCore that communicates to the tracker the creation of the group and updates local data structure. The return value of these functions is then used by the module myP2PSyncClient to generate any successful or error messages. The script also manages some initialization functions, such as the configuration of the peerID, obtained from the MAC address of the machine, the retrieval of the server coordinates from a configuration file and the tracker reachability verification.

The peerCore module contains and manages the two data structures:

- `groupsList`: Python dictionary for the various groups of the single peer. It contains information on the status of the groups i.e. role of the peer, its status (active or not), number of active peers and total number of peers.
- `localFileTree`: is a tree structure that keeps track of the status of files and their organization. For more information, refer to the `fileSystem` module.

An important operation that is carried out within the peerCore module is the initialization of the files of a group. The system manages files with messages between peers: for example, when a peer adds a file to a group or removes it or loads a new version, it sends a notification to all active peers who can react accordingly. It also records the change on the server. However, some peers may be offline i.e. they are not using myP2PSync or are disconnected from the group or have abandoned it. In the last case they may re-access the group in the future. In each of these cases the notification is not received and therefore the appropriate action is not handled. The problem is solved by asking the tracker for the updated file list for that specific group every time a group is restored or the peer accesses it. Peers keep their own list of files, updated at the last moment the peer was active in the group. If the peer has

never been active in the group, this list is empty. This list is compared with that of the server that is assumed to be always valid and updated because all changes are reported to him and the server can never be offline. In this way a peer can find out:

- that new files have been added and start synchronizing them;
- that some files have been removed and then handle the situation locally by removing them from its list of files;
- that new versions are available for some files it already has and then request the new version through a file synchronization.

For more details, refer to the `updateLocalGroupTree` function.

5.2.8 Peer modules: `fileManagement.py`

This module contains a class for saving information about files and manipulating them. The `File` class has the following properties:

- `groupName`, represents the name of the group in which the file was added;
- `treePath`, is the name of the file if it's taken into account any directory to which it belongs if they are also added to the synchronization of the group e.g. `dir1/dir2/file.txt`;
- `filename`, is the real name of the file e.g. `file.txt`;
- `filepath`, represents the path in the user's file system where the file is located;
- `filesize`, is the size in bytes of the file;
- `timestamp`, represents the version of the file. It is obtained from the operating system last modified timestamp, so it can be compared to it to see if the local version is synchronized with the group version;

- `status`, can be 'S' or 'D', depending on whether the file version is the latest available ('Synchronized') or not ('Download');
- `previousChunks`, is a list of chunkIDs, i.e. identifiers of parts of a file. It is used to save the status of partial downloads in order to resume them;
- `lastChunkSize`, is the size in bytes of the last chunk that can have different size from the others that have fixed size. All the other chunks have a fixed size of 1 MB;
- `chunksNumber`, represents the total number of parts in which a file is divided during a synchronization;
- `missingChunks`, is the list of missing chunks to complete the download of a file;
- `availableChunks`, is the list of chunks owned by the peer. A download is completed when `availableChunks` contains `chunksNumber` elements and `missingChunks` is empty;
- `progress`, which is simply the percentage ratio of the number of chunks owned respect to the total number of chunks. It indicates the download progress status. At the end of the download it is 100%;
- `syncLock`, is a useful `Lock` object for thread synchronization, it prevents that there are for example unforeseen changes on other properties during a synchronization;
- `stopSync`, is a boolean value that is used to block the synchronization of the file in case of errors or other situations that do not include the completion of the download.

The class has methods for configuring these properties. For example, the `initSync` method sets the various properties useful for file sharing, such as the size of the chunks and the various lists. Other methods use the `os` module to retrieve information such as the timestamp and file size directly from the file system. These methods use the `filepath` property to reach the file.

A user can add individual files or entire directories to the synchronization of a group. The latter are not associated with any File type object and are not directly subject to any synchronization operation. They are an abstract concept, in fact they only serve to keep track of the files organization. Adding a directory simply means adding all the files it contains, but taking into account the fact that all of them are part of the same directory. The `treePath` property is useful for storing the location of the file in relation to a directory that has been added to the group. For example: the user adds the entire `dir1` directory, which contains a second `dir2` directory, which in turn contains the file `file.txt`. In this case the `treePath` of the file is `dir1/dir2/file.txt`. In the case of adding a single file, the `treePath` property is equal to the `filename` property. The `myP2PSync` server is unaware of the organization of files in directories. It considers the `treePath` as the filename of the file.

The filepath is set in two different ways:

- in the case of the peer that adds the file it will be equal to the filepath of the file in the user's file system.
- if another peer has added the file, the filepath will be created automatically by the application, according to the rule:

$$\text{filepath} = \text{pathToScripts}/\text{filesSync}/\text{groupName}/\text{treePath}$$

- `pathToScripts` is the path in the user's file system where the scripts of the `myP2PSync` client are saved, except for the script names; e.g. `C:/Programs/myP2PSync/peerApplication/`
- `filesSync` is a directory automatically created by the application to contain all files in synchronization;
- `groupName` is simply the name of the group to which the file belongs, the application automatically creates a directory with this name.

In the current implementation, a `treePath` cannot contain spaces, i.e. the filename and names of all directories, if any, must be free of spaces.

5.2.9 Peer modules: `fileSystem.py`

The files synchronized with `myP2PSync` have the following features:

- can belong to different groups;
- can belong to a folder, which is also synchronized;
- because peers can use more than one group at the same time, the same file can belong to more than one group;
- within the same group there can be no files of the same name, except in different folders. The `treePath` property must be different in order to distinguish them univocally;
- are associated with an object of the same File type as the one defined in the previous paragraph.

Organizing and keeping track of files is therefore an important issue, considering that information about a file must be retrieved efficiently. Choosing the right data structure was therefore subject to change during application development. The first choice was to use a Python dictionary, i.e. an hash table. This data structure has key-value pairs and in the implementation of the `myP2PSync` file system they had a structure like the following:

```
key: 'groupName_treePath' string
value : File object
```

Unfortunately, although it is very efficient for retrieving file information, it is not very compatible with creating the list of files in the graphical user interface. This list must keep track of the organization of files in directories, as indicated by the `treePath` property of the files, and is therefore a tree structure. Consequence of this incompatibility is an exaggeratedly large time in the rendering of the files list in the GUI due to the complexity of the algorithm to create it. For example: to create the entry in the file list associated to the file with `treePath` equal to `dir1/dir2/file.txt` in an already partially filled file list you must proceed iteratively as follows:

1. is considered the first part of the treePath: dir1. We evaluate if an entry for dir1 is present in the file list, operation with cost $O(n)$ in the number of entries already present.
2. If it is not present, it is added to the first level; if it is present, check that it is at the first level.
3. The same is done for dir2. This time if the entry exists, check that it is at the second level and that the first level is equal to the entry dir1. Otherwise you have to create the entry dir2 under the entry dir1.

The complexity of this approach is due to the fact that the class that implements the files list, a `QTreeWidgetItem`, does not allow you to search for a certain entry at a certain level, but only in the entire set of entries. It is then necessary to manually check the depth level going backwards. In other words, if I look for the entry dir1 and there is a match, the entry is not necessarily the correct one, it could be an entry named dir1 under an entry with another name. The series of searches and checks is really time consuming, and is not feasible for a large number of files, especially in the case of numerous nested folders.

So we opted for a second tree data structure, which recalls that of a file system, hence the name of the module. It follows a scheme like the one shown in the figure:

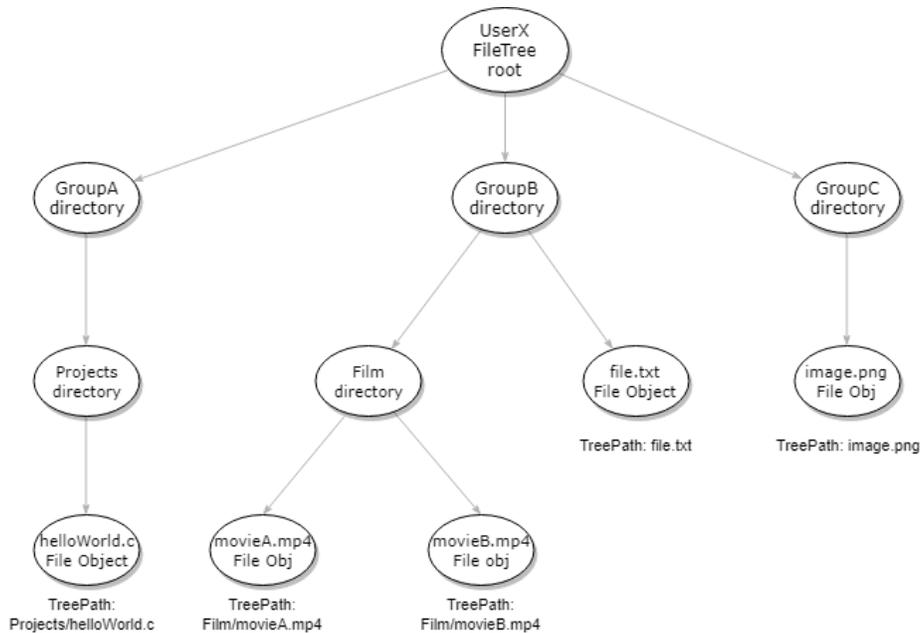


Figure 5.5: Architecture of the myP2PSync file system.

The tree root is a FileTree type object that keeps track of all groups. Each group is associated with a directory root node, from which the structure of all the files in the group is developed. Each node contains different information:

- the name of the node, equivalent to the name of the directory or the name of the file;
- a Boolean value that indicates if the node is a directory or a file;
- a dictionary containing all child nodes;
- a File type object.

A node can be of two types:

- Directory node, has the childs dictionary not empty while the file object is empty. Any directory nodes with 0 children are removed from the structure;

- File node, has the children's dictionary empty being a leaf node but has a File object.

This structure makes the creation of the file list much faster because it is enough to recursively scroll the tree associated with a certain group generating the labels, distinguishing the entries of the directories from those of the files.

The module also provides the necessary methods to remove nodes, add them and search for the node associated with a file. These methods base the search on the `treePath` property of a file, using it to scroll through the tree to the node you are looking for.

In the `treePath` of a file node, the root node of the group is not taken into account because the search for a node takes place from the directory node of the group. For example: you want to delete the file with `treePath` `Film/-movieB.mp4`. The fact that this file is within `GroupB` is known to the application why:

- if it is the peer itself that wants to delete the file from the group, it has started removing the file from the `groupB` file manager within the GUI;
- if another peer requests its removal, it has sent the group name in the request.

At that point the peer knows `GroupB`, and looks for its node among the children of the `FileTree` root. Finally it follows `treePath` to reach the node and delete it.

This data structure is slightly less efficient than the hash table: we talk about an $O(\log N)$ cost in the search for a node, where N is the level of depth of the node searched, against the $O(1)$ obtained with the hash table.

The `fileSystem` module also contains the necessary functions:

- to save the current status of the `FileTree` in a JSON file;

- to the initialization of the FileTree starting from the information contained in the JSON file about a possible previous session.

5.2.10 Peer modules: peerServer.py

This module implements a server similar to the one already seen previously in the myP2PSyncTracker.py script. It is in fact a server that uses multi-threaded programming to serve multiple requests simultaneously. A peer needs this function to be able to respond to any requests from other peers, such as requests for chunks or messages about files in the group. An incoming message is then processed and the appropriate function to serve the request is called up. A connection, and with it the thread that manages it, ends when a 'BYE' message is received. The various messages that the peer receives concern:

- requests for chunks or a list of chunks for a certain file, managed by going to call the appropriate functions of the fileSharing module;
- additions or removals or the availability of new versions of files in a certain group. These situations are managed by taking advantage of the appropriate functions of the syncScheduler module.

5.2.11 Peer modules: syncScheduler.py

This module contains the synchronization processes scheduler. It is a thread that is created by the peerCore module within the startPeer function and that periodically, every second, checks if there are synchronizations that need to be executed. It is terminated when the application is closed. The application need this scheduler thread in order to limit the maximum number of simultaneously synchronizations. This last feature introduces a lot of synchronization issues that the scheduler address and solve.

The modules capable of adding a synchronization request are:

- peerCore when the peer itself adds new files or new versions. Refer to the addFiles and updateFiles functions of the module;
- the syncScheduler module itself, following messages received by peerServer from other peers communicating the addition of new files or new versions. Refer to the addedFiles and updatedFiles functions.

A synchronization request refers to a single file. It is an object of type syncTask, class defined in the module, which has as its property the group of the file, its treePath which is the path within the file structure to retrieve information about the file and the timestamp or version of the file. All tasks are stored in a FIFO structure, implemented using a deque i.e. double ended queue. New tasks are placed at the tail of the structure. The choice of the deque data structure is explained in section 6.1. When a task related to a new version of a file is added to the queue, the module verifies that this version is not obsolete because another peer may have sent the request to an even more recent version. In addition, the module also checks that the deque does not contain other tasks related to the same file with an older version. If they are present, they are deleted. If the task is added after adding a new file, this check is not performed because there can be no other tasks related to the same file.

The scheduler, which periodically checks the status of the deque in search of new tasks, if find one removes it from the head of the queue and then:

- it checks if the number of threads working on a synchronization is less than a certain number i.e. MAX_SYNC_THREAD, in the current implementation this parameter is equal to 5. It serves to avoid that there are too many synchronizations and therefore too many active downloads at the same time, which would slow down the overall progress of processes;
- it checks if the task is still valid, i.e. if the group it refers to is still active and the file is still present, as it may have been removed from other peers or from the peer itself in the meantime;
- it checks that there is no other synchronization in place on the same file, acting on a previous version. In this case the scheduler has to wait

for it to end up activating the new thread on the new version. To avoid the waiting process in this case the scheduler simply reinserts the task in the list.

If none of the conditions are met, the scheduler launches the synchronization thread:

```
1   syncThread = Thread(target=fileSharing.startFileSync ,  
                        args=(fileNode.file, task.timestamp))
```

The target parameter represents the starting function of the new thread while args represents its parameters, in this case a File object (see fileManagement) and the timestamp of the task.

The syncScheduler module also offers some useful methods to manipulate the deque, for example to remove tasks related to a certain group for example when it is no longer active. The management of active synchronization threads is also delegated to the module, as it keeps track of their information in a data structure. This information also includes the status of the thread. The list of possible states for an active synchronization thread is as follows:

- SYNC_RUNNING: The thread can continue to run;
- SYNC_SUCCESS: The thread has completed the synchronization process, i.e. it has successfully downloaded the file;
- SYNC_FAILED: Synchronization failed due to lack of resources e.g. lack of active peers;
- SYNC_STOPPED: Synchronization was blocked due to disconnection from the group;
- FILE_REMOVED: Synchronization was stopped after removal of the file;
- FILE_UPDATED: synchronization was stopped due to the arrival of a new version of the file.

These states are declared within the module in the form of numerical constants. They are used to communicate with the synchronization thread, it periodically checks its status in order to react to any external events.

5.2.12 Peer modules: fileSharing.py

This module contains all the functions necessary for a peer to share a file, both as an uploader and a downloader. The main and most relevant functions are related to the file download. They are described in detail in the paragraph 5.3. The rest of the functions are used by the peers for:

- request and send their own chunks lists to other users. A chunks list is the list of chunks that the peer has about a certain version of a certain file. The chunks list is the list of chunks that the peer has relative to a certain version of a certain file.
- the operation of merging the chunks into a single file, once they have all been retrieved. It must also handle the possibility of an empty file, which must still be created even without receiving chunks.

When downloading a file, the various chunks are stored by the peer in a temporary folder. This folder is present at the same location of the file and has the same name followed by `'_tmp'`. The various chunks are represented by small files named as the chunk identifier e.g. `chunk13`. Finally, they are merged into a single file with a temporary name, which is the one of the original file followed by `'_new'`. For example, the new version of a file on the path `'C:/Users/abc.txt'` will be rebuilt in `'C:/Users/abc_new.txt'`. If the merge operation is successful, the old file, if any, is removed and the new file is renamed. Also the time of last modification to the file is modified, going to place the one communicated by the source of the synchronization, that is the peer that added the file or its new version. This is necessary because otherwise the timestamp of the file would be greater than the one communicated because some time has passed, however little it may be. If the timestamp is higher, the new peer could send an update request to the other peers without really having a modified version of the file.

Considering the chunks sending operation, we can distinguish between two cases:

- the peer has already finished synchronizing the file or is the one who added the file or its new version. In other words, it has the full version and behaves like a seed. In this case the chunks are retrieved directly from the file, through a positioning operation in the chunk location and with a reading operation;
- the peer is still in a phase of downloading the file, so it does not have a full version, but only splitted chunks. The required chunks are then recovered from the temporary directory.

The positioning operation is performed with the seek function that allows to move the pointer to the file. The offset given as parameter is obtainable as:

$$\text{offset} = \text{chunkID} * \text{CHUNK_SIZE}$$

5.3 File-sharing protocol

This section contains a detailed description of the protocol used by myP2PSync for sharing files in synchronization. It explains the approach used, the algorithm used to retrieve a file from other peers in the same group and illustrates some significant parameters.

5.3.1 P2P Approach

The file-sharing mechanism used in myP2PSync is based on a Peer-to-Peer protocol. This means that there is not a single node on the network where all peers request a file, as in the case of a client-server approach, but rather the file can be requested from several other peers, i.e. nodes on the network that play the same role. The protocol used is based on the well-known BitTorrent

which is also a protocol for P2P file-sharing. The common points between the myP2PSync protocol and BitTorrent are as follows:

- the files are divided into smaller pieces of fixed size;
- all peers are asked which chunks have of a certain file;
- the pieces of a file are requested in order of rarity (rarest-first approach), i.e. a piece that is owned by a few other peers compared to a more common one is first requested.

The myP2PSync protocol lacks the concept of choking, which consists in favoring peers who are using chunks of a file at a great speed over others who behave like leechers, downloading much more than they upload. This approach has no reason to be used in myP2PSync because the purpose of the application is only to spread as quickly as possible the new version of a file, without discriminating between different users. Think in fact of the domestic case in which a user simply wants to distribute a file to all his devices: it would not make sense to slow down the transmission of the file to a specific machine because they all belong to the same user. The same reasoning can be applied to the business case: the ultimate goal is to distribute a new version of a file to all employees as quickly as possible, it would not make sense to slow down the transfer to some machines just because they are more limited in terms of upload speed.

5.3.2 Chunks size

Regarding the size of the chunks, myP2PSync uses a fixed size of 1 MByte, large enough to avoid excessive segmentation of large files and small enough to avoid large losses of efficiency and time in case of necessary retransmission of a chunk following an error. Initially a more dynamic approach was used, i.e. fixing the size of the chunks according to the size of the file. This approach was not successful because it made the synchronization of small files really too slow, being the chunks really too small. Using a fixed size all chunks have the same size, only the last one could be smaller, exactly like in the BitTorrent protocol.

5.3.3 File-sharing algorithm

The file-sharing protocol follows a multi-threading approach, to try to make the most of the simultaneous online presence of a multitude of peers. The download of a file is implemented through three different types of threads:

- the first thread, see `downloadFile` function in the `fileSharing` module, is responsible for managing the creation of all other threads and the operations of initialization and termination of the download. It instantiates the `chunksManager` thread and all `getChunks` threads. The maximum number of instantiable `getChunks` threads is defined by the constant `MAX_THREADS` in order to limit the number of coexisting threads. It is well known that an excessive number of threads working in parallel tends to deteriorate performance. This thread creates a `getChunks` thread every time a new peer is active and you can ask it for chunks of a file.
- a second thread, called `chunksManager`, periodically calculates the `rarestFirstChunksList`, that is the list of missing chunks sorted by rarity. It:
 1. gets from the tracker the list of active peers in the group;
 2. if the number of active peers exceeds a certain constant `MAX_PEERS` considers only a subset, which varies with each iteration through a shuffle operation on the list of peers obtained;
 3. requests each selected peer its own list of chunks for the file to be obtained;
 4. using these lists calculates the list of missing chunks sorted by increasing rarity; finally places itself in a waiting state for `REFRESH_LIST_PERIOD` seconds.
- the last type of threads are called `getChunks` threads, i.e. those that actually request chunks from other active peers. Each different thread requires chunks from a single other peer. The chunks to be requested are taken from the `rarestFirstChunksList` generated by the `chunkManager`. Of course, the various threads do not require the same chunks and can only require chunks that the peer with whom they communicate has previously notified that they have. The connection with the

remote peer is created immediately after the thread instantiation and is closed only at the end of the download or when the remote peer disconnects. In case of continuous errors on the connection, it is closed and re-established. These threads work in the following way:

1. extract from the `rarestFirstChunksList` a list of chunks to be requested with a maximum length of `MAX_CHUNKS`. This parameter favors the distribution of the workload between the threads.
2. They require all selected chunks and save them locally. In case of an error on a single chunks they reinsert it in the shared list so that it can be reconsidered later, also by other `getChunks` threads.
3. They repeat the previous steps until the end of the download or until they realize that the remote peer is no longer active.

The three types of threads coordinate and communicate through a shared data structure. It contains the `rarestFirstChunksList`, the list of active peers and the download status. Once the download is complete, the `getChunks` and `chunksManager` threads end, while the main thread handles the merging and closing operations of the download. At this point the download process of the file is complete and the synchronization ends.

It is important to note that the timestamp of the obtained file is of course subsequent to the desired one, i.e. the one transmitted by the peer that sent the update message, because it corresponds to the instant when the last chunks was written inside. For this reason, the timestamp is forced to the value transmitted by the following instruction which modifies the timestamp directly in the OS file-system node:

```
1 os.utime(file.filepath, file.timestamp)
```

At this point, if the peer tries to force an update of the file, the application would compare the timestamp of the file system with that defined in the data structure of the file, they would be the same and it would not allow the operation.

5.3.4 Random discard approach

At the beginning of a file download it is important to consider a random factor in the choice of chunks to request. If we consider the case in which a peer adds a file to the group and all the others try to get it, it is likely that they will have the same `rarestFirstChunksList` and request the same chunks, not taking good advantage of the crowd of peers. Adding instead a more random choice of chunks to request can improve the spread of chunks between peers. The `getChunks` threads, at the beginning of the download, therefore tend to add a chunks to their list of requests with a certain probability equals to `INITIAL_TRESHOLD`. As the download proceeds, this probability is increased by a `TRESHOLD_INC_STEP` factor. Finally, at the end of the download, when the progress is greater than a certain amount of `COMPLETION_RATE`, all chunks are requested without applying the random approach in order to complete the synchronization as soon as possible.

5.3.5 Synchronization stopped or failed

A synchronization can be interrupted for several reasons:

- the file is removed from the group;
- another version of the file is available and the current synchronization is outdated;
- file download cannot be completed due to lack of resources:
 - there are no other active peers in the group;
 - other active peers do not have the missing chunks of a file;
 - connection with other peers fails.

To manage these situations, each download is associated to a thread that checks the status of the operation. It periodically checks the status of the thread through the `syncThread` data structure managed by the scheduler module and reacts to the stop operation by closing the download cleanly. Clean closing means:

- saving the download status, i.e. it stores the list of chunks obtained in order to be able to restart the synchronization in the future from the point where it stopped.
- the release of synchronization locks, so that can trigger any other synchronization operations on the same file, perhaps following an update request received from another peer.
- the blocking of any still active threads related to the download.

Three different synchronization termination scenarios can be defined and evaluated:

- The first is when the download finishes correctly and the file is reconstructed without errors (`SYNC_SUCCESS`). In this case the chunks lists are simply updated as follows: `missingChunks` and `previousChunks` are reinitialized to empty lists, `availableChunks` contains all chunks.
- The second occurs when the file download can not complete due to lack of resources availability (`SYNC_FAILED`) or when the download is interrupted by leaving or disconnecting from the group (`SYNC_STOPPED`). In these cases the `availableChunks` list is copied to `previousChunks` so you can resume synchronization from the current state later. In fact, the synchronization is reschedule if in the meantime a new version has not arrived or if the file has not been removed.
- Finally there is the case where the synchronization is blocked by an external request, such as a new update (`FILE_UPDATED`) or removal of the same file (`FILE_REMOVED`). In these cases the thread notifies the closure of the download to all the other threads that can be still in progress, that are the main thread and those that may be requesting chunks in order to stop them. All lists of chunks are reset because they are now useless or obsolete.

The block of possible threads still active is managed by a boolean value contained in the File object i.e. `stopSync` property. At the beginning of the synchronization this value is placed equal to `False` and if an event occurs

that requires the manual termination of the threads it is placed equal to True. This value is checked:

- periodically from the download thread ;
- from the chunksManager thread between two update operations on the chunksList;
- from each thread requesting chunks, just before requesting each single chunk.

Throughout the synchronization process a lock is acquired on the file. It is only released after the operation has been closed. This ensures that the parameters of the file, such as its size, are not changed by other threads during the synchronization process, for example following an update request from another peer. These requests will then have to wait until the end of the synchronization and, if still valid depending on the timestamp, will be processed. For example: when synchronizing a file with timestamp 1000, an update with timestamp 2000 is received. This causes the download of the file to be blocked, but this may take some time as the check is periodic. During this time the request cannot be served and is put into effect. If we consider the case in which during this time a third update request with timestamp 3000 arrives it would be the only one to be served, because the second one is already obsolete.

5.4 Devices communication

This section discusses the solutions used for communication between the various devices, going into detail how connections are established and how the various messages are encoded, sent, received and decoded.

5.4.1 Choice of the communication protocol

Since myP2PSync is a distributed system, one of the first questions that need to be asked in the design phase was how to make peers and tracker server able to communicate. A first answer was with the HTTP protocol which is generally well supported by the libraries of any programming language, it is very easy to interpret but being an application level protocol very well defined and with its own rules it probably would have led to a lack of flexibility in writing the application. Furthermore, by using HTTP you get an overhead of information transmitted as each time the HTTP header would be sent and this can be avoided with a lower level approach. The second and last approach considered and implemented is a lower level solution by going to program a small transport protocol directly to the socket level. A socket is nothing more than the abstraction of a network access point through which a machine can communicate with another. In other words, if two machines have a socket and a connection is created between these two sockets, the two machines can exchange messages and data. The use of sockets guarantees an excellent flexibility in deciding how the machines should communicate, since the programmer has to take care of different aspects such as:

- opening and closing the communication;
- low level characteristics like for example which transport level protocol to use;
- guaranteeing a certain reliability to the communication.

More specifically, stream sockets are used. They provided a connection-oriented and sequential communication with well-defined mechanisms for creating and destroying connections and for detecting errors. A stream socket transmits data reliably, in order, and with out-of-band capabilities. Stream sockets are typically implemented on top of TCP so that applications can run across any networks using TCP/IP protocol. The characteristic of being connection-oriented was the cornerstone of the choice. This allows to have reliable connections even in case of fast exchanges of messages as happens during file-sharing.

One thing that should be taken into account when programming sockets and exchanging messages is that establishing a connection is quite time consuming, so you need to minimize the number of times connections are created. Unfortunately, it is also unthinkable to establish a connection and leave it active for as long as necessary, for example for the entire session using myP2PSync. In the application we have therefore tried to exploit the single connections to the maximum trying to group the messages as much as possible for example making sure that more chunks of the same file are requested within the same connection. We also tried to send the minimum number of messages possible, even at the cost of sending large messages, favored by the fact that the communication protocol used is not sensitive to the size of the message. For example, when a peer adds a directory to the synchronization group, it sends a single message to the server and the other peers containing the complete list of added files, and not a different messages for each file.

Next paragraphs presents how the peers and the tracker server manage the creation of connections, starting from the latter as it is simpler. For any doubts or other information about socket programming, please refer to the official documentation of the Python3 socket module, which can be reached at the following URL:

<https://docs.python.org/3/library/socket.html>

5.4.2 Tracker side

The tracker server just creates its own TCP socket stream and associates it with its own IP and a specific port number, the 45154. This last operation is called socket binding because it binds the socket to a unique address consisting of the tuple (IP address, port number). The port number was arbitrarily chosen from those not registered for any application within the registers of the IANA (Internet Assigned Numbers Authority), which is the association that allocates IP addresses and port numbers to various applicants. Immediately after the binding, the tracker server puts itself in a listening state, which means it is ready to accept new incoming connection requests that will come from the various clients. The listening socket is called a passive socket.

Whenever a connection request is detected, the tracker creates a connected copy of the passive socket that is connected to the the client socket. This connection can now be used for communication.

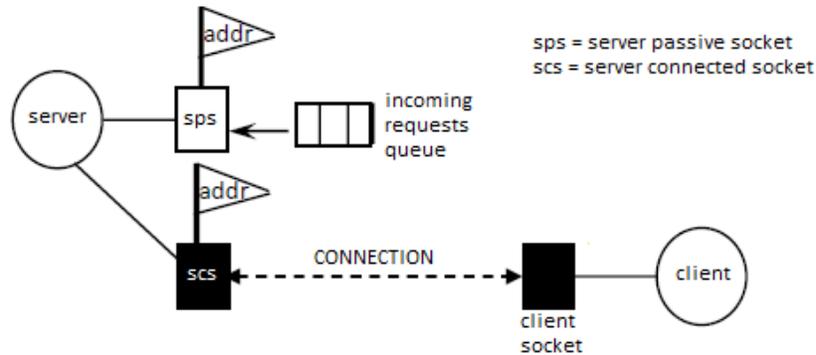


Figure 5.6: Socket connection establishment.

This is the portion of code that does the operations described above:

```

1 # declares the socket type TCP stream
2 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3 # Associates the socket with the chosen IP address and
  port number
4 sock.bind((IPAddress, 45154))
5 # Put the socket in a listening state
6 sock.listen()
7
8 # until the server is stopped accepts new requests by
  creating a new connected socket
9 while not serverStop:
10     clientSock, clientAddr = sock.accept()

```

At this point it's possible to communicate using `clientSock`, while the variable `clientAddr` contains the IP address of the client. Other information such as the port number of the client can be retrieved directly from the `clientSock` object. Now that the connection is established client and server can talk. Usually it is the client who speaks first so the server puts itself in a blocking state of reception. Once the message has arrived, the request is processed

and a response is provided to the client. When the message exchange is finished the client sends a 'BYE' message and the tracker server closes the connection and the socket.

The server uses multi-threading, which means that every time an incoming connection is detected and the copy of the passive socket is generated and connected, a new thread is also instantiated with the unique purpose of exchanging messages on the new connection until it is closed. We can think of these threads as disposable objects: once their task is finished they simply end after releasing the socket resources. The main thread, the one waiting on the accept function, does not respond directly to requests, but it allocates a new thread that will do so. This feature makes the server much more performing and responsive. In the case of single thread solutions it is not possible to serve a new request until the previous one is completely finished. In fact, the thread would be engaged in another and would not be stuck on the accept function, ready to detect new requests.

5.4.3 Peer side

Half of the network programming structure of the peers is similar to that of the tracker server. In fact, a peer works by definition both as a client and as a server. To work as a server, it follows exactly the same scheme described above with just one difference: the binding of the listening socket is not done on a predetermined port as in the case of the server that always runs on port 45154 but it's done on the first port found free by the operating system. This port discovery is performed simply using 0 as the port number during the bind operation. The port number actually chosen by the OS can then be retrieved directly from the socket object.

```
1 # using port = 0 the server will start on an available
   port
2 sock.bind((peerIPAddress, 0))
3
4 # retrieve selected port
5 port = sock.getsockname()[1]
```

This port number must then be communicated by the peer to the tracker so

that other peers can reach it. This is done immediately after the peerServer is created by sending a 'HERE' message containing the IP address and port number on which the peer is listening. When the peer has to work as a client, for example to talk to the server or to make requests to the peers the situation is slightly different. There is no need to listen on your socket, but simply to launch a connection request to the listening socket of the destination machine, be it the tracker or another peer. In the first case the tuple (IPAddress, PortNumber) is read from a configuration file or manually entered by the user. In the second case, instead, the coordinates of the active peers are obtained by asking the tracker server first. Following a connection attempt, the socket module automatically manages the initial handshake and if all goes well, the client socket can now be used to send and receive data. When the peer decides to close the connection, it sends a message 'BYE' which makes the destination understand that the exchange of messages is over. At this point both can close their sockets releasing the resources. The following image summarizes the whole mechanism of creating a socket connection between two machines, send and receive messages and finally close the connection.

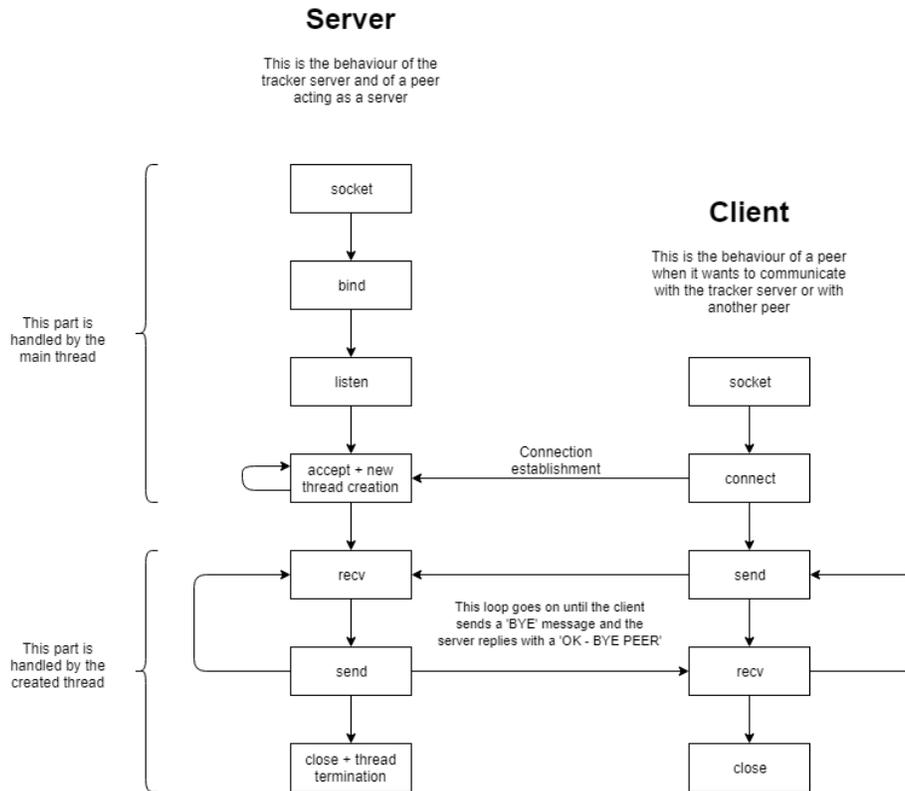


Figure 5.7: Communication on a socket connection.

5.4.4 Messages exchange

Using the Python3 socket module it's possible to exchange messages in the form of an array of bytes. Two types of messages are exchanged in myP2PSync:

- requests in the form of a string string, for example: JOIN <group-Name> <token> or CHUNKS_LIST <filename> <timestamp>
- file chunks, which in the case of both text and binary files are exchanged as byte sequences.

In the first case string messages, which in Python are Unicode code points sequences, must be converted to byte sequences using a certain encoding e.g. UTF-8, latin-1 before they are transmitted and decoded from the destination. Fortunately, encoding and decoding methods are already provided by the Python String class. The chunks of a file, on the other hand, do not need any conversion since they are already arrays of bytes and can be transmitted and received as they are. It is important to note that the send method of the socket module in python3 gives no guarantee that all bytes will be sent, but returns the number of bytes actually sent. So a wrapper function was built: it writes on the connection iterating until the entire message has been sent. In reception another wrapper function was built in order to read from the connection. It reads any pieces one at a time and reconstructs the entire message. Unfortunately in reception we do not know a priori how much we will have to read. The solution adopted for this problem consists in having the sender send the number of bytes that make up the message first and then the message itself. The length must be written on a number of bytes that both sender and recipient know. In myP2PSync this number is 16 bytes. For instance: `message = "RESTORE GROUP1"`, with `length = 16`. The sender writes 16 in a 16 bytes string: `"0000000000000016"` and sends it. The recipient iterates on the reception until the reading of the 16th bytes. At this point it decodes the string and gets the length of the message it is about to receive. Knowing this length it can iterate on the reception until the complete reception of all the bytes. Pieces must also be rejoined.

To send a text message, which can also be the casted to string version of a data structure such as a list or a dictionary that does not contain objects, the following steps should be executed:

1. the message is encoded using latin-1 codes;
2. the length of the message is extracted and it is placed in a 16 byte string which is in turn encoded in latin-1;
3. size is sent iterating up to 16 bytes;
4. message is sent iterating until all the bytes have been transmitted.

To receive a message, the destination follows the steps below:

1. it iterates on reception until 16 bytes have been read;
2. eventual pieces are rejoined and decoded to obtain the actual length of the message;
3. it iterates on reception until all the bytes of the message have been received;
4. it reunites any pieces of the message and decode them, obtaining the original message.

This small sending and receiving protocol is used by both the peers and the tracker server. A very similar approach is also used to send file chunks. The difference is that both the transmitting and the receiving peers know the length of the chunk: they have a fixed length, only the last one can have a different size but it is known anyway. So there is no need to send the length of the chunk but it's possible to directly send the sequence of bytes always iterating in order to avoid partial transmissions, even considering that we are talking about chunks of size of 1 MB, which are then easily segmented. Indeed, considering the documentation of the socket module are usually sent maximum 4096 bytes at a time. In reception simply iterates on reading and rejoin the various pieces. No encoding or decoding operation is necessary as already mentioned above.

The implementation of this protocol is present in the networking module that both the peers and the tracker server use.

5.4.5 Message format

The format of the messages is the following for both the exchange of messages between peers and for the exchange between a peer and the server. In order to make a request a message of the type “<peerID> <request> <param-1> <param-2> ... <param-n> “ is sent. The field <request> indicates the purpose of the message. Examples of <request> can be "JOIN", "RESTORE", "LEAVE", "GET_FILES", "ADDED_FILES", "CHUNKS_LIST". The <peerID> field identifies the sender peer and the various parameters give information such as the name of the group to which you want to register and

the access token. The answers have instead the following format "OK/ERROR - INFORMATION STRING". That is, the first word tells us if the request was served successfully or not, while the Information String gives us more explanation, for example by saying the error that occurred. In some particular cases the Information String contains data such as the list of active peers in a certain group or a list of groups registered on the server. Appendix of this document contains the complete list of requests that can be made.

5.5 Multi-threading architecture

As can be deduced from the previous paragraphs both applications are strongly based on the use of multi-threading as a tool to perform multiple tasks simultaneously. This can be trivial in the server application, as most servers nowadays use this technique to serve multiple requests in parallel. In the peer application instead the situation is a bit more complex because threads play an even more pervasive and dominant role, and above all they have a less regular pattern. This paragraph wants to highlight their role and how they interact.

The following figure summarizes the action flow of the peer side threads:

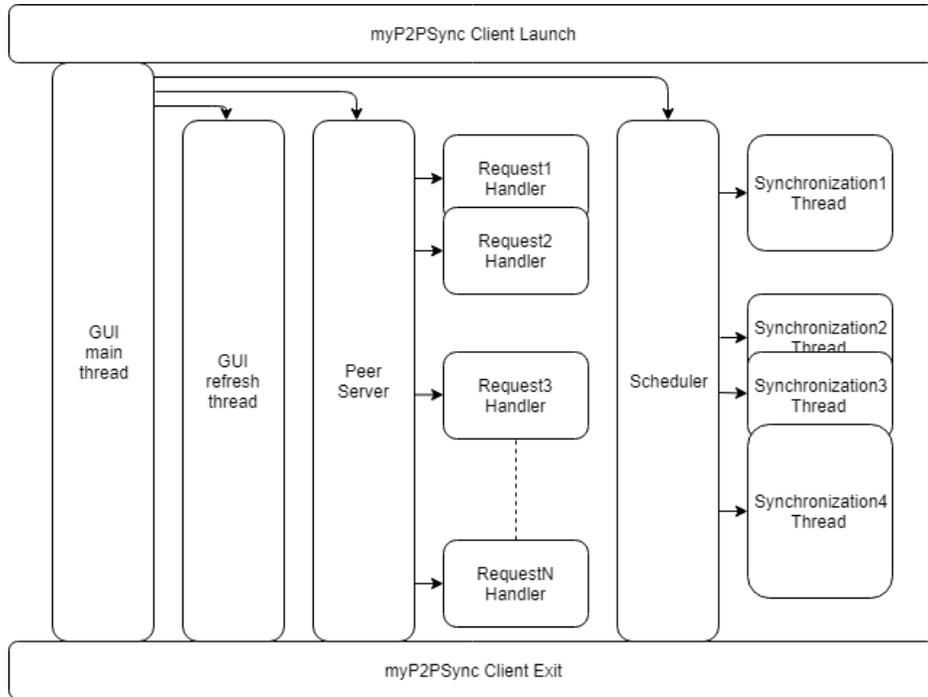


Figure 5.8: myP2PSync client application multi-threading architecture.

The main thread is the one handling the graphical interface: it is waiting for events that it deals with using proper handlers. It also takes charge of the initialization phase and of the dialogue with the server using the functions provided by the peerCore module. Immediately after the initialization phase it instantiates three other threads:

- the thread that manages the periodic refresh of the graphical interface;
- the thread that allows the peer to work in server mode, which in turn creates threads to serve the requests;
- the thread that manages the scheduler, it in turn creates threads to manage the synchronization of individual files.

In addition, each synchronization thread uses multi-threading: the main

thread, which was created by the scheduler, manages the operations of initialization, termination and verification of the synchronization status, in order to identify any external stop signals and manages them. It instantiates a second thread for downloading the file. This thread in turn creates a chunks-Manager thread which periodically obtains from the tracker the list of active peers and ask them for their list of chunks for the file. Furthermore the downloadFile thread manages other threads that request chunks from several peers in parallel. These threads are created as soon as a peer is notified as active by the chunksManager and are terminated as soon as it is no longer active or the download ends.

The following figure resumes the organization of threads that work in parallel in order to perform a synchronization task:

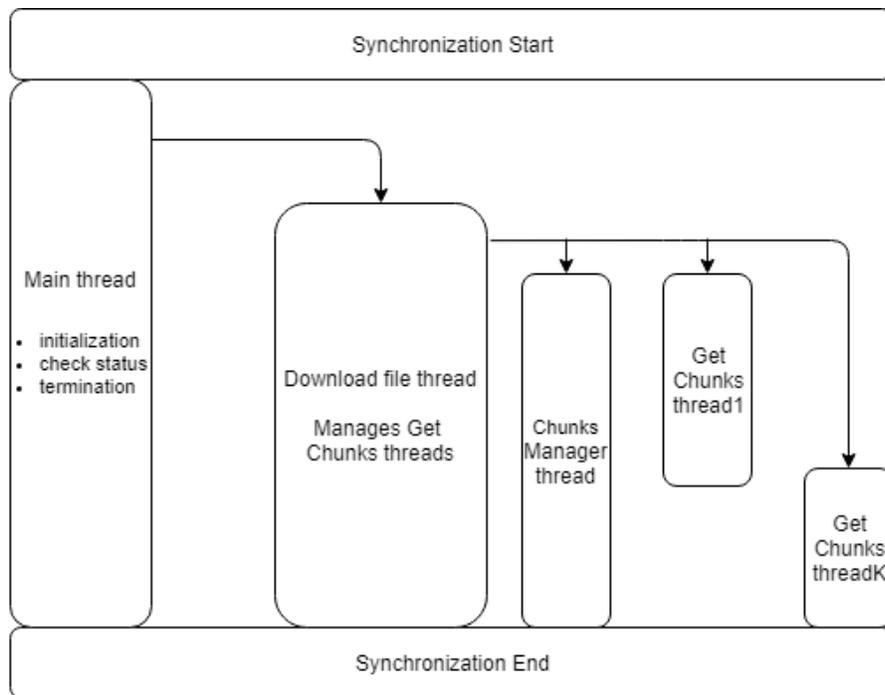


Figure 5.9: Organization of a synchronization multi-threading.

Finally, threads in Python3 can be configured as daemon. For example:

```
1   syncThread = Thread(target=fileSharing.startFileSync ,
2   args=(file , timestamp))
3   syncThread.daemon = True
4   syncThread.start()
```

This allows the parent thread to continue running without worrying about having to retrieve any return value from the child thread.

Chapter 6

Implementation choices

This chapter discusses and motivates some important implementation choices. They were not discussed in the previous chapter in order not to make the reading too long and heavy. Moreover, these implementation choices are often common to most software development projects, so they have been treated separately. The chapter is divided into the following paragraphs:

- 6.1 Data Structures
- 6.2 Server reachability
- 6.3 Previous session information

6.1 Data Structures

As is well known in the field of computer science, it is not only important to use efficient algorithms, but also data structures that work well with them. Since the system is written entirely in Python, the main focus has been on data structures offered by the language [11]:

- list, an array of elements which offer operations such as append and pop at cost $O(1)$. Both of these operations operate efficiently on the

tail of the data structure. All other operations, including inserting and removing elements in the head, as well as in any intermediate position, have cost $O(n)$.

- deque i.e. double ended queue, a data structure present in the collections module. It operates as a list but its implementation allows to have unitary cost also for operations on the head of the data structure. It is called double ended because it effectively implements both the LIFO i.e. stack approach and the FIFO i.e. buffer approach.
- dictionary, i.e. hash table working on key-value pairs. It provide access to data at unitary cost, automatically managing any collisions.
- set, an array similar to a list but which ensures that there are no equal elements. Its implementation is actually similar to that of a dictionary, thus having the same pros and cons in terms of access time and memory used.

The following figure shows a comparison of performance between a list and a dictionary in the case of search operations [12]. The performances of the deque are practically comparable to those of a list.

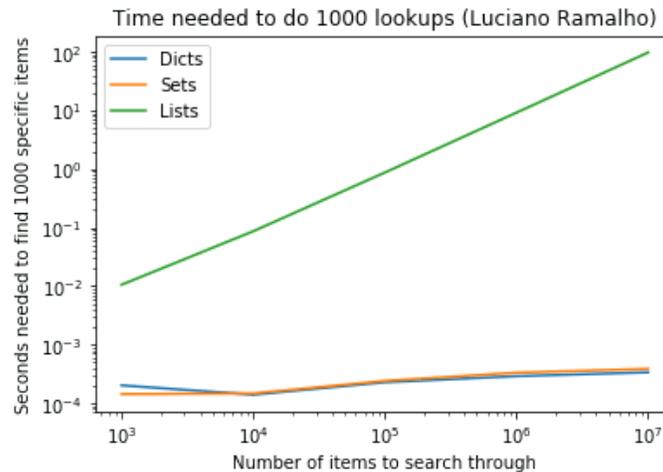


Figure 6.1: Comparison of access time to specific elements for different Python data structures.

It is important to note that in the case of linear scanning, i.e. processing all elements sequentially, the three data structures behave in an almost equivalent way.

In myP2PSync the list is usually used in cases where operations need to be performed on each element of the data structure, such as to retrieve the list of peers from the tracker. In this case the server responds with a list, and the peer simply scrolls it completely. The deque structure is used for the implementation of the scheduler. It is in fact nothing more than a FIFO queue i.e. a buffer, where the first element to be processed is the one inserted less recently. In other words, the one on the head of the structure. Operations on the head are not efficient on a simple list, while they have unitary cost on a deque. Furthermore, the scheduler does not search for a specific item, but only performs operations in the head or queue or scan the complete list. This is why a dictionary was not considered for the scheduler data structure implementation. In the rest of the application dictionaries are used more persistently. For example, both on the tracker side and on the peer side, groups and related information are stored in dictionaries whose key is the name of the group while the value contains all the group information.

A disadvantage of using dictionaries is that they require a lot of memory to be implemented, are therefore not suitable to contain large amounts of data and are especially not recommended in the case of nested dictionaries. A dictionary in Python is an hash table, an array-like data structure with an efficient element addressing mechanism. The index of an element in this data structure is obtained by applying a hash function to a key. In the CPython implementation i.e. the official and reference implementation of the project, the key, hash code and the element itself are stored in each field of the array. This tuple occupies a considerable amount of memory space. Moreover, the hash table is subject to the problem of collisions, that is, different keys that are mapped, through the application of a hash function, on the same code and therefore on the same memory cell. To make the problem less relevant sparse arrays are used, with more than half of the elements empty. Unfortunately, this leads to a considerable waste of memory, because the empty tuples still occupy the same space as a tuple (key,hashCode,value). The problem has been solved with a new and efficient implementation used by Python 3.6 and later versions [13]. The new solution uses two distinct arrays: the first is equivalent to the one used previously, so sparse, but as elements it only

contains indexes pointing to a second array, this time compact containing tuples with actual values. The first vector in this case wastes only the space used to store a real number, instead of the one used for an empty tuple. This justifies the pervasive use of dictionaries in myP2PSync, they are in fact more efficient than lists and dequeues in terms of performance and are only slightly more expensive in terms of space thanks to their new implementation.

6.1.1 Synchronization on access

It is also important to remember that both lists, dequeues and dictionaries are thread-safe data structures, i.e. the individual operations are atomic and race conditions cannot occur between the various threads. However, in some cases like the one in which before inserting an item in a dictionary it is verified that it is not already present, they are possible. For example, considering the case in which two peers want to almost simultaneously create a group of the same name 'abc':

- threadA looks for key 'abc' in a dictionary of groups and it discovers that is not present yet, so it assumes that there isn't a group with the same name;
- meanwhile threadB does the same and it adds a new group object for the same group name 'abc';
- at this point threadA, that is not aware of threadB insertion, add its value for 'abc' overwriting what threadB added.

This leads to a state of inconsistency because the peer served by threadB sees that the group was created but actually it is not part of it. The solution to the problem consists in using a lock that encloses both the verification of the key and the insertion in the data structure. For example:

```
1 groupsLock.acquire()
2 if newGroupName not in groups:
3     newGroup = Group(newGroupName, newGroupTokenRW,
4                       newGroupTokenRO)
```

```
4     newGroup.addPeer(peerID, True, "Master")
5     groups[newGroupName] = newGroup
6 groupsLock.release()
```

6.2 Server reachability

The current implementation lacks of an automatic mechanism to reach the tracker server. In fact, its IP address and port number must be written by the user in the configuration.json file. Alternatively, if the file is not present or badly formatted, or if the specified server is not reachable, the user is allowed to manually enter this information when starting the application. It was decided to use this approach instead of the one based on a public domain more easily accessible by all users because the system is designed to be as private as possible. That is, each user or company should install and use their own isolated instance of the server, being it a light application that requires a limited amount of resources to work effectively, especially in a domestic context. In the case of the public domain, all users would have used the same server and would also have had information about groups of other users or companies, even if they might not be able to access them due to the lack of tokens. With the current implementation, however, users only have information on a limited number of groups, namely those registered on the single server, which makes everything lighter and increases privacy. Of course, this makes everything a bit less plug and play, because many users may not know about concepts such as IP address and port number, and even knowing what they are you may have difficulty retrieving them from the machine on which you run the server. However, this was an implementation choice. The system is designed for intermediate users who have a minimum of experience in network configuration and command line use. In fact, the user must also be able to install the ZeroTier software and must also run the scripts from the command line by invoking the Python interpreter, which in the rare case that it is not present must be installed. All information on installing, configuring and running myP2PSync can be found in chapter 8.

6.3 Previous session information

The applications, both the server and the myP2PSync client used by the peers, need to save the data of a closing session so that it can be reused in the next session in the initialization phase. This information includes:

- on the peer side the information needed to contact the server, i.e. its coordinates in terms of IP address and port number, and a list of all the files managed by the various synchronization groups;
- on the server side all the information about the groups, such as access tokens, obviously encrypted, lists of peers registered in a certain group and lists of files currently synchronized within a group.

Two solutions have been considered for saving this information. The first was to use a central database, such as mySql, or a local database such as SQLite. The second, which is the one actually implemented in the current version, provides for the use of simple configuration files in JSON format. The choice was dictated by the size of the problem. Databases are usually used to store large amounts of data and are slower, especially those that require a remote connection. As myP2PSync is a project at an early stage, the amount of data is unlikely to justify the choice of databases. We have therefore opted for the second, also considering the excellent compatibility between the Python language and the JSON formatting language. In fact, JSON formats any data in the form of lists and dictionaries, possibly nested. These two data structures are identical to those provided by default by Python. That is, a JSON dictionary is mapped exactly to a Python dictionary.

For example, if we consider the file where the server saves the complete list of peer entries to groups, we have a structure like this:

```
1  [  
2    {  
3      "peerID": "52232004257",  
4      "groupName": "ITDepartment",  
5      "role": "RW"
```

```
6     },
7     {
8         "peerID": "132098950711908",
9         "groupName": "ITDepartment",
10        "role": "Master"
11    }
12 ]
```

This file is mapped directly by the load function of the json module to a list of dictionaries, where each dictionary identifies the entry of a peer to a group. At this point it is very easy to add a single peer to the list of a group. The resulting code is as follows:

```
1     f = open(groupsPeersFile, 'r')
2     try:
3         peersJson = json.load(f)
4         for peer in peersJson:
5             peerID = peer["peerID"]
6             groupName = peer["groupName"]
7             role = peer["role"]
8             groups[groupName].addPeer(peerID, role)
9     except ValueError:
10        pass        # wrong JSON file structure
11    f.close()
```

This approach is used to read all session files used. Moreover, using the dumps function, also provided by the json module, it is possible to carry out the inverse operation, that is to say to store a snapshot of the data structures in a JSON file.

```
1 peersJson = list()
2 with open(groupsPeersFile, 'w') as f:
3     for group in groups.values():
4         for peer in group.peersInGroup.values():
5             peerInfo = dict()
6             peerInfo["peerID"] = peer.peerID
7             peerInfo["groupName"] = group.name
8             peerInfo["role"] = peer.role
9             peersJson.append(peerInfo)
```

```
10     json.dump(peersJson, f, indent=4)
```

In the specific case of the above example, the only preliminary step required is to map the properties of Group objects in the fields of a peerInfo dictionary. This is only necessary because the data structure is not directly a dictionary.

Chapter 7

Main issues

During the development of myP2PSync there were several problems and difficulties, sometimes easy to solve and sometimes definitely less so. The fact that it is a distributed system implies most of them, in fact such problems are often common in all applications of this type. This chapter discusses how most of them have been resolved, including the various discarded alternatives. The chapter is divided into the following paragraphs:

- 7.1 NAT Traversal Problem
- 7.2 Synchronization problems
- 7.3 Path compatibility
- 7.4 Debugging

7.1 NAT Traversal Problem

The main problem in developing a Peer-to-Peer application is the so-called "NAT Traversal Problem". Today, most corporate and domestic networks use private IP addresses and interface with the outside world using a technique called NAT (Network Address Translation) applied to the router. This mechanism simply maps private IP addresses to a single shared public IP

address whenever a machine in the network tries to reach or establish a connection with another machine outside the network. In addition to translating of the IP address, the NAT also stores a mapping between the port numbers of the router interface and the machine within the private network. This mechanism allows the router to correctly redirect the response that is received after a request e.g. an ICMP ping request or an HTTP request. The IP address is not sufficient to distinguish the various machines. Unfortunately, this does not work well in the case of P2P applications because the machine within the private network must be able to be reached even without a first request leaving the network because peers must also work as servers and accept connections from outside. The problem is that in this case there is no preliminary message instructing the NAT on a certain mapping and therefore the internal machine is not reachable because the router rejects the request. In other words, hosts behind a NAT are only authorized to initiate outgoing traffic and connections while they are never authorized to receive incoming traffic and connections initiated by a foreign host.

There are several solutions to this problem, even considering that this problem may arise in different contexts (e.g. only one of the two peers is behind a NAT, or both are behind a NAT) These solutions are also different depending on the level of transport used. Some solutions have proved to be more efficient, others more stable. Unfortunately, there is no solution that can be applied in any case, also because the success of the NAT traversal also depends on the type of NAT and the router model and its configuration.

A first solution could be that of relaying, i.e. using a server or in any case a machine with a public IP as an intermediary between the two ends of the communication (e.g. the sender peer and the recipient peer, using the tracker server as an intermediary). This solution is highly inefficient because it requires that data is always forwarded from this central machine, wasting bandwidth and computational capacity, and introducing some delay. This solution is the one used by Skype to solve the problem.

Another approach is the so-called "NAT hole punching", perhaps the most efficient and used solution. It always requires the presence of a central server that knows both the private and public IP addresses of the two machines that want to communicate. When two machines want to communicate, they ask the server for the addresses of the other machine and try to communicate

simultaneously. This can mean that at a certain point one of the NATs has a port mapping configuration that makes a connection request "enter". Unfortunately, this technique has a success rate of 60% when it comes to TCP connections and 80% for UDP, because it is very dependent on the behaviour of the router [14]. It also requires that both machines want to communicate as in the case of BitTorrent where the communication takes place on a stable and usually bidirectional connection, in fact once created it is maintained until one of the two peers does not leave the torrent. In the case of myP2PSync this situation does not occur because the connection between the peers is established only when necessary, in fact it would be useless to keep an active connection if there are no files to be synchronized. The technique would be implemented with a notification system for the peer to be contacted. E.g. peerB wants to communicate with peerA, so peerB alerts the server that it passes the news to peerA, for example through a mechanism of polling peerA to the server. At this point both peers try to connect. All this can be really inefficient and would introduce significant delays in synchronization, especially in the case of small files.

A third solution is that of a simple port forwarding. There are several tools that implement this technique, including `miniupnpc`. This tool also has a binding for the Python language, which unfortunately is not yet compatible with Windows. It was therefore decided to use the command line application directly by integrating calls to instructions in Python (using the `os` module). This solution unfortunately only works with routers that enable `upnp`, which is a rather rare case. In fact it usually has to be activated manually by accessing the router, which is possible in home routers but hardly works in environments like the corporate one. This approach has another disadvantage: if two peers run the server on the same port (which is arbitrarily chosen from the free ports) and both use port forwarding, it will only be effective for one of the two machines. The other one will then not be reachable from outside but only from other peers within its own network. The latter case is very rare but it is still possible.

Finally, the use of a last solution, that of VPNs, was considered. There are several options that focus on the problem of NAT traversal, including PeerVPN and ZeroTier. The former is unfortunately only compatible with Linux while the latter is cross-platform but requires administrator privileges. It was therefore decided to implement a resolving approach based on Ze-

roTier, which is completely open source. Since there is no binding available for Python, the command line utility has been integrated into the application using the `os` module. Citing the definition given in the official ZeroTier documentation [15]:

“ZeroTier is a distributed network hypervisor built atop a cryptographically secure global peer to peer network. It provides advanced network virtualization and management capabilities on par with an enterprise SDN switch, but across both local and wide area networks and connecting almost any kind of app or device.”

In short, ZeroTier allows all nodes in the same network to communicate as if they belonged to the same LAN. It also encrypts and decrypts messages that are exchanged. The operation is based on the use of the methods previously described, in particular the NAT hole punching and relaying. With the free option offered by ZeroTier it is possible to create and manage networks up to a maximum of 100 nodes. There are also paid options that allow you to create networks with an indefinite number of nodes. All you had to do was create and configure a personal network for the application with its own ID. Within the application, the peer connects to this network using the provided ID and disconnects from the network shortly before closing. The rest of the logic of the program remains unchanged with the only difference that the IP addresses used to generate the connections are no longer the real ones but those assigned by ZeroTier to the various peers. In order to do that the tracker server registers a peer using its ZeroTier IP address and not the real one. The ZeroTier addressing approach has been extended also to the server, although it does not need it because it can be reached at a public address, in order to exploit the cryptographic mechanisms provided by ZeroTier also in the exchange of messages with peers. Since the ZeroTier IP address of the server is unknown to the peer when the application is started, it sends a request to its public IP address in order to retrieve it. From then on, all messages are sent from the peer to the ZeroTier IP address to ensure the security of the connection. With regard to the security of communications above ZeroTier I report the following statement extracted from the documentation:

“Packets are end-to-end encrypted and can’t be read by roots or anyone else, and we use modern 256-bit crypto in ways recommended by the profes-

sional cryptographers that created it. Asymmetric public key encryption is Curve25519/Ed25519, a 256-bit elliptic curve variant. Every VL1 packet is encrypted end to end using (as of the current version) 256-bit Salsa20 and authenticated using the Poly1305 message authentication (MAC) algorithm. MAC is computed after encryption (encrypt-then-MAC) and the cipher/MAC composition used is identical to the NaCl reference implementation.”

More specifically, the functioning of ZeroTier is as follows:

“Nodes start with no direct links to one another, only upstream to roots (central server/s provided by ZeroTier). Every peer possesses a globally unique 40-bit (10 hex digit) ZeroTier address, but unlike IP addresses these are opaque cryptographic identifiers that encode no routing information. To communicate peers first send packets "up" the tree, and as these packets traverse the network they trigger the opportunistic creation of direct links along the way. The tree is constantly trying to "collapse itself" to optimize itself to the pattern of traffic it is carrying.”

Peer to peer connection setup goes like this:

1. A and B are two peers and both are registered in the ZeroTier network
2. A wants to send a packet to B, but since it has no direct path it sends it upstream to R, a ZeroTier server.
3. If R has a direct link to B, it forwards the packet there. Otherwise it sends the packet upstream until a ZeroTier server with knowledge about B is reached and the latter forwards the packet to B.
4. R also sends a message called rendezvous to A containing hints about how it might reach B. Meanwhile the root that forwards the packet to B sends rendezvous informing B how it might reach A.
5. A and B get their rendezvous messages and attempt to send test messages to each other, possibly accomplishing hole punching of any NATs or stateful firewalls that happen to be in the way. If this works a direct link is established and packets no longer need to take the scenic route.
6. Since roots forward packets, A and B can reach each other instantly. A

and B then begin attempting to make a direct peer to peer connection. If this succeeds it results in a faster and lower latency link.

Furthermore:

"If a direct path can't be established, communication can continue through (slower) relaying. Direct connection attempts continue forever on a periodic basis. ZeroTier also has other features for establishing direct connectivity including LAN peer discovery, port prediction for traversal of symmetric IPv4 NATs, and explicit port mapping using uPNP and/or NAT-PMP if these are available on the local physical LAN."

The following figure shows a simple case of establishing a connection using ZeroTier. Both machines are behind a NAT and both are registered to the same root ZeroTier server. If the direct connection between the machines is not possible, the server takes care of the relaying operations.

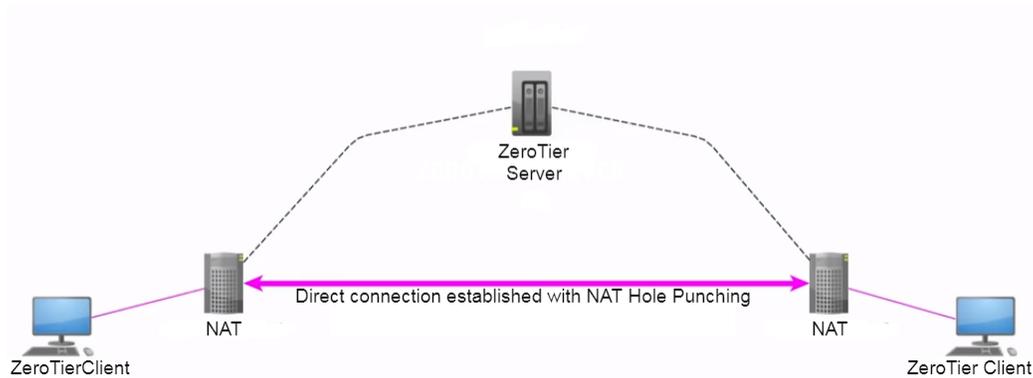


Figure 7.1: Example of NAT traversal connection established using ZeroTier.

In conclusion, all solutions, except port forwarding, have as their main disadvantage a slight increase in the time needed to establish connections and communicate. Unfortunately, there are no alternatives today. The final choice was ZeroTier because it is the most reliable and easy to implement solution. Its main disadvantage is that it requires administrator privileges and therefore forces users to launch applications as follows:

- in Windows you have to run the python interpreter from a command

prompt launched in admin mode;

- in Linux/macOS the interpreter must be run as sudoers using `sudo python3`.

If ZeroTier does not work, i.e. if the peer is still not reachable, the firewall configuration must be changed to allow the ZeroTier application to cross it over private networks.

7.2 Synchronization problems

Another crucial and problematic aspect of distributed application development is synchronization. The fact that different users have access to common resources, in the case of myP2PSync we talk about files or groups, should be managed as precisely as possible to avoid situations of inconsistency and error. Unfortunately it is not always possible to manage these situations in a precise and rigorous way, because the synchronization of files is not an instantaneous operation but it takes some time to be completed. It has already been pointed out in previous chapters that a system like myP2PSync works better in contexts where the sources of synchronization, i.e. users who actually modify files and submit new versions to the system, are limited or unique. If only one user forces the synchronization of the files, their distribution will be precise, coherent and sequential. If instead we consider more peers that modify the same file, the ideal case of myP2PSync functioning is the one in which each peer modifies the file only after the last synchronization has ended and in the meantime no other peer acts on the same version.

E.g. Ideal case synchronizations flow:

1. peerA adds file.txt;
2. peerB completes the download of file.txt;
3. peerB modifies file.txt and sends an update message to all the other peers;

4. peerA receives the message and downloads file.txt's new version;
5. peerA modifies file.txt and sends an update message to all the other peers and so on..

If instead we consider the case in which in the same group two or more peers modify and submit more or less simultaneously different versions of the same file can occur critical situations and that lead to loss of useful data. There is no mechanism in myP2PSync that allows you to merge the changes, if not overlapped, and distribute only a version of the file that includes them all exhaustively. This approach is similar to the one used by file versioning software like Git, based on pull, push and merge operations and could be considered and added in the future to make it more stable. In the current implementation of myP2PSync the various versions of the files are synchronized sequentially, i.e. the first peer to submit a change is the first to be served, distributing its version. However, if in the meantime another peer sends a synchronization request, and it is related to a more recent version of the file (with reference to the timestamp of the file), the first request is blocked and the new served, making sure that all the changes of the first peer are lost. It is also possible that the second peer submitting the change is ignored by the other peers because its version of the file is older. In other words, there may be different situations depending on the timing of the synchronization requests and timestamps of the various local files, but the weak point of such a mechanism is that some changes may be lost because they are not applied to the latest version of the file.

E.g. Bad case with lost of information

1. peerA modifies file.txt and finishes at t_1 ;
2. peerB modifies file.txt and finishes at $t_2 > t_1$;
3. peerA and peerB send an update message to all the other active peers at time t_3 and t_4 respectively;
4. at this point two situations are possible considering a peerC point of view:

- $t3 < t4$: peerC receives the update message from peerA and he adds the synchronization task to the scheduler queue, possibly triggering immediately the execution of the download process. Then he receives the peerB update message, he blocks the synchronization of the peerA's version and starts the one of the peerB's version (because $t2 > t1$);
- $t4 < t3$: peerC receives the update message from peerB and he adds the synchronization task to the scheduler queue, possibly triggering immediately the execution of the download process. Then he receives the peerA update message, but $t1 < t2$ so he simply ignores the message.

In both cases the changes made by peerA on the file are lost.

Another possible case that can lead to an overwrite of some modifications made by a peer on a file is the following:

1. peerA is not currently using myP2PSync, but he is registered in synchronization group. Alternatively, he is using myP2PSync but he is not active in that specific group. Instead peerB is active in the same group;
2. peerA modifies one of the file at time $t1$;
3. peerB modifies the same file at time $t2 > t1$;
4. at this point two things can happen:
 - if peerA goes online before $t2$ the file is correctly synchronized;
 - if peerA goes online after $t2$ its version will never be synchronized, instead it will be overwritten by the new version provided by peerB.

A possible solution to this problem is to have only one peer in the group that can submit new versions of a file, but this greatly limits the experience of using the application. Another possible solution is to aim to make synchronization much faster by sending only the modified parts of a file instead

of the entire file and automate synchronization, this would make everything much more real-time as for example happens in products like Google Docs. A last approach that can be considered is to keep track of all the overwritten modified version of a file. Of course, this can be dangerous and slow in case of big file, while it can be feasible for small files.

7.3 Path compatibility

Since the application is aimed at multiple desktop platforms, specifically multiple operating systems, the problem of paths had to be solved. Windows uses the `'\'` i.e. backslash as the path delimiter, while UNIX based systems like macOS and Linux use the `'/'` i.e. slash. However, using the Python3 `os` module the management of delimiters is left to the module itself. In creating paths, for example in the case of a file download where a temporary file is created, a UNIX-like approach was used, which thanks to the `os` module does not create problems even in Windows. So the only thing that happened in the programming phase was to force the UNIX-like style, going to convert the backslashes into slash. For example:

```
1 scriptPath, scriptName = os.path.split((os.path.abspath(
    __file__)).replace("\\", "/"))
```

This line of code retrieves the location of the script in the user file system, so it can be used as location to create the folder containing the synchronized files. Backslash conversion is just a stylistic detail that ensures that the paths files are consistent in terms of the delimiter used. In fact, if you do not replace the path files, inside the file Manager in a Windows environment, they would appear as follows:

```
C:\myP2PSync\client\filesSync\groupname\filename
```

As you can see in this case the first part of the path, the one that leads to the folder containing the application is Windows style, while the second part uses the UNIX like approach because forced by the application.

7.4 Debugging

The last aspect to be discussed is the difficulties encountered in debugging the system. Being the application based on a distributed system, we had to test its use using multiple machines at the same time and evaluating all possible critical situations that could occur, in order to try to avoid them and make the system as stable and reliable as possible. This proved to be very time-consuming, as there are so many possible situations. For the distribution of the software on the various machines, Git was used, exploiting the remote repository as a point where to submit the latest version of the software and especially as a point from which to recover it. In other words, the software changes were made on a machine and then transmitted to the remote repository with a push operation. Finally, on the other machines that you used as testers we simply cloned all the software in order to use the latest version of the application. We thank the Department of Computer Architecture of the Universitat Politècnica de Catalunya - BarcelonaTECH for the kind permission of two virtual machines and two physical machines for all the time necessary to develop and test the project.

Chapter 8

How to use myP2PSync

This chapter provides the information necessary for a user to use myP2PSync, both considering the installation and startup of the tracker and the installation, configuration and launch of the client application. It also details some aspects and constraints of using myP2PSync. The chapter is divided into the following paragraphs:

- 8.1 How to run a myP2PSync Tracker application
- 8.2 How to run a myP2PSync Client application
- 8.3 Usage constraints

In the following paragraphs we assume that the user has a local version of the code, which can be easily obtained from my personal GitHub pages:

```
https://github.com/flcasciaro/myP2PSync
```

From the command line it's possible to retrieve the code by executing the instruction:

```
1 git clone https://github.com/flcasciaro/myP2PSync
```

To run the tracker you need to have the directories `trackerApplication` and `shared`, while to run the client you need to have the directories `peerApplication` and `shared`.

8.1 How to run a myP2PSync Tracker application

To run the tracker application, you must first have a machine that is as stable as possible, both in terms of reliability and network connection. A possible disconnection or unavailability of the tracker device leads to the crash of the entire application and in the current implementation there are no recovery mechanisms.

Before running the tracker server, make sure that the Python3 interpreter is installed in the device. If it is not, it can be installed using:

- the executable that can be retrieved directly from the site in the case of Windows systems;
- directly from the command line or a package manager in a UNIX-like environment e.g. `apt install python3`.

Once you have `python3` installed, make sure you have `ZeroTier` installed, the software used to solve the NAT traversal problem. The `ZeroTier` site provides all the necessary information. In the case of Windows you will have to use an installer while in a UNIX environment you will only need the command line. For further details, please refer to the `ZeroTier` website where the various cases and instructions that can be used in the terminal of the various Operating Systems are listed.

At this point to run the tracker you simply need to run the `myP2PSyncTracker.py` script using the Python3 interpreter. For example:

```
python3 myP2PSync/trackerApplication/myP2PSyncTracker.py
```

To ensure that ZeroTier works properly, you must have super user privileges.

- Under Windows, you must open the command prompt using Administrator mode;
- In a Unix environment, you just have to put 'sudo' before the instruction.

The tracker application execution will remain active until the user decides to stop it using the key combination CTRL+C.

The following figure shows the output following the activation of the tracker, also highlighting the real IP address of the server and the port number on which it is executed.

```
Previous session status restored
200 join OK
Obtained 10.243.200.199 address from ZeroTier
Starting socket server (host 147.83.131.5, port 45154)
█
```

Figure 8.1: Output of the myP2PSync tracker execution starts.

These IP values and port numbers must be entered in the configuration file of all clients that want to use this server.

8.2 How to run a myP2PSync Client application

To launch the client, the process is practically identical to that of the tracker: you need to install the python3 interpreter and the ZeroTier software using the same methods already described in the previous paragraph. To make the client work, however, you must install two additional Python modules: pyqt5 and qdarkgraystyle. The order in which they are mentioned is also the

order in which they need to be installed. To install both you can use the pip3 utility which is usually installed automatically with python3. For example, in order to install pyqt5 the command is:

```
pip3 install pyqt5
```

Now you have to configure the client so that it can reach the designated local tracker server. You must then enter the coordinates read in the output of the tracker application, as highlighted in the previous paragraph, in the JSON file configuration.json stored inside the folder myP2PSync/peerApplication/sessionFiles.

The following figure shows a correct setting of the file.



```
{
  "serverIP" : "147.83.131.5",
  "serverPort" : 45154
}
```

Figure 8.2: Example of JSON configuration file setting.

You can now run the myP2PSync client by using:

```
python3 myP2PSync/peerApplication/myP2PSyncClient.py
```

If the tracker is not reachable at startup, due to an error in formatting the configuration file, you can enter the coordinates in a pop-up window. Use the format <IPaddress>:<PortNumber> e.g. 147.83.131.5:45154.

8.3 Usage constraints

The myP2PSync system manages the filepath of the files in synchronization in two possible ways depending on the role of the peer in the addition of the file:

- if the peer is the source of the file, i.e. the one who added it, it remains saved in the location from which it was added, and each version will override that location;
- if the peer is not the source of the file, it is saved to the default path:

`myP2PSync/peerApplication/filesSync/groupname/filename`

filename also takes into account any levels of directories.

The system is unable to handle files with names containing spaces, which are rejected and not added to a sync group. The same is valid for directories names. It is also required that group names and tokens do not contain spaces.

Chapter 9

Testing

This chapter describes the final phase of the project, which is aimed at testing performance, also with a view to evaluating and optimizing some parameters of the file-sharing algorithm. One of the following paragraphs also describes some previous versions of the file-sharing protocol, highlighting the limitations that led to the development of the current version described in chapter 5. The system has been compared with a very similar system developed specifically for testing that works in Client-Server mode, so as to have a comparison between the two approaches. The performances were also compared with those of a similar commercial application. The chapter is divided into the following paragraphs:

- 9.1 Testing environment and tools
- 9.2 File-sharing protocol parameters optimization
- 9.3 Previous versions of the file-sharing protocol and their limitations
- 9.4 P2P vs CS performance
- 9.5 myP2PSync vs similar product
- 9.6 Testing result overall

9.1 Testing environment and tools

The system was tested using four devices, one of which was used to run the tracker server and three others were used for the clients. The detailed list of devices is as follows:

- the first device, used to run the tracker, is a Linux machine with Ubuntu 16 distribution. It is associated with a public IP address.
- the second and third devices used to run the client are Linux machines with the Ubuntu 16 distribution. They also have a public IP address.
- the fourth device, which always performs client functionality, is a machine with a Windows 10 operating system. It has been used within a private network, which is useful for testing the NAT traversal capabilities of the application.

All four devices belong to different networks. Access to the three Linux machines was controlled remotely via SSH connection and using an X11 server to utilize the myP2PSync graphical interface from the local machine.

The integrity check of the transmitted files was carried out using the `sha1sum` command which calculates the hash sha1 function of a certain file. The command is applied to transmitted and received files and the two values obtained are compared. If they are identical, the file has been transmitted without errors.

The following figure is a representation of the testing system.

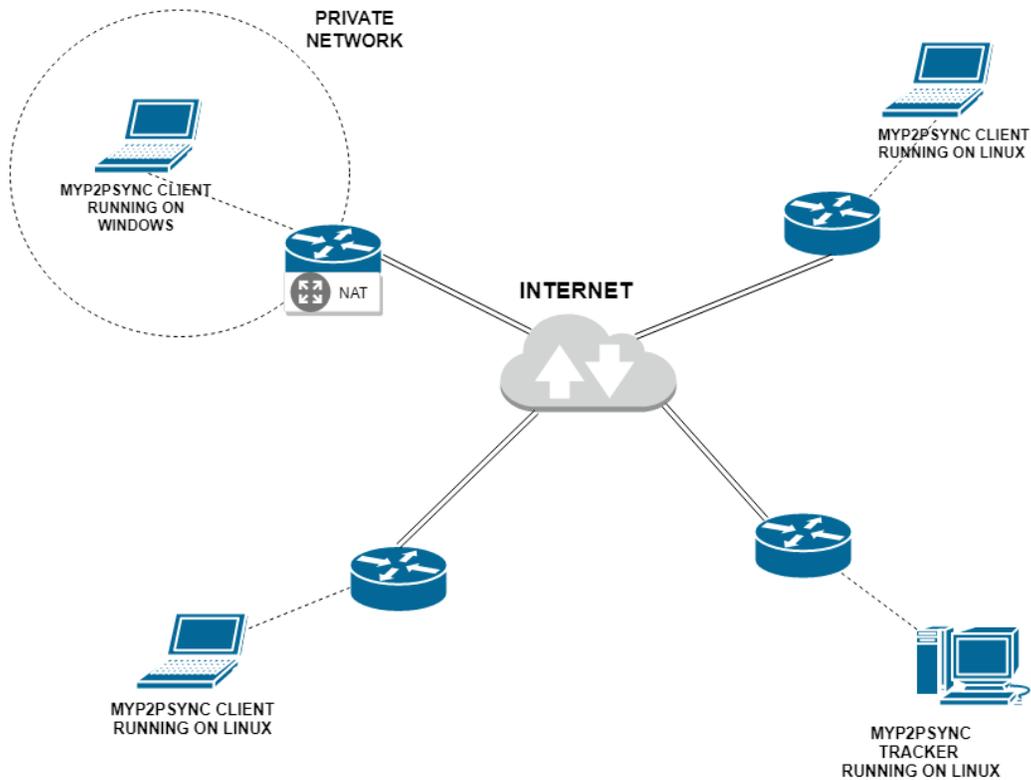


Figure 9.1: Testing environment representation.

The study carried out in this chapter focuses on the situation where a peer adds a certain file to a group and all other peers activate synchronization. In other words, one node of the network acts as a seed for the file, while all the other nodes act as normal peers. The study of the parameters carried out in paragraph 9.2 takes into consideration only this case being the most common one for a file synchronization system. However, this study could be extended in the future to other cases, such as the one in which several peers already own the file and only one other carries out the synchronization process, working with a view to finding the best configuration of parameters that work optimally in all conditions.

9.2 File-sharing protocol parameters optimization

Within the file-sharing protocol described in chapter 5, some parameters that influence the behavior and efficiency of this protocol have been highlighted. During the testing phase we tried to determine the most appropriate values for these parameters, going to perform tests in the transmission of files, keeping track of the transmission time used using certain values of the parameters. In order to be as independent as possible from the randomness of the network condition, the tests were always repeated three times, then calculating and considering the average value of the three tests. The system has been tested considering a seed machine that adds the file to the group and therefore has all the chunks and two machines that perform the download. The value indicated represents the average value of the transmission time on the two machines. The average transmission time relative to files of different sizes: 100MB, 500MB, 1GB was analysed in order to highlight how the size of the file influences the efficiency of the protocol.

During the testing phase, the values of `MAX_THREADS` and `MAX_PEERS` were not optimized, since they are relevant in the presence of a high number of peers active in the same group, while in the testing phase a maximum of 4 peers active at the same time was considered. Furthermore, the study of the parameters `INITIAL_TRESHOLD` and `TRESHOLD_INC_STEP` has not been done in detail and is therefore not reported below. However, it has been verified that the values used in the current implementation are sufficiently efficient.

9.2.1 `MAX_CHUNKS` evaluation

Initially a certain value was set for the parameters related to the Random Discard approach, in order to test the parameter `MAX_CHUNKS`. The test on `MAX_CHUNKS` was carried out using:

- `COMPLETION_RATE=95%`;

- INITIAL_TRESHOLD=50%;
- TRESHOLD_INC_STEP=0%;
- MAX_PEERS=10;
- MAX_THREADS=5.

The tests returned the following results:

MAX_CHUNKS	100MB	500MB	1GB
10	47	224	482
30	43	180	385
40	40	177	384
50	39	165	377
60	40	164	360
70	40	163	358
80	39	162	357
100	44	162	348
125	45	175	360
150	47	180	371

Table 9.1: Transmission times as a function of MAX_CHUNKS.

The rows of the table show the trend of the transmission times, in seconds, for files of different sizes as the MAX_CHUNKS parameter changes. As you can see the parameter MAX_CHUNKS is also related to the size of the file, in fact for small files a parameter value less than or equal to 80 is the best. For larger files, better performance are obtained with a value close to 100. A too high MAX_CHUNKS value limits the exchange of chunks between non seed peers and tends to monopolize the connections, i.e. you try to recover too much from a single, or at least a few peers. This is good for small files, but for large files it is better to have an intermediate value. If MAX_CHUNKS is too small, too much time is wasted on chunks allocation from the different threads. It was therefore decided to set the value of the parameter MAX_CHUNKS to 100, as it was particularly effective for medium and large files.

9.2.2 COMPLETION_RATE evaluation

At this point, having set the value of MAX_CHUNKS equals to 100, we looked for an optimal COMPLETION_RATE value, which corresponds to the percentage of files obtained during the download beyond which the random discard approach is no longer used. To perform the test, the other parameters have been set to the following values:

- INITIAL_TRESHOLD = 50
- TRESHOLD_INC_STEP = 0;
- MAX_THREADS = 5;
- MAX_PEERS = 10.

The results obtained when the COMPLETION_RATE parameter is varied are shown in the following table.

COMPLETION_RATE	100MB	500MB	1GB
80%	46	164	372
90%	45	161	362
95%	44	160	343
99%	47	165	359

Table 9.2: Transmission times as a function of COMPLETION_RATE.

A high Completion Rate value means that the random approach is used for much of the download, facilitating the spread of chunks outside the seed user. An extremely high value (99%) is however slightly less performing because it probably forces the peer to a last iteration very abundant in terms of chunks, slowing down the closure of the download. The optimal value therefore seems to be 95%.

9.3 Previous versions of the file-sharing protocol and their limitations

Before the current implementation of the file-sharing algorithm, another version was implemented. It was based on the use of threads in a more schematic and synchronous way. The main thread of the download was responsible for generating the `rarestFirstChunksList` and properly creating lists of chunks to be requested from different peers. Once all lists were generated, the main thread instantiated all `getChunks` threads and waited for them to be terminated. When all the threads had finished their task, it would start again from the beginning until the download was complete. This approach of generating `getChunks` threads all at once and waiting for the termination of each of them before they could continue ended up slowing down threads that communicated with fast peers. In fact, the threads that were communicating with slower peers blocked all the others.

An improved version of this algorithm that tries to solve the problem of blocking fast threads was then implemented using the concept of Round Trip Time. The main thread, when asking the various peers for their `chunksList`, also took note of how long it took them to receive the message and send their response. In the creation of the lists of chunks to request he used the RTT as a discriminating factor. A thread that worked with a fast peer was therefore led to request more chunks than a slower one. All this with the aim of making more or less the execution time of the various `getChunks` threads similar. Unfortunately the networks are very dynamic entities and their congestion changes over time very quickly. Even the second approach did not give satisfactory results, and this led to the birth of the current protocol.

To realize the difference in performance between the old protocol optimized with the RTT and the current one, some tests on the transmission performance have been carried out and are contained in the following comparative table.

	OLD ALGORITHM	NEW ALGORITHM
100MB	85	44
500MB	268	160
1GB	686	343

Table 9.3: Transmission times comparison of old and current file-sharing algorithms.

The data relative to the new algorithm are obtained considering the optimal parameters `MAX_CHUNKS` and `COMPLETION_RATE` obtained in the previous paragraph and using the same values for all the other parameters. The previous version of the protocol used different parameters that will not be discussed in more detail in this description. As you can easily see, the difference in performance is evident, since the new algorithm halves the average transmission times.

9.4 P2P vs CS performance

During the testing phase we wanted to verify that the P2P approach was really more efficient than the classic Client-Server approach. A second distributed application was developed using the CS scheme for sending files, using transmission protocols similar to those of `myP2PSync`, of course only for what concerns the transmission on sockets and not the actual file-sharing protocol. This application can be found at the following url:

<https://github.com/flcasciaro/fileSharingCS>

To compare the two systems we have operated as usual in the case of the `myP2PSync` system, i.e. a seed user loads a file and a certain number of peers users receive it. In the case of the Client-Server system instead the seed corresponds to the server application while the peers are simple clients that launch requests for files almost simultaneously. The following table summarizes the data related to the comparison between `myP2PSync` and

the fileSharing system based on the CS approach, when you consider only one seed machine and two that get the file at the same time.

	myP2PSync	File-sharing CS
100MB	44	29
500MB	161	144
1GB	343	309
4GB	1250	1180

Table 9.4: Transmission times comparison between myP2PSync and a CS file-sharing application when two users retrieve a file at the same time.

If we consider a seed machine and three machines that get the file:

	myP2PSync	File-sharing CS
100MB	48	45
500MB	159	255
1GB	303	513
4GB	1064	1990

Table 9.5: Transmission times comparison between myP2PSync and a CS file-sharing application when three users retrieve a file at the same time.

As you can see in the CS approach, times increase almost linearly to the size of the file and the number of clients that require a certain file at the same time. However, for small files, the CS approach and a low number of users, the CS approach is more effective. If we consider the case of 3 machines, the P2P approach is already slightly more effective than the Client Server approach. The seed machine at best transmits only one copy of the original file, while all the other peers synchronize by exchanging chunks between them. Of course this is the ideal case, but it is unlikely to happen. We can see how the P2P approach is also more effective in the case of large files.

Finally, the behaviour of the two systems was considered when all three files

are requested at the same time and therefore the level of congestion is high. The results are as follows:

	myP2PSync	File-sharing CS
2 machines	151 - 386 - 598	192 - 415 - 660
3 machines	107 - 325 - 467	265 - 579 - 812

Table 9.6: P2P vs CS: transmission times comparison in case of high congestion of the network.

The table considers both the case in which the machines that receive the file are two and the one in which they are three, with a higher level of network congestion. For each case, three values are indicated, corresponding to the medium duration of the transmission for the small file (100MB), the medium file (500MB) and the large file (1GB). As can be easily observed under these conditions, the Peer-to-Peer approach is much more efficient than the Client-Server approach.

9.5 myP2PSync vs a similar product

The performance of myP2PSync was compared with that of a similar commercial system, Resilio Sync. The performance of Resilio Sync. was better, although in the case of small files (100MB) they are very similar. In the case of medium and large files, 500MB and 1GB respectively, the performance of Resilio Sync. was much better. In detail, using 3 machines, one of which works from seed:

	myP2PSync	Resilio Sync.
100MB	44	42
500MB	160	143
1GB	343	280

Table 9.7: Comparison of myP2PSync with a similar commercial product in terms of file transmission time.

The result is negative for myP2PSync but not daunting. With some future optimization of the file-sharing algorithm the results could be improved.

9.6 Testing phase results

The results of the testing phase are satisfactory. The file-sharing algorithm has been subject to numerous changes during development, with the aim of making it as efficient as possible. It has been shown that the Peer-to-Peer approach is more efficient than the Client-Server approach even with a limited number of peers. Moreover, the comparison with a similar but much more consolidated system has given almost positive results. In addition to the tests described above, it was also verified that the system is stable and meets the functional requirements and not described in section 3.3. The system meets expectations, being able to restart synchronizations interrupted by a partial state, to work in stress conditions where the number of synchronizations required is high and is able to react to dynamic changes in the network, going for example to detect inactive connections or no longer working.

Chapter 10

Conclusion and future improvements

The myP2PSync system is currently functioning and meets all the requirements outlined during the project planning phase. The system is intuitive and easy to use, except for an initial configuration phase that must be performed by users with a minimum of experience and skills in the field of computer science. The development time was slightly longer than expected during the initial project management phase. In fact, there were several problems to be solved, such as those described in chapter 7. In addition, the testing phase has often highlighted limits and errors, the correction of which has led to new phases of coding and new testing phases. The current state of the project is satisfactory as evidenced by chapter 9 but the system can certainly be improved, both in terms of efficiency and performance and in terms of stability and reliability. Below is a list of possible future improvements that can be considered and developed:

- availability of myP2PSync on mobile platforms (Android and iOS);
- better integration of ZeroTier in myP2PSync, maybe integrating ZeroTier in a standalone module;
- transmission of file modified parts only in order to guarantee faster synchronizations and avoid incoherence problems, in a way similar to rsync [16];

- the possibility for the user to add a file or a directory directly into an already synchronized directory;
- auto sync option: the system should be able to automatically recognize modified files in order to forward the update messages to all the other peers in the group;
- addition of new functionalities and options for a master user. For example: remove a peer from the group, delete a group, block new updates for a file, change tokens;
- higher efficiency of the merge operation using an incremental approach: instead of merging all the chunks at the end of the download, every time a chunks is received write it directly in the temporary file;
- recovery mode for server: restore status of the server collecting information from peers;
- adding of a compression mechanism before message send operation, in order to save bandwidth and decrease transmission time;
- removal of some usage constraints: users should be able to add files, directories and groups with name containing spaces.

List of Figures

2.1	Comparison of Client Server and Peer-to-Peer model architectures.	19
2.2	Example of DHT chord structure with use of shortcuts.	23
3.1	General architecture of the myP2PSync system.	32
3.2	Messages exchanged during the join group operation.	33
3.3	Messages exchanged in order to retrieve the list of peers of a group.	33
3.4	Messages exchanged in order to retrieve the chunks list from another peer for a certain file.	33
3.5	Messages exchanged in order to retrieve file chunks.	34
3.6	Gantt chart representing the timeline of the project.	38
5.1	Overview of the myP2PSync system architecture.	43
5.2	Organization of the client application modules.	50
5.3	Example of client application window on startup.	51
5.4	Example of client application window for a group master user.	52

5.5	Architecture of the myP2PSync file system.	60
5.6	Socket connection establishment.	75
5.7	Communication on a socket connection.	78
5.8	myP2PSync client application multi-threading architecture. . .	82
5.9	Organization of a synchronization multi-threading.	83
6.1	Comparison of access time to specific elements for different Python data structures.	86
7.1	Example of NAT traversal connection established using Ze- roTier.	98
8.1	Output of the myP2PSync tracker execution starts.	106
8.2	Example of JSON configuration file setting.	107
9.1	Testing environment representation.	111

List of Tables

9.1	Transmission times as a function of MAX_CHUNKS.	113
9.2	Transmission times as a function of COMPLETION_RATE.	114
9.3	Transmission times comparison of old and current file-sharing algorithms.	116
9.4	Transmission times comparison between myP2PSync and a CS file-sharing application when two users retrieve a file at the same time.	117
9.5	Transmission times comparison between myP2PSync and a CS file-sharing application when three users retrieve a file at the same time.	117
9.6	P2P vs CS: transmission times comparison in case of high congestion of the network.	118
9.7	Comparison of myP2PSync with a similar commercial product in terms of file transmission time.	118

Appendix

The following list identifies all the requests that myP2PSync peers can send. Each request is initially generated by a peer and never by a tracker, who only provides answers. The first parameter of each request is always the peerID, so in the following lists it has been omitted.

List of requests that peers can send to a tracker:

- INFO
- HERE <zeroTierIP> <PortNumber>
- GROUPS
- RESTORE <groupName>
- JOIN <groupName> <token>
- CREATE <groupName> <tokenRW> <tokenRO>
- ROLE <action> <destinatonPeerID> <groupName>
- PEERS <groupName> <active/all>
- ADDED_FILES <groupName> <filelist>
- REMOVED_FILES <groupName> <filelist>
- UPDATED_FILES <groupName> <filesInfo>
- GET_FILES <groupName>
- LEAVE <groupName>
- DISCONNECT <groupName>
- EXIT

List of requests that peers can send to other peers:

- ADDED_FILES <groupName> <filelist>
- REMOVED_FILES <groupName> <filelist>
- UPDATED_FILES <groupName> <filesInfo>

Bibliography

- [1] Wikipedia contributors, “File synchronization — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/w/index.php?title=File_synchronization. [Online].
- [2] K. B. Robert Capra, Emily Vardell, “File synchronization and sharing: User practices and challenges,” *School of Information and Library Science - University of North Carolina at Chapel Hill*, 2015.
- [3] Wikipedia contributors, “Client-server — Wikipedia, the free encyclopedia.” <https://simple.wikipedia.org/w/index.php?title=Client-server>. [Online].
- [4] Wikipedia contributors, “Peer-to-peer — Wikipedia, the free encyclopedia.” <https://en.wikipedia.org/w/index.php?title=Peer-to-peer>. [Online].
- [5] D. S. Stephanos Androutsellis-Theotokis, “A survey of peer-to-peer content distribution technologies,” *ACM Computing Surveys (CSUR)*, 2004.
- [6] K. D. K. M. F. B. H. Stoica I., Morris R., “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM Computer Communication Review*, 2001.
- [7] J. I. K. Marling Engle, “Vulnerabilities of p2p systems and a critical look at their solutions,” *Networking and Media Communications Research Laboratories Computer Science Dept., Kent State University*, 2006.
- [8] Cohen, Bram, “The bittorrent protocol specification.” https://bittorrent.org/beps/bep_0003.html. [Online].

- [9] P. M. Arnaud Legout, Guillaume Urvoy-Keller, “Rarest first and choke algorithms are enough,” *ACM SIGCOMM/USENIX IMC’2006*, 2006.
- [10] S. S. B. Jahn Arne Johnsen, Lars Erik Karlsen, “Peer-to-peer networking with bittorrent,” *Department of Telematics, NTNU*, 2005.
- [11] Python wiki contributors, “Time complexity.” <https://wiki.python.org/moin/TimeComplexity>. [Online].
- [12] Jessica Yung, “Python lists vs dictionaries: The space-time tradeoff.” <https://www.jessicayung.com/python-lists-vs-dictionaries-the-space-time-tradeoff/>. [Online].
- [13] Jessica Yung, “How python implements dictionaries.” <https://www.jessicayung.com/how-python-implements-dictionaries/>. [Online].
- [14] D. K. Bryan Ford, Pyda Srisuresh, “Peer-to-peer communication across network address translators,” *ATEC ’05 Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.
- [15] Adam Ierymenko, “Zerotier manual.” <https://www.zerotier.com/manual/>. [Online].
- [16] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

Acknowledgements

Il mio primo ringraziamento va al professore Jordi Domingo Pascual per avermi seguito e consigliato durante tutto il mio percorso da tesista. Ringrazio inoltre il professore Fulvio Risso per la disponibilità e professionalità dimostrata in questi mesi.

Vorrei inoltre ringraziare il Politecnico di Torino e la Universitat Politècnica de Catalunya per avermi concesso la possibilità di vivere quest'ultimo anno del mio percorso universitario in una città fantastica come Barcellona.

Grazie alla mia famiglia spagnola, che di spagnolo ha ben poco: Alberto, Aldo, Gloria, Luca e Manuela. E' stato un piacere condividere con voi le lunghissime pause pranzo e tutti i momenti che hanno reso l'ultimo anno uno dei più belli della mia vita.

Un ringraziamento speciale a Valentina, che in questi ultimi mesi mi ha spronato a dare sempre il meglio di me. Grazie per le tue attenzioni e la tua positività. Grazie per aver creduto in me anche quando ero io stesso a non crederci.

Ringrazio i miei amici del Poli: Alessandro, Carlo, Lorenzo, Simone B. e Simone C. Grazie per aver condiviso con me i momenti più belli e più brutti di questi ultimi 5 anni, dal primo fino all'ultimo esame.

Grazie agli "amici di giù": Arianna, Leonardo e Manuel. Voi che nonostante la distanza mi siete sempre stati vicini e che ogni volta che ci rivediamo, anche dopo mesi, è come se non ci vedessimo da un paio di giorni.

Ringrazio i miei nonni, che mi hanno cresciuto nonostante fossi una gran rottura di scatole. Voi che siete stati praticamente dei genitori aggiuntivi per me e non potrò mai ringraziarvi abbastanza.

Grazie a mia sorella Roberta per aver sempre tenuto alta l'asticella che mi ha spronato e motivato sin dai primi anni della mia vita da studente. Sei stata, sei e sarai sempre per me un punto di riferimento.

Il ringraziamento più sentito va ai miei genitori, i quali hanno sempre supportato le mie aspirazioni, credendo in me più di chiunque altro. E' a voi che dedico questo sudato titolo. Siete sempre stati per me un grande esempio di integrità, impegno e perseveranza.

Vi voglio bene,

Francesco