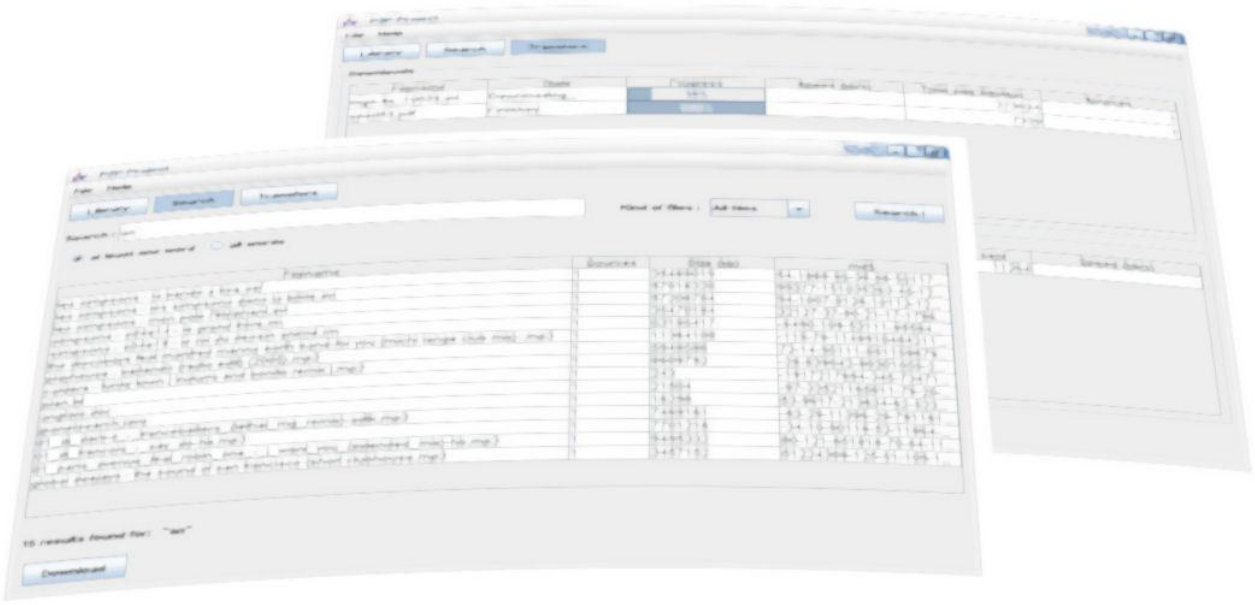
Glasgow Caledonian University – Erasmus Program 2004-2005

DEGREE PROJECT

REPORT



DEVELOPMENT OF A PEER-TO-PEER SYSTEM

Nicolas BRIGODIOT – François PERCHE – Julien ROBBE
Supervisor : Martin GALLACHER

Abstract

Our project consisted in creating a new peer-to-peer system. This included choosing a network architecture and writing client and server's applications adapted to the network.

Indeed we had to write a server application that would accept clients' connections and manage the files they were sharing, and a client application that would let users share their own files and download files from several other network peers at the same time.

We have chosen to write these programs using JAVA, and followed a full software engineering process to assess our development. This included implementation phases, but also testing and validating phases, not mentioning debugging phases to clear the few bugs we had.

Working on this project was very interesting for us, as we discovered many new things that will help us later in our career and in future developments.

Acknowledgements

Thanks to Martin GALLACHER, our project supervisor, for his help and his support.

Thanks to Eddie GRAY for his welcome on our arrival at the university.

Thanks to all the students who helped us testing and validating our project.

Finally, thanks to the authors of the external libraries we have used in the implementation of our programs.

Contents

Introduction

With more than half of ISP's bandwidth used by them, peer-to-peer networks have to be popular. And useful, to a certain extent. This project aims at creating a new peer-to-peer system.

Indeed, when we had to choose the project's subject, we thought developing a new peer-to-peer network would be something very interesting. We all had worked on different kinds of projects before, including programs with artificial intelligence or creation of a new language with its syntax and semantic parsers, but we never worked on a real network-based application. And isn't the point of such a project to discover new horizons?

Therefore we had to create a server and write a client that would let users share their own files, search for files shared by other peers, download some from other peers and even download one file from several clients at the same time.

Thus we created them, following a full software engineering process, from the project specification quickly mentioned above to the testing and validation phases, not mentioning development stages, such as implementing our solution, testing it, debugging it, or validating it with potential users; these users will be selected randomly, for the application is to be used by anyone, just like already existing peer-to-peer networks.

Working on this project has been extremely interesting, whether it is for the new scope of JAVA we had with developing a network, or for managing a project development from scratch.

This report will tell you more about our project development, the researches we have conducted and the development stages we went through, as well as the detailed design of our application, including full details on how the server works or how clients send files to each other. Finally, you will be able read about our conclusions, and see why we think this has been a great project for us.

I. Project Development

1. Investigation of context and other products

The very first thing we had to do was to examine other existing peer-to-peer systems and the environment in which they evolved. This was what was going to give us a precise idea of what we wanted to achieve and how we wanted to do it.

1.1. Clients features

The first stage consisted in having a look at the clients connecting to the existing networks, in order to see which features they all provided. This includes *KaZaA*, *eMule*, *WinMX*, *LimeWire*, and some others. It soon appeared that they all shared almost the same functionalities: basically, the ability to search for a file by its filename and of course, to download files from other peers. A few clients offer some additional features, such as a chat, an explorer that can open and read downloaded files, statistics about uploaded or downloaded data... But obviously, the two critical operations of a peer-to-peer client are searching and downloading files.

Then, we started studying how these worked. Some clients offer the possibility to search for specific kinds of files, such as audio or video files; search results would display differently depending on the kind of search that was performed. For example, searching for audio files would result in displaying the artist, the title or the length of the music, while searching for video files would result in displaying the resolution or the codec of the video file. It is also possible in some of the clients to search for an audio file by its author or title, or even type of music, using the audio tags contained in audio files. But this proved not that efficient, for not all audio files have their tags filled up. Finally, one of the clients associated a file category to each of the users' shared files: for example, when you shared a file, you could define in which category it belonged, such as action, horror, thriller for movies or personal, landscapes or celebrity for pictures. But this again proved not too efficient as this category was chosen by the user and thus could easily end up with many hoaxes on the network.

1.2. Different kinds of networks

Each client is associated to a network. One can connect to a given network using one of multiple clients though, the same way one can connect to the Internet using one of several browsers. The most famous and most used networks appear to be *FastTrack*, *eDonkey*, *BitTorrent* (although this one is a bit different than the others), and *Gnutella*. See the table below for examples of clients using these networks (fig. 1).

P2P networks	Examples of clients
BitTorrent	ABC, Azureus, BT++, BitTornado, Shareaza
Gnutella	Gnucleus, Limewire, Morpheus, Shareaza
FastTrack	Kazaa, iMesh, Kazaa Lite, Poisoned
eDonkey	eDonkey, mlDonkey, eMule, xMule

Figure 1. Examples of clients associated to the most famous P2P networks

Our study of these networks showed that they could be classified in three kinds of networks.

1.2.1. Centralized networks

The first peer-to-peer networks were called centralized. It was the case of *Napster* or *AudioGalaxy*. Such networks have only one server, managing connections between all the users (see fig. 2). Having only one server has some advantages: besides the fact that it is easier to set up than multiple servers or any other technology, it also ensures that any search request sent by a user will hit the entire network (for the only server will get the request and "ask" all the network clients). This is a real issue, and really the only real advantage of such a network. The main disadvantage is that if the server is gone for some reason – may it be technical or judicial – then the network is gone. Moreover, because all the network traffic will pass through this only server, it is quite easily overloaded.

We thought this kind of network would be a great starting point for us to develop our own peer-to-peer application; indeed, having only one server makes it easier to set up, and once it would be up and running, it would be possible to transform its architecture into a semi-decentralized one (which is really more robust and more realistic nowadays for there are no public centralized networks still running).

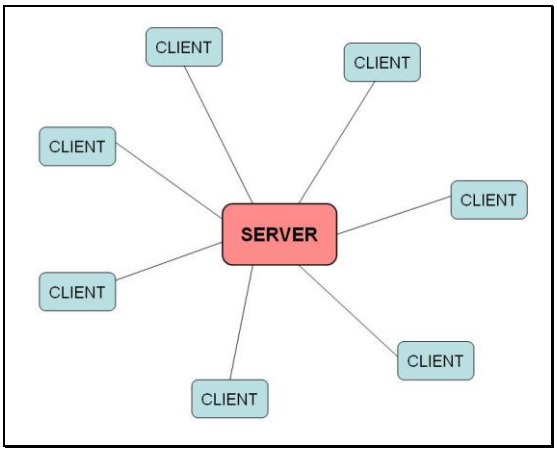


Figure 2. Centralized networks architecture

1.2.2. Semi-centralized networks

Because of the instability of centralized networks, due to their unique server, a next generation of peer-to-peer networks was born and they were called semi-centralized. For example, eDonkey and FastTrack are semi-centralized. In these networks, there is not one but several servers (see fig. 3). A client will connect to one of the servers, and if one of them is down, there should be many alternative ones still running. This makes the whole network extremely difficult to shutdown, and that is probably the best advantage of such networks. Another advantage is that, as the network is growing up, setting up a few more small servers suffices to keep the whole network going without being overloaded. Servers are able to talk to each other when processing search requests, but because of their large number, they cannot talk to every other server, thus resulting in requests not hitting the entire network. Actually, not even 75% of the network will receive the request. This is a great disadvantage of semi-centralized networks. The most advanced networks will let users choose the depth of their search request (i.e. the number of servers to transmit the request), but there is no chance that the request will hit the entire network.

This architecture is currently the most commonly used in peer-to-peer communities, but it is reaching its limits as there are always more and more people joining in. But because the third architecture is too recent and still a work-in-progress, we believe semi-centralized networks still have some time ahead of them and thus stay a good value anyway. That is why we have chosen to head for this kind of architecture, even if we thought the best thing to do was to start with a centralized one and then modify it to transform it into a semi-centralized one .

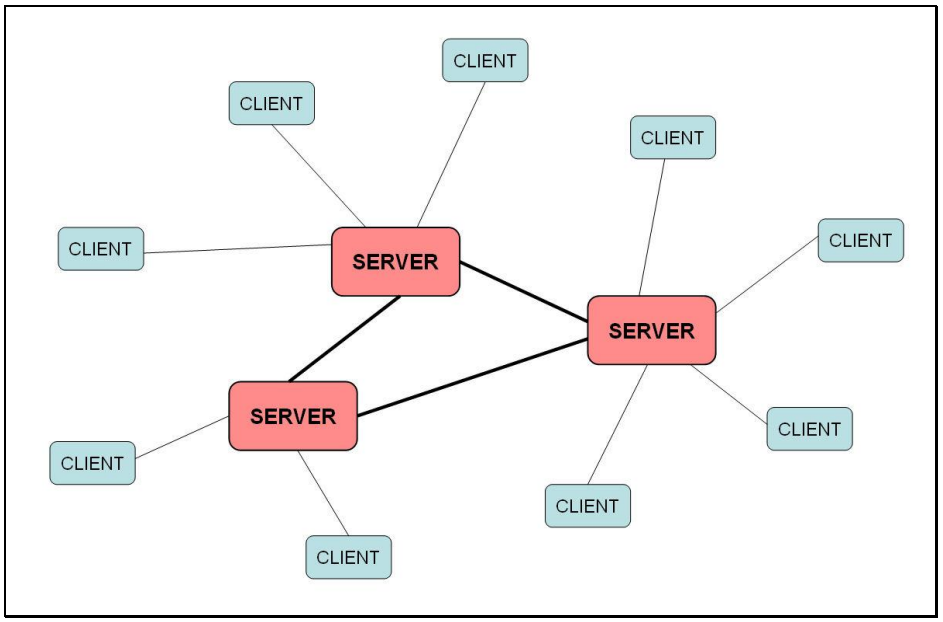


Figure 3. Semi-centralized networks architecture

1.2.3. Decentralized networks

The next generation of peer-to-peer networks is still new and still in testing. It is clearly the most interesting architecture on paper, but it still has to prove itself valuable. In this architecture there is no server at all (see fig. 4). Clients connect directly to other clients. This implies the existence of some service to tell the clients who are the last active peers they can connect to. This is quite a big disadvantage as this service must not be shut down otherwise clients will not be able to connect to newest peers. Another disadvantage is that, because there is no central server, search requests will, like in semi-centralized networks, not hit the entire network. This is particularly true in decentralized networks where searches never hit even half the network. The greatest advantage though is that such networks are virtually impossible to shut down; as long as there are two users online, they will continue to exist.

We found this architecture was the most interesting one. It provides the strongest networks, but it is quite difficult to set up and unfortunately, mainly because not all the users who connect have huge bandwidth, networks tend to be easily overloaded. We think these networks are not mature enough and thus, we were not going to start developing our own peer-to-peer application over an unstable and difficult to set-up architecture.

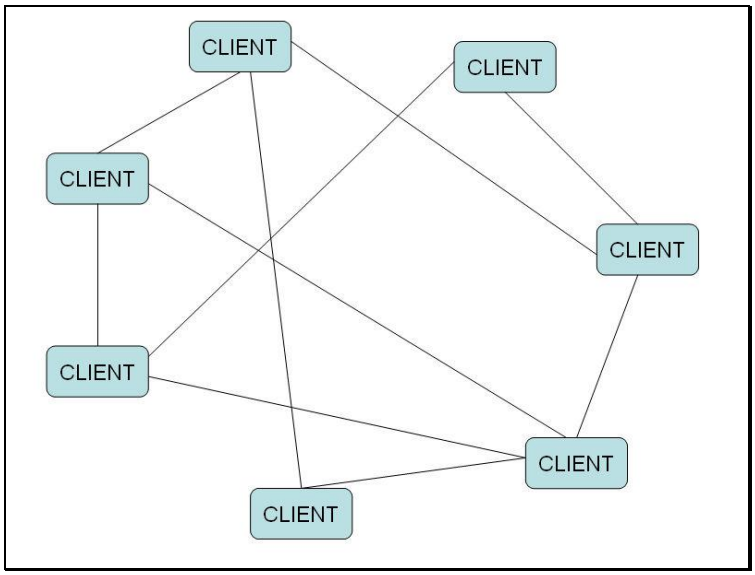


Figure 4. Decentralized networks architecture

1.2.4. Summary of possible networks and the reasons for our choice

When we had to choose which architecture we were going to use, we quite quickly crossed the decentralized one from the list. For technical reasons, because it is the most difficult one to set up and manage, but also for practical reasons, mainly because the networks built on this architecture are not efficient enough at the moment.

Therefore we had to choose between a centralized architecture and a semi-centralized one. In the real-world, centralized networks are clearly obsolete. In the public domain that is. Their unique server makes them far too vulnerable to all kinds of attacks, or even technical failures. Semi-centralized architecture has become what we could call the default architecture for most peer-to-peer architectures so far.

Hence our purpose was to create a new network based on a semi-centralized architecture. But for various reasons we decided to start with a centralized one. First, we did not have enough computers available, and a semi-centralized network requires several servers, which required the same number of computers. Also, and that would be the main reason, the semi-centralized architecture is not very different from the centralized one. Only the server really differs, client applications are quite the same. So, since the semi-centralized networks could be obtained from centralized ones, and since centralized ones are way easier to create, we saw no reason to start with a semi-centralized rather than a centralized network.

1.3. Peer-to-peer networks context

Once we knew which kind of network we were going to create, we had to know who would use it because this would determine some technical aspects of the program.

We started investigating how existing networks were used before deciding for ours. Obviously, the choice of the architecture does not depend of the end-use of the network. Architecture only evolves as time goes by; first generation is no longer used, second generation is currently overused and the third generation is approaching. So, what makes a network different to another? And we are not talking about clients' features here, but really about networks properties. For example, what makes the current networks so slow? Either they are slow because of a low transfer rate, or because of a long queue before the download actually starts, overused networks such as *KaZaA* or *eMule* are often too overloaded and therefore very slow.

In the case of long queues, this is fairly obvious: the number of users connected is too high. Unfortunately, there is not much we can do about this. *BitTorrent* does provide a limitation on the number of users downloading the same file, *via* the number of users allowed to connect to the tracker of the file. But as you would surely have noticed, we have not been talking of the *BitTorrent* protocol until now, as it is quite different from usual networks such as those we studied. By definition, there is no actual *BitTorrent* "network". This is the reason why one cannot do files searches using BitTorrent. Anyway, in the case of classic peer-to-peer networks (centralized, semi-centralized and decentralized), the number of users connected cannot really be monitored, and even less controlled.

Though, low transfer rate is an issue we can surely do something about. Well, there is something we cannot do anything about: it is users connecting to the network with a low bandwidth connection, such as 56k dial-up or even 128k ADSL connections. Even 512k connections, which are the most common connections, do not provide a great upload rate. The only actual thing that can be changed in peer-to-peer networks and that affect transfer rate, besides protocols, is the amount of data transferred per block. Tests have shown[1] that it was best to send the least data possible per block. Well, the least data possible in a reasonable way. It appeared that blocks of 128k were a good average, being not too large or too small.

[1] These tests have been conducted by KaZaA and eMule owners through the first versions of their products.

But this applies to conventional networks, hosting clients with connections as fast as T1 or T3 and as slow as 56k. We thought that was an inheritage from the older days when these networks were created and when there still were a lot of slow connections. That is why we decided that our own network would favour faster connections. That is, at least 512k which is really nowadays the “standard” bandwidth for most users.

In order to decide what size the transferred data blocks should be, we performed tests (see appendix C, *Files transfers, speed comparisons*) and ended up thinking that 512k was a nice size for blocks. This represents a decent amount of data quickly transferred with nowadays connections (less than 10 seconds to download on a 512k connection, and about 30 seconds to upload on most 512k connections which have a 128k upload bandwidth), and still possible to transfer with lower bandwidths.

Now we had critical elements about the program we wanted to write, all that was left, prior actually starting writing the application, was to decide how we would manage the project within our timescale.

2. Project Planning

Although we never studied any software engineering process in France, it was fairly obvious that after some research about software engineering processes, that we had to split our work into several phases: analysing the problem and obtaining a description of a solution, implementing this solution and finally testing then validating it, seemed to be the main stages for every module of the application that had to be developed.

Our first task was then to split the work into modules that could be developed separately. Clearly, there were a client and a server to write and they could be written separately. But it soon appeared that, even working with interfaces to make sure the two classes would be able to communicate, joining the two works together created problems. That is why we all worked on creating the client-server basic model, then we could add things to it separately without having to wonder if that was going to fit in or not. So, actually, we did not really plan what had to be done in a given time, mainly because we did not know what had to be developed long before we actually had to write it; instead we just added features here and there and that let us work alone when needed and we were able to concentrate on larger tasks.

Actually, there were a few larger problems we knew of from the beginning and that would need to be settled as soon as possible because of the critical features they would provide for the application. This includes files transfers, the ability to split files into several parts downloadable separately and mainly all features of the server, such as being able to update the database correctly, to execute search queries, etc.

That is why we have chosen to write a small sample application using critical features before writing up the actual application. For example, we had written a small program that could transfer a file beginning transferring the end of the file and then finishing with transferring the first parts of it. Then we performed tests with these classes, and wrote the actual classes that would be contained into our application.

Then we tested them in two stages. First, we tested each module alone; that is what is called unit tests. The point is to make sure the class itself is working correctly. Once we were sure the class could not fail in any way, we added it to the main program and performed integration tests; we were sure the class was working, now we had to make sure it was working with the other existing classes.

You can see main phases of development (from the choice of the subject to the testing and validation phases) on the following scheme (fig. 5).

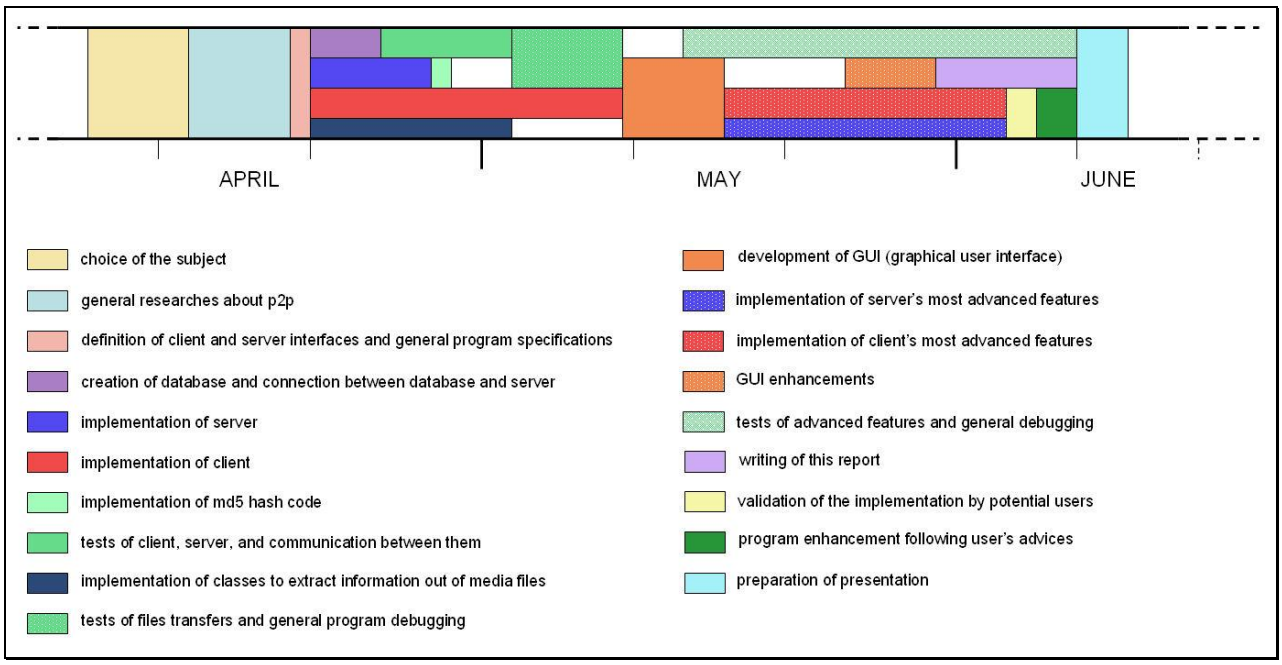


Figure 5. Timeline of project development

3. Development stages

We can quickly distinguish several stages when we look at the timeline above. This section will tell you more about each of them.

3.1. Researches

The first main development stage was to conduct researches on peer-to-peer networks in general. When doing that, we browsed the Internet to find elements that could be of interest to us. We found a number of interesting websites that let us discover which kind of peer-to-peer systems already existed. The results of our researches have been discussed in the first section of this chapter. We then wrote a summary of these results and submitted it to our supervisor who validated it and then we started implementing our solution to the problem.

3.2. First implementation of client and server

The second phase of the development consisted in setting up our server, therefore having the database up and running and writing the server application, as well as writing up the client application. You will find details of how we implemented it in the *Detailed Design* chapter.

To work on that, we have used *JCreator* and *emacs* as IDE's and JAVA as our development environment. We have chosen JAVA as a programming language because it is clearly the language we know the best. Not mentioning its portability, and that it is clearly a language taking more and more importance each day. To help us with the development phase, we used, as usual, Sun's API documentation, *javadoc*.

The tests of what was being written were performed during the development phase to make sure everything was going smoothly. You will find more details about how the tests were conducted in chapter 3 of this report, *Testing*.
We did not validate the written modules with users as we did not have any real clients or users waiting for the product.

3.3. Development of the GUI

Once we had a server and a client working together, we started working on the user interface. To read more details about the GUI, please go to chapter 2, *Detailed Design*, section 4.

We did not use anything new to create our screens, except maybe a pen to design them on paper first and find out which layouts should be used to lay out the components. We have coded the interface from scratch, not using one of these modern programs creating UI's by dragging'n'dropping components. We believe writing the GUI ourselves result in something closer to what we want, and at least we know exactly how it works so we can modify it without any problem.

We did not really test the user interface, other than looking at it on the paper to see if there were enough controllers to do everything it was supposed to.
We did not validate it with anyone either, for the same reason that we did not validate our first implementation of client and server.

3.4. Implementation of advanced features

The fourth phase of the development consisted in upgrading our server and our client to implement more advanced features, like multisourcing. Again, you will find details about this features and how they have been implemented in chapter 2, *Detailed Design*.

Writing these new features was similar to writing the first version of our applications in terms of the tools used. The only thing we needed was to have access to more computers to have a network large enough to permit multisourcing.

Once again, tests were performed during the development phase and you will find more details about them and the debugging phases it led to in chapter 3, *Testing*.
Though this time, when the advanced features where fully implemented, we validated the whole application with potential users. Read more about the validation process in chapter 3, *Testing*, section 2.

II. Detailed Design

1. Client-Server Protocols

1.1. The role of the server

The server does not do anything during file transfers. Files are never sent from a client to another by passing through the server; otherwise this would not be a peer-to-peer system! Therefore, the role of the server is quite limited.

Indeed, the server is going to make the link between clients... and that is about all. Basically, a client connects to the server, sends search requests and the server replies with the results of the search, nothing else. Once the client has its results, it does not need the server anymore.

So let us see in details what the server is able to do. It is able to receive a list of files the client just decided to share in order to add them to the database. In the same way, it is able to receive a list of files the client was sharing and decided to stop sharing. The server can also perform a search in the database and send to the client the results of the search. Finally, it is able to search for more sources for a given file and send them to clients that are downloading the file and in need of more sources.

1.2. Protocols

Protocols need to be clearly defined. Indeed, they are what allow clients and the server to communicate together. Without a well-defined protocol, for example if the server is waiting for the client to tell him something while the client is also waiting for the server to tell him something, no network-based program can work at all.

We have chosen to use keywords, or commands, that the client would send to the server to let it know what was going to be sent and what was expected back. Obviously, we needed one keyword per action described above.

1.2.1. Client sharing new files

When the client is sharing new files, it sends the keyword UPDATE_SHARED_FILES to the server. The client is then expected to send the number of files it is going to add to the database. Knowing that, server can reply by sending to the client the first unused index in the database (see section 2, *Server's database* for more details about how the database works). Then it gets a bit tricky. Client is now expected to send four *Strings*. But each of these contains \n characters, which means the server will stop reading them as soon as it will find one of these *return* characters. To avoid that, server must read *Strings* until it reads the keyword END OF FILE. Then it can start reading the next *String* or just go on if that was already the fourth of them. See fig. 6 for a summary of the protocol.

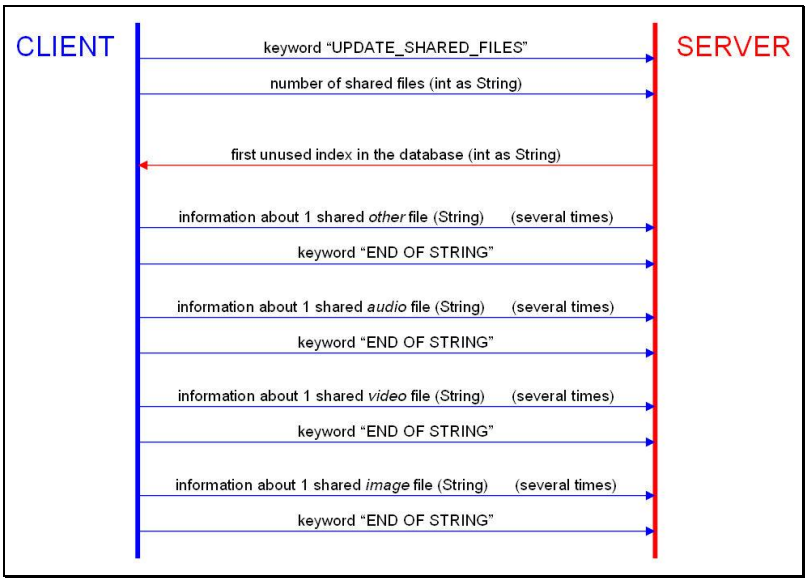


Figure 6. Updating shared files protocol

1.2.2. Client stops sharing files

When the client stops sharing files, it sends the keyword REMOVE_SHARED_FILES to the server. The server then expects a *String* containing md5 hash codes of files to be removed. See fig. 7 for a summary of the protocol.

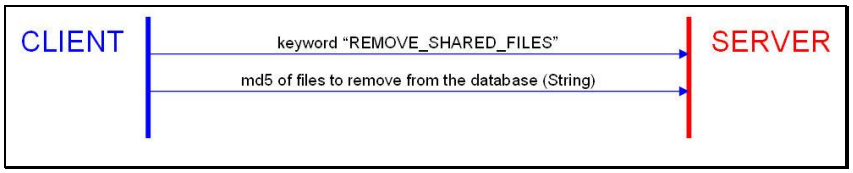


Figure 7. Removing shared files protocol

1.2.3. Client asking for its IP address

Because it seems JAVA is not able to retrieve the IP address of the local host's network card used for transfers (it always returns the loopback address), the server must be able to give a client its IP address on demand. When a client wants its IP, it just sends the keyword GET_MY_IP to the server which replies with the client's IP and port. See fig. 8 for a summary of the protocol.

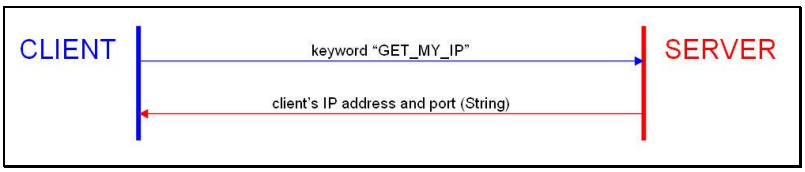


Figure 8. Client asking for its IP, protocol used

1.2.4. Client asking for more IPs addresses sharing a file

When a user wants IPs addresses of clients sharing a file, it sends the keyword MORE_IPS to the server, followed by the md5 of the file. Server performs the appropriate search (see section 2, *Server's database* for more details) and then sends back a list of IPs to the client. See fig. 9 for a summary of the protocol.

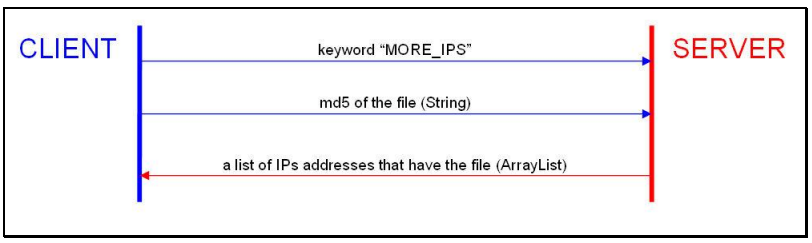


Figure 9. Client asking for more IPs for a file, protocol used

1.2.5. Client performing a search by filename

The keyword a client must send to the server when performing a search for a file is SEARCH. Then it sends an integer corresponding, according to a chart, to the kind of search he wants to do; more precisely, it depends on the kind of files it wants to search for. This integer should be 1 for all files, 2 for audio files, 3 for video files and 4 for image files. After that, the client is expected to send the name of the file it is looking for. Then it should send to the server a *String* which value will tell whether the server should look for all of the words or at least one word in the filenames. This *String* should be "allWords" or "oneWord" (which does what being quite obvious). Then server is able to look into the database, and can send back to the client a list of results. See fig. 10 for a summary of the protocol.

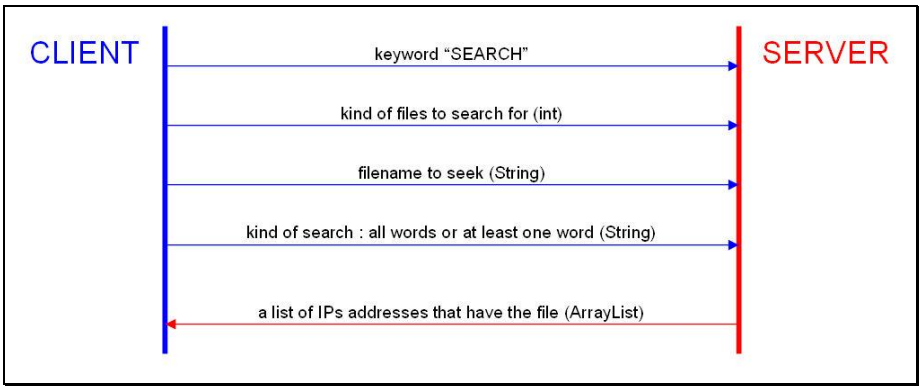


Figure 10. Search protocol

1.2.5. Client disconnecting from the server

When a client disconnects from the server, whether it is because he explicitly asked for a disconnection or because he closed the application, it sends the BYE keyword to the server which then closes all opened sockets properly. See fig. 11 for a summary of the disconnection protocol.

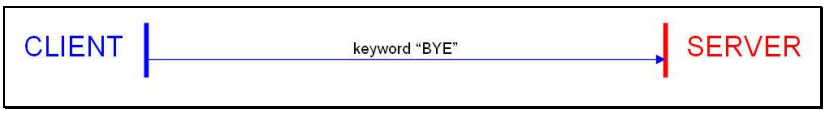


Figure 11. Disconnection protocol

2. Server's database

As we have seen above, the server machine also hosts a database – although it could be hosted on another computer but there is no point in doing this, not mentioning the fact it would be way slower – which contains all the information about all the files shared by all the users. We have tried several architectures for the database, but what we thought the best solution was to use four tables.

2.1. Database tables

Indeed, since there are four kind of files (audio, video, image, others), it makes sense to have four tables in the database; one table for each type of file. Well, not exactly. The *others* table contains all the information shared by all kinds of files, that is their filename, file size, and their md5 hash code. The other tables, called *audio*, *video* and *images* only contain additional information about these specific kinds of files. Like the *audio* table which has *bitrate* and *length* columns. See the table's creation scripts in appendix D, *database table's creation scripts* (p. 52).

Since there is one table per kind of files but all files are in the same table, named *others*, each file in the *others* table has an id that is reported into the other tables. Don't get confused, and see fig. 12 for a scheme summing it up!

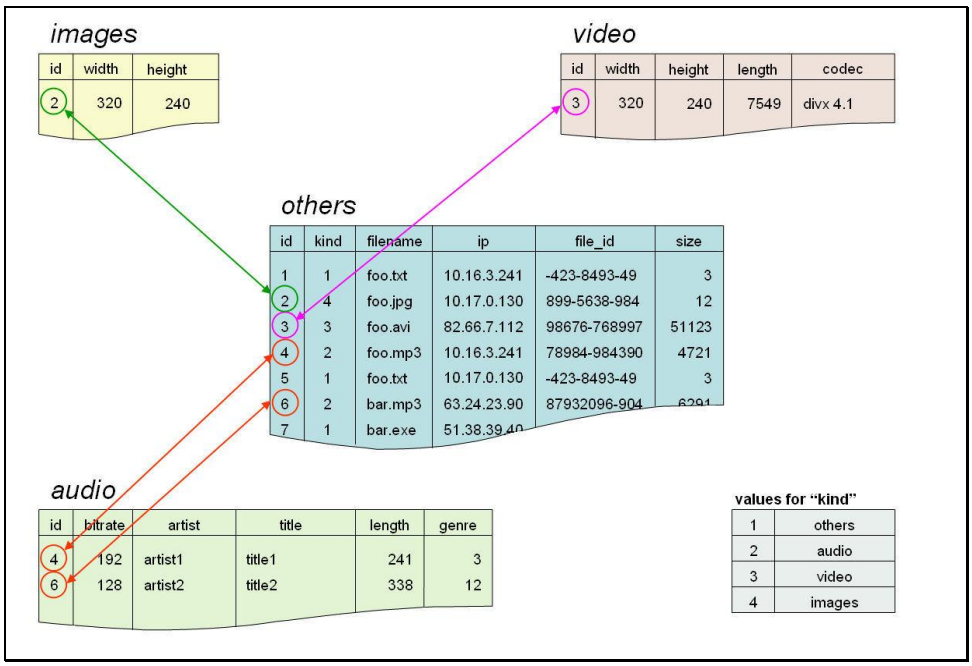


Figure 12. Relations between database's tables

2.2. Accessing the tables on requests

During its normal uptime, the server is very often prompted by clients to update the database or to search for a file. You will find in this section an explanation on how requests are built up and processed by the server.

2.2.1. Adding files

When a user decides to share a new directory, it scans it and analyses every file it contains. For all files, client calculates their md5 hash code and retrieves their size; then, it collects additional information depending on the file type. As the client gets the information, it adds them into four *Strings* to be sent to the server. In every case, these *Strings* are formatted as follows:

```
file1info1    file1info2    file1info3    etc. (\n)
file2info1    file2info2    file2info3    etc. (\n)
                          etc.
```

It is important to start a new line with each file's information, and that data about one file are separated by tabulation characters. Indeed, this is the format directly recognized by *MySQL* when importing data files into tables.

So, you might have guessed it, once the server gets the *Strings*, it writes them down to files on the disk and then executes the *MySQL* command `LOAD DATA INFILE <filename> INTO TABLE <table>` to load data in tables.

2.2.2. Removing files

After a client has shared a directory, it can decide to stop sharing it. In such a case, it just sends a *String* containing the md5 hash codes of the files contained in that directory to the server. Codes are separated by semicolons. Thus, the *String* is formatted as follows:

```
md5-1-764829-48etc;md5-2-4834-etc;md5-3-48903-8etc
```

The server then retrieves each md5 from this *String* and, for each of them, executes the four queries listed below (fig. 13). The first three delete entries in the tables in case of an audio, video or image file, while the last one deletes the entry if the file was classified as "other".

The important decision made here, was to choose to execute four queries per file to be removed, while we could have stored somewhere into server's memory the kind of each file, and thus execute only one query. But we thought it was best to save server's memory, first

because it has to run fast at all time, and due to the large amount of files that usually ends up in this kind of network, its memory could easily get overloaded. Also, executing queries on a database engine is extremely fast compared to the time needed by the java virtual machine to retrieve the kind of a file in a large array or in whatever we could have used to implement that. In all cases, it would have represented a lot of memory, a lot of data, and searching through it would have taken too much time. And it is extremely important that the server runs as fast as it can.

```
DELETE FROM others, audio USING others, audio
WHERE others.id = audio.id
   AND ip = clientIP
   AND file_id = md5;

DELETE FROM others, video USING others, video
WHERE others.id = video.id
   AND ip = clientIP
   AND file_id = md5;

DELETE FROM others, images USING others, images
WHERE others.id = images.id
   AND ip = clientIP
   AND file_id = md5;

DELETE FROM others
WHERE kind = 1
   AND ip = clientIP
   AND file_id = md5;
```

Figure 13. SQL requests executed when removing a file from the database

2.2.3. Getting more IPs addresses sharing a file

When a client downloads a file, it might (most likely, it will) happen that there are no more sources for the download (i.e. all people sharing the file have disconnected from the server). In such a case, the client will have to ask the server for more IPs addresses from people sharing the file, in order to resume the download and hopefully complete it. To do so, the client just sends the md5 of the file to the server. Then the server just has the database execute a simple query (see fig. 14) which returns all the IPs addresses of all computers currently sharing the file.

```
SELECT ip FROM others WHERE file_id = md5
```

Figure 14. SQL request asking IPs addresses sharing a file described by its md5

2.2.4. Searching for a file

There comes the probably most important part in the role of the database. Before it can search for a file, the server needs to know the kind of files to look for and the words to seek into the filenames. It also needs to know whether it should look for all the given words or just one word into the filenames. Once it knows all that, it builds up the query to execute on the database.

There are basically two kinds of queries. Those looking for audio, video or image files, and the others. What changes is the tables to look into and the columns to select. In both cases, the query will have to use the *MySQL* `regexp` function to search for words into the filenames. Though, depending on whether we are looking for all words or just one word, the condition will be different. In the case of a search on all the words, it is necessary to check whether the filename contains each of the words, one after the other. This gives us a condition on the filename, for example:

```
AND filename regexp word1
AND filename regexp word2
        etc.
```

In the other case, when we are looking for just one word in the filename, the condition is as follows:

```
AND filename regexp (word1 | word2 | etc.)
```

In this case, *regexp* is validated as soon as one of the words in parameters is found in the filename.

See fig. 15 below for a scheme detailing how the whole request is constructed.

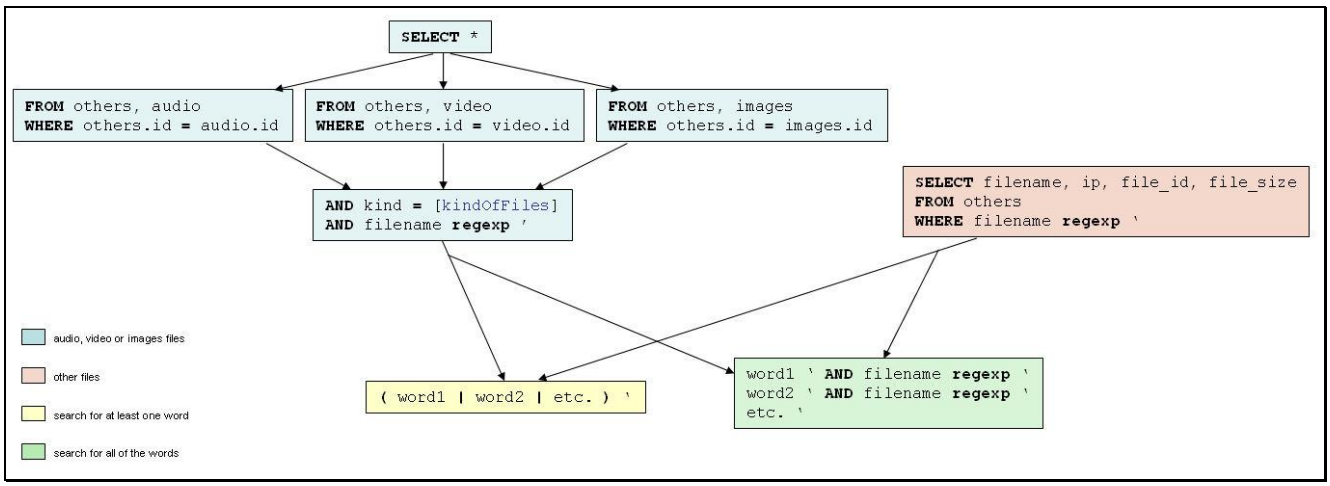


Figure 15. Building up the SQL query executed when searching for a file

3. Key technical aspects

You will find in this section explanations of the most advanced (or so we estimate them) aspects of our implementation. Let us begin with how downloading work; but before we start, you should make sure you understand the general process of downloading a file. This is explained in appendix E, *General downloading process*, page 54.

3.1. Multisourcing, downloading in a multithreaded environment

First of all, what is multisourcing ? Multisourcing is the term used to describe the ability to download a file in several parts, usually all of a given size, and that from several peers at the same time. For example, a client on a network supporting multisourcing can download the first half of a file from a peer and the second half from another peer. This speeds up downloads as the client's downloading rate is not limited by the unique peer's uploading rate, usually very smaller than downloading rates.

Thus, the key for implementing multisourced downloads is to have the files split in a number of small parts. Our tests have given us a size for these parts, which is 512kb (see chapter 3, *Testing*, and appendix C, *Files transfers, speed comparisons*). It is also crucial to have each part of each file downloaded in a particular thread.
To manage that, we created a *DownloadManager* which, as its name says it, manages each download. One instance of this object is created for each file being downloaded. When instanced, it knows – *via* its constructor parameters – all the information about the file to download, except who has it. It is the client who just "adds" to the *DownloadManager* IPs addresses of peers having the file.

Once the *DownloadManager* has a list of IP addresses, or even just one, it can start downloading. It uses an instance of *BitSet* to remember which parts have been downloaded and which parts have not. Now, what is a *BitSet*? Well, basically, it is nothing more than an array of *boolean*. In our case, if the first part (index 0) of the file has been downloaded, we set the first bit (index 0) of the bitset to true. That way, the *DownloadManager* is able to determine, at any time, which parts are still to be downloaded.

Once a part is downloaded, it is written to the file on the disk. There comes the most tricky part of the operation. The thing is, there is only one file on the disk, and there are several threads writing to it. Sometimes, two threads will try writing to it at the exact same time. This is why the whole section of code writing to the file is declared *critical*. This means that at a time, one and only one thread can execute this code. That way, if two threads try to access it at the same time, the first one to execute the first instruction will lock the whole section of code. The second thread will just have to wait until the section is unlocked, which will not happen before the first thread has finished executing all the instructions concerning the writing on the file. This is a problem of concurrent access to a data, and this is something very common in multithreaded environments like this one, or in databases in general (hence the *lock* series of SQL instructions).

3.2. Keeping the user interface updated

Another problem was to keep the GUI (Graphical User Interface) correctly updated at all times. The real problems occurred in the transfer panel (see section 4 of this chapter for more details about the GUI) when user decided to remove a line from the table of downloads.

When user starts a new download, the instance of *ClientImplementation* adds a line in the panel; it notes which line it is (its index in the table) and then passes it as a parameter when creating the instance of *DownloadManager* associated with this line. This way, when the *DownloadManager* needs to update the line of the table for some reason (a new part has been downloaded, or the download has been paused for example), it knows which line to update. It is also *ClientImplementation* which will update the line for the last time to say the download is finished.

So, at a time, it can be either *ClientImplementation* or *DownloadManager* which can update the line in the transfer panel of the GUI. Problem occurs when user cancels a download. When doing so, *JPanelTransfers* must be able to access the *DownloadManager* actually managing the download displayed on that line. To do so, it keeps a list with references of all corresponding *DownloadManager* instances in chronological order, to all of the lines of the table of downloads. For example, if user cancels the download that is on the third line in the table, *JPanelTransfers* can get a reference to the *DownloadManager* handling the download by looking at the third index of this list. Then it is able to get the md5 of the file downloaded and to remove it from the files being downloaded.

But that is not all; now, the download that was displayed on the fourth line went up in the table and now populates the third line. Hence the necessity for the *JPanelTransfers* to update its list of references to the instances of *DownloadManagers*, and then to update the associated line of each *DownloadManager* accordingly.

4. Graphical User Interface in details

Although we started developing the GUI quite late in the overall development (see section I.2. for project planning), we still started to think about it early in the engineering process. Actually, it was quite hard not to think about it when looking at existing clients and their GUI, and that was on the very first days of our project. But anyway, we had to wait until some client's features were written and working before we could really start writing a GUI.
Indeed, only the client does have an interface. The server is run in command line, and has a log file and a console for all its communication needs. He is alright, servers are used to that way of life... Clients though, run by regular people, really need a highly intuitive and easy to understand interface. But the interface must still be adaptive and able to accept new buttons or new features as the application keeps growing up. This section describes our program's GUI, its features and why we think it fits its role.

4.1. First sight: global layout of the window

The window of the application contains several components. It has a menu in the top, and then buttons lined up. These buttons are contained in a *JPanel*, which means it is possible without any problem to add or remove buttons from there. Then, under these buttons, is a panel hosting some other components, depending on the button selected. So there are actually several panels here, layered on a *CardLayout* and buttons on the top are here to switch these panels. Finally, under this (these) panel(s) is another line of buttons, which this time depends on the panel displayed just above them.

So it is really easy to add panels and their controllers in this interface. Examples of panels that could be added are one with a chat, another with an explorer of downloaded files, etc.
But for now, let us do with the three existing panels. As the buttons say, they represent the library, search and transfer panels. The first time the application is launched, only the library

is available. Indeed, it does not make much sense to let the user access the search or transfer panel when not connected to the network yet. Once it will connect to the server, the two other panels will become accessible.

See what the first window you see when you fire up the application looks like on fig. 16 below.

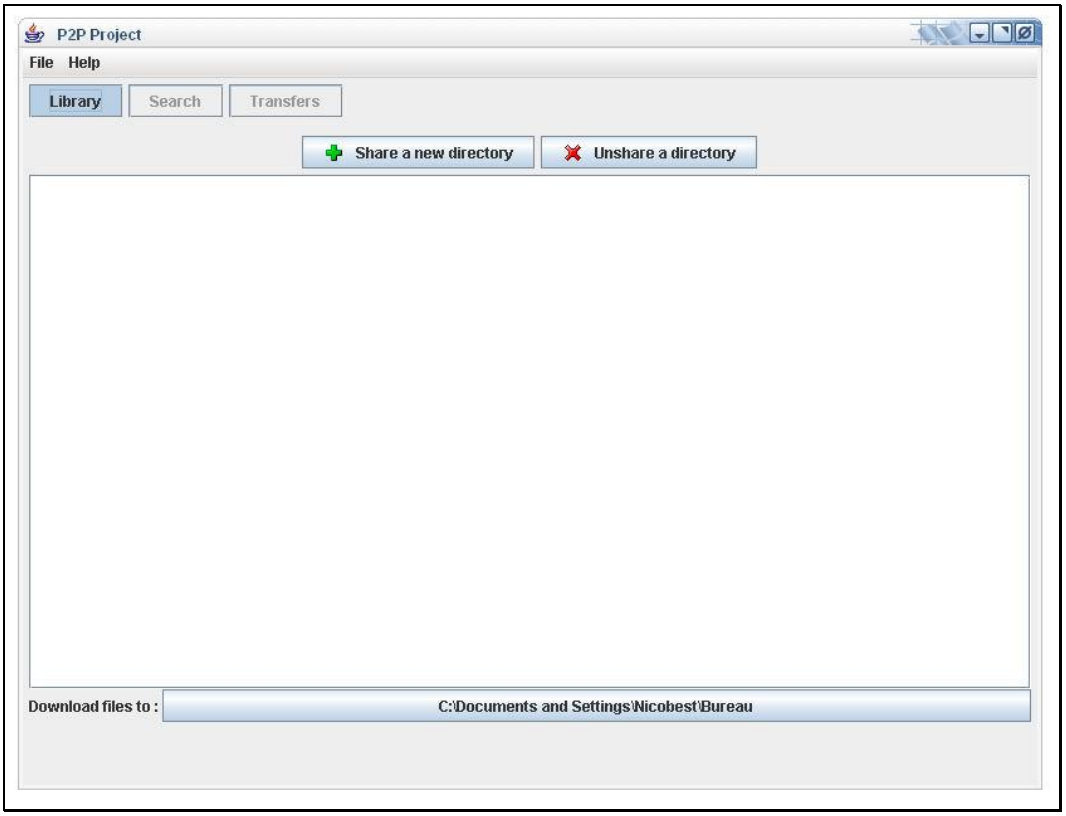


Figure 16. Main window of the program's GUI, currently displaying the library panel

4.2. Window's menu

Now that we have this window, let us do something with it. For example, let us see what is hiding in the menu bar. Well, not much actually, but still something vital for the program: the *Connect* option from the *File* menu will connect to the server. There is also a *Disconnect* option in that menu, as well as the traditional *Quit*, unfortunately quite abandoned since the arrival of the little x in the top right corner of the windows many years ago. Anyway, you will also find in the *File* menu an access to the *Preferences* window, which is described a bit later in this section. The other available menu, *Help*, only contains the traditional (once again) *About* option. See fig. 17 below to have a look at these menus.

Figure 17. Application's menus

4.3. Convincing the user that the program did not crash

Now the user has chosen to connect to the server. Because of *Swing*'s (strange?) way to manage the events and the execution thread, the GUI would freeze when connecting to the server which can be a long operation (up to 15 seconds on a Windows system). To avoid the "I erase the window just by moving it" syndrome, we added a modal progress bar continuously moving while the client was establishing the connection. This also works for other long operations such as sharing a large directory. See examples of these windows on the figure below (fig. 18).

Figure 18. Modal windows appearing during long processes

4.4. Setting up user's preferences

Whenever the users will decide to open the preferences window *via* the window's menu, they will discover the window you can see below on fig. 19. This window lets them change the address of the server, which of course is not suggested but may still be of some use some day. Also, the users can check a box if they want the program to connect automatically to the server at next start-up.

The input fields are checked by the program when the users click on the OK button, and an error message is displayed whenever one of the IP field is not a number or is a number not included between 0 and 255 or the port is not a number or not a number included between 1 and 65535.

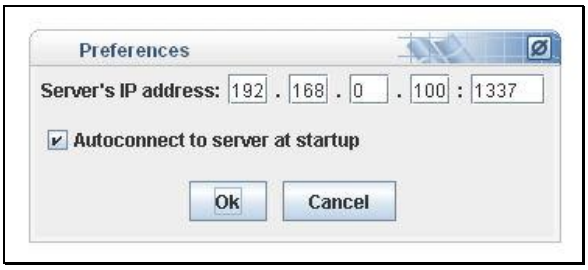


Figure 19. Preferences window

4.5. Sharing files with the library panel

At any time, users can choose to share some directories or to stop sharing some others. They just click on the corresponding button on the library panel. When sharing a directory, it adds its path into the list. Obviously, when unsharing a directory, it removes it from the list. Of course, users cannot add two times the same directory (in which case an error would result, see fig. 20 below), nor they can share a directory which path is invalid.

The last button they can click on is the one under the list. Clicking on this button will open an explorer that will let them choose a directory where they want to save the files they are going to download. Once again, it is not possible to choose a directory that does not exist.

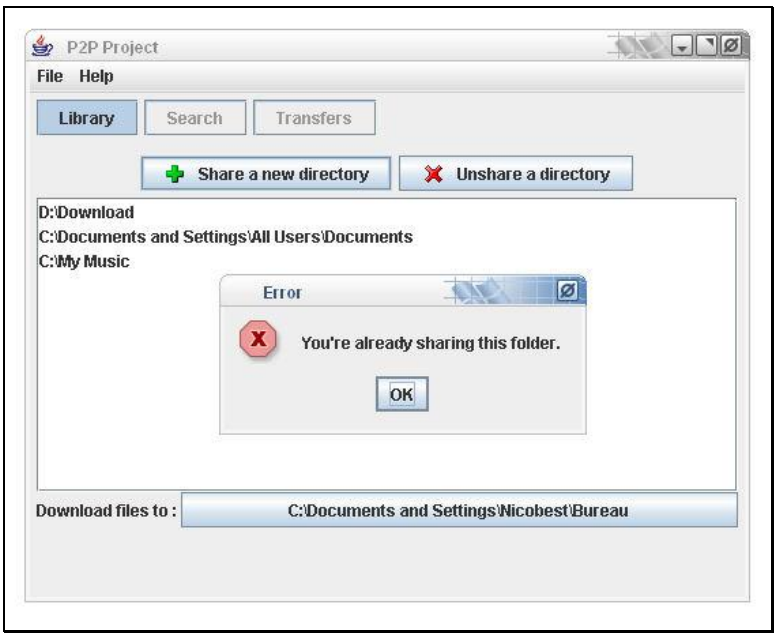


Figure 20. Library panel in action; here user tried to share the same directory two times

4.6. Searching files within the search panel

Once users are connected, they can access the search panel by clicking on the *search* button in the top bar.

The search panel offer the ability to choose from a list the kind of files to search for, and two linked radio buttons (linked in the sense that only one can be selected at a time) let the users choose whether they want to search for all the words they enter or at least one.

Results are displayed into a table while the number of results is written under the table. Of course, the columns are modified to fit the kind of files we are searching for. When users click on the *Download* button while a line is selected (or if they double click on the line), it launches the download and therefore automatically switch the view to the transfer panel. It is also possible to order the lines per ascending or descending order per column. See fig. 21 and fig. 22 below for examples of how the search panel works.

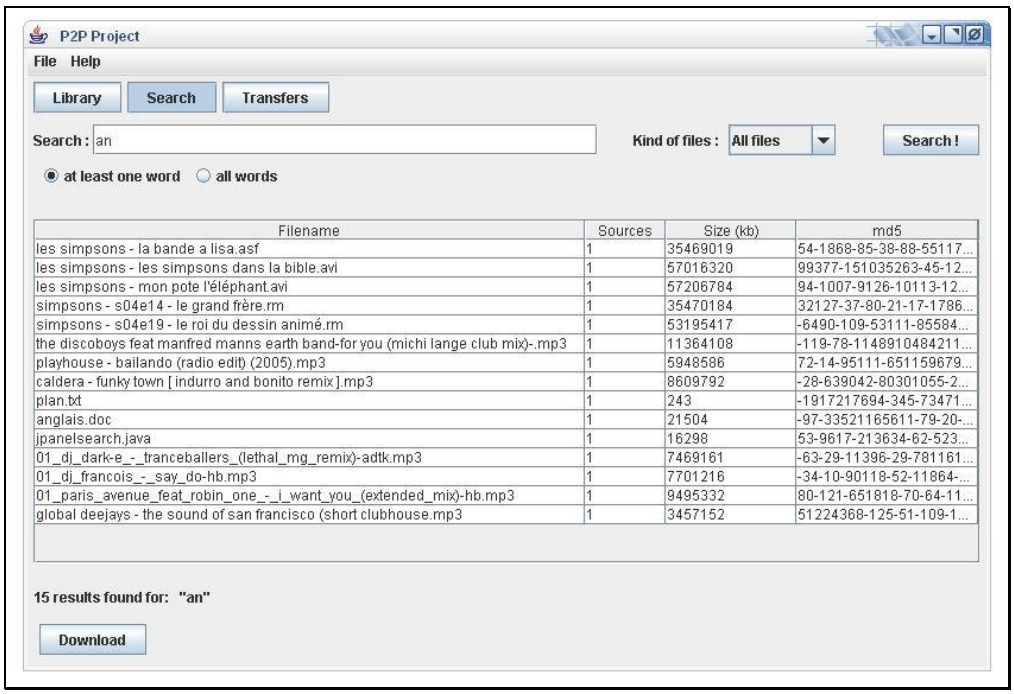


Figure 21. Results of a search performed on all files and on keyword "an"

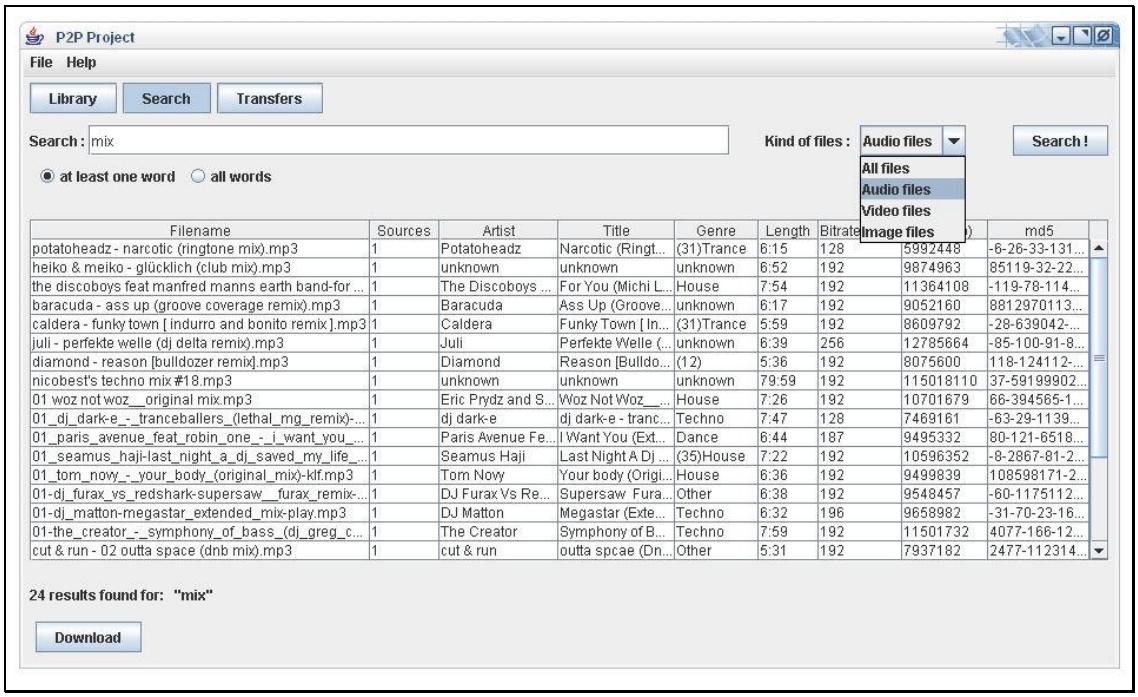


Figure 22. Results of a search performed on audio files only and on keyword "mix"

4.7. Monitoring transfers with transfer panel

Launched a lot of downloads, have we? Now it is time to see how they are going! Clicking on the *Transfers* button will display the transfer panel. This panel contains two tables; one for the files being downloaded, and the other for the files being uploaded. A line corresponds to a download or an upload. In the case of a download, a progress bar indicates the progress of the transfer; the number of sources is displayed too, and a column displays the state of the download, which may be “connecting”, “downloading”, “waiting for sources”, “paused” or “finished”. Unfortunately, there is no transfer rate displayed as we were not able to calculate it.

Users cannot interact with the upload table (which is the one at the bottom, because less important than the download table to the user), though they can interact with the downloads. They can stop a download and resume it later. They can also cancel their downloads, in which case the line is removed from the table and the file is deleted from the disk while the download is stopped.

See fig. 23 below for a screen capture of the transfer panel in action.

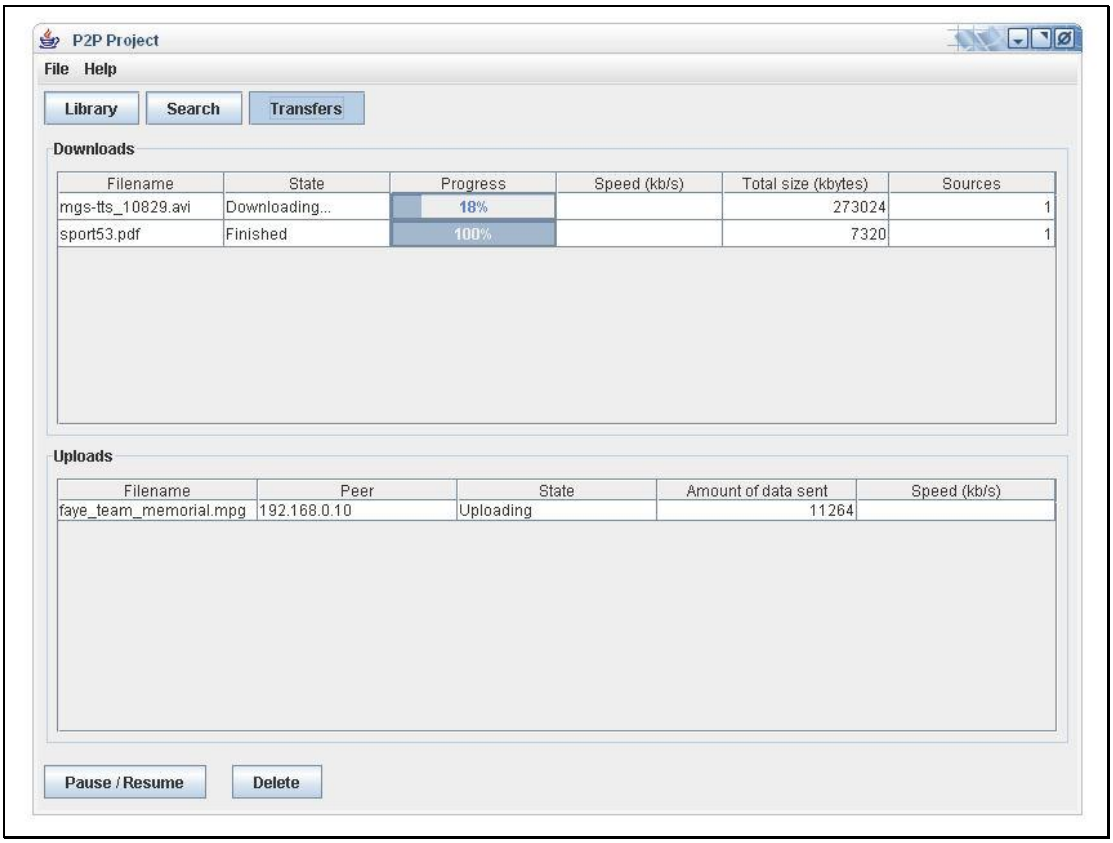


Figure 23. Transfer panel of the GUI, downloading two files and uploading one

4.8. Our choices during the development of the GUI

We have explained through this section how the user interface worked and what it looked like, and sometimes we have even explained why we did it that way and not differently. But not always... for example, why have we chosen to use a list of directories to represent the library? Well, actually, our first plan was to make some kind of a tree showing users' directories, with checkboxes in front of each directory. Users would have just expanded the tree and checked the repertories they wanted to share. It would have been faster for them to choose directories. Unfortunately, we have tried doing this, but we have been unable to have it work. The component, that is. JAVA does not provide a "tree with checkboxes" by default, so we have been able to create one, but actions were not working on it. So after many tries, we decided to go for a simple list of paths, which is also the solution chosen by *WinMX*.

All the other components were quite obvious, and we cannot see which other components we could have used to make the search and transfers panels.

Now that you know everything that is to know about the user interface, go on to the next section and finally discover the full specifications of our program!

5. Full functional specifications

You surely have noted through the previous chapters what our program was capable to do. This section summarizes it but also explains what does not work because we could not implement it.

5.1. Skipped features

There are indeed a few things we wanted to implement and that we have not been able to. Mainly due to lack of time, but also in a few cases because we could not find a way to do it.

The first thing we would mention is that the transfer rate is not calculated. This is quite annoying in such an application where the transfer speed has its importance. Unfortunately, JAVA provides no way, for what we have seen, to retrieve the transfer rate from a socket connection; and since we're just sending buffers *via* abstract objects, we let the JAVA Virtual Machine manage the packets to send and we therefore do not control the network layer enough to evaluate the transfer rate. The only thing we could have done is to calculate an average speed, based on the time spent and on the amount of data transferred, but this is quite useless information actually and we did not have much time to spend on it anyway.

In the same way, progress bars only moves when a whole block has been received. This means that on slower connections, it will not move constantly but rather suddenly. This is also because we cannot control how the Virtual Machine sends the data.

Another thing quite annoying we did not have time to do is to save files' information on disk. Indeed, right now, when users share a directory the program retrieve all the information about all files, which may take some time when there are a lot of files. What we wanted to do was to save these information somewhere on the disk so when the user reconnects to the server, it does not calculate them again. Programming this would not have been as easy as it looks though, because a number of things can happen, like files have been added to the directory, removed from it, or maybe some filenames have been swapped (e.g. foo.txt has been renamed in bar.txt and bar.txt in foo.txt, and of course contents of both file are completely different...).

This is about it for the features that have been skipped. Fortunately, there are a lot of things working, and some of them are really great, or so we think, so we are quite happy to bring you to the next section to show you the full specification of our program.

5.2. Full program specification

Indeed, the application is capable of some great things. But let us begin with the beginning, and let us start with general clients' features.

5.2.1. Saving users' settings

The application saves users' settings in a file on the disk. The file is named *preferences.dat* and you can learn more about it in appendix F, *Saving users' settings*, page 56. Using this file, the application is able to resume downloads at its launch; it remembers the server's address and also the shared directories and the path where downloaded files should go.

5.2.2. Scanning files

The program scans shared files to calculate their md5 hash code and retrieve some information about them (size, tags of audio files, resolution of video files, etc.). Using md5 hash codes of files ensures users always download the file they want and not a fake file. It also lets the search engine regroup files by their md5, thus letting users download a file from multiple peers even if all the peers have the file named differently (for example, *Rolling Stones – Satisfaction.mp3*, *the_rolling_stones_-_satisfaction.mp3* and *01-Satisfaction.mp3* will all be considered the same file).

5.2.3. Sharing files

Users are able to share their files through the networks just by selecting the path of the directory they want to share. They can share as many directories as they want. The shared files are automatically uploaded to the peers who are interested in them. Users can at any time stop sharing their directories, or even add more directories to the list. Server will be updated even if clients are already connected to it.

5.2.4. Searching for files

Users can search for files using quite a few useful options, like the possibility to search for only audio, video or images files. They can also search for all the words they entered or just at least one. Searches are very fast. Results are displayed into a table; its headers depend on the kind of search users have done. Lines can be ordered by any column's ascending or descending order. If they happen to find the file they wanted, they can select it from the list and click on the *download* button to initiate the download. They can also simply double click on it and this will work too. In both cases, they will automatically be brought to the transfer panel where they will be able to see their download start.

5.2.5. Downloading files

In the transfer panel, users can monitor their downloads and their uploads. They cannot interact with the uploads. Though they can stop, resume and cancel their downloads. Each download line let the users know the progress of the download, the number of peers they are connected to while getting this file among some other less important information such as the file's name or size.

5.2.6. Multisourcing

Indeed users can download a file from several peers at the same time. This speeds up greatly the average download speed as it is not blocked by one single peer's maximum upload rate. Clients can ask the server for addresses of more peers sharing the file being downloaded to add even more sources and download it even faster. Once there are no more sources, it will ask for more to the server every 8 seconds.

5.2.7. Client's crashes prevention

A lot of cases where clients could crash have been noticed and treated as needed. Usually the process that caused the error is aborted and an error message is displayed. But the client application is supposed to never terminate abnormally. The more sneaky potential problems like a peer disconnecting while a client is connecting to him, have been settled. We believe the client application to be stable enough to do its work.

5.2.8. GUI features

You have seen the features of the user interface in the previous section, but let us summarize them up here. The GUI offers an intuitive and easily understandable way to share files, to search for them and to download them. We added details for user's convenience, such as the ability to double click to start a download or to type *enter* to validate a search. Modal windows with a progress bar also show up during long treatments to let users know that the application is still alive. All popup windows are centred relatively to the client's window, which is itself initially centred on the screen. The main window is totally resizable, as the components will adjust their size to keep a nice looking interface. The GUI also does look the same on Windows and Unix systems (which is far to be always the case unfortunately).

5.2.9. Portability

Well this is not our doing really; it is more JAVA that is portable and thus letting our application run on any system having a Virtual Machine installed. But we have tested our client application on Windows (most of the time) and Unix systems, and it worked well on both of them. Actually we had to modify some little things to have it work perfectly on Unix systems, this is why we are mentioning our application's portability here.

Now let us see what the server's features are.

5.2.10. Storing files' information

Server uses a database to store all shared files' information. This assures quickness in searches that we for sure could not even think of approaching if we were using JAVA's objects to store this information. Plus a database can contain way more entries as it stores them on the disk and not in computer's memory. The database used here is *MySQL 4.1* and the link with the JAVA program is made using ODBC. Thus, the current version of the server is made to run on a Windows system, but it is really easy to change the ODBC driver and replace it with a jdbc connector in order to have the server run on any other system (Windows, with its average uptime, is surely not the best platform for running a 24/7 server).

5.2.11. Linking clients together

The server is used for linking clients together. That is, send to a client the addresses of other peers. This way, the server is not used at any time during actual files transfers between clients. Server shutting down would not stop the current clients' downloads. We consider the amount of work done by the server minimal right now, and this is important to have the server execute the least operations possible. Indeed, in an actual peer-to-peer network, there are thousands of clients asking for something at the same time to the server. Having clients do the more they can is crucial for server's reliability.

III. Testing

Because testing is a very important step in software development, we performed as much tests as possible. In fact, as soon as a module or a class was written, it had to be tested, and, most usually, debugged. When we were sure that the application was running perfectly as it was supposed to, technically speaking, we had to have it tested by some people who had never seen it yet, and have them give us their feelings about it. This section describes how we managed both the verification and the validation processes.

1. Verification of the Project Implementation

The first series of tests consisted in testing the server implementation to make sure it would work as expected with the database. The first test ever was just to make sure the server could actually connect to the database and send requests to it. Fortunately, we worked on a quite big project involving database access *via* JAVA in France the weeks before we arrived in Scotland. Therefore, we had no real problem setting up the database and the drivers to make the link between our server and the database. We had no problem either establishing the connection between clients and server, as we were already quite experienced in network development too. So this did not required tests interesting enough to be mentioned here.

The first real test was about the client sending data to be inserted into the database to the server. It took us several tries to find a good way of sending the data to the server; data had to be written in syntax directly understandable by the database, in order to not have the server parse the data, which would have slowed it down. The tests consisted of looking to the database tables to make sure data has been well-received, and if not, try again until it is. Not very interesting, but efficient enough.

Then came what we could call the second series of tests; this time, it was all about file transfers. That was actually the first real tests we performed, using an actual set of tests, supposed to check on everything the program was supposed to be able to handle. You can find this set in appendix A, *Testing, first wave: files transfers in a mono-threaded environment* (page 42). At the time, we had written enough of the client implementation to

have it supposedly handle transfers of files from a client to another. We also had it written in a single trait; thus, it was more than likely that it would not work and some severe testing and debugging would be needed. Surprisingly enough, it was not working that bad. Actually, it almost worked on our first try! Though, there still was a problem, and it took us no less than a week to sort it out. The problem was the client (downloading the file) not reading the data stream sent by the peer (uploading the file) correctly. Hence the failure of the transfer and worse even, the crash of the application. You will find more details about these tests, this problem and how we solved it, in appendix A, page 42.

After that, having the transfer of a file between two peers working, we had to test transfers of files between a client and several peers. The code for that has already been written along the code for transfers from a single peer. It was only a matter of testing by then. This was our third series of tests, our second real set of tests. You can find this set in appendix B, *Testing, second wave: files transfers in a multi-threaded environment* (page 46). There came our first real issue with threads management. We never really studied how threads worked before, and we never really used them in our personal programming either. So it was quite a big discovery of an important element of JAVA (and an interesting one!); but it was a great source of problems, too. It was thanks to our tests that we found out, in the first place, that we were dealing with a threads management problem. First transfers worked very well, or so it appeared, but it was a comparison test between the source and the destination files that showed something was wrong. You will find more details about these tests, this problem and how we solved it, in appendix B, page 46.

Another kind of tests we did was performance tests. We timed the transfers of a given file using different settings, mainly different blocks size. Thus, we found out what we think were the best settings giving the best performances, not forgetting we were testing the application in a local network, which does not provide the same conditions for transfers' rate than distant connections, like they would be in a real context of use. You will find the results of our tests in appendix C, *Files transfers, speed comparisons*, page 50.

Finally, the last tests were tests decided "on the ground", to detect small errors of coding in the latest written classes. Other than the two problems described in appendices A and B, we did not encounter any other serious problem during the development of our application.

Therefore, not long after we performed the last tests of files transfers, we have been able to go for the validation of our application

.

2. Validation of the Project Implementation

We had our application tested by other people only by the end of the development. Since we had no real clients, it was useless to have it tested by people in the middle of the development to finally have them suggest us to do what we were going to do on the rest of the development period.

We have shown our client application to ten people. We tried to have people already used with peer-to-peer networks, and people that truly have never heard of it before. Well, actually, people that never heard of peer-to-peer networks before do not seem to exist. At least, not in this age range, for they were all students. Some of them were studying in computer networks, while some others were studying economics sciences, which is quite different. The least they knew about peer-to-peer networks was a quick view of *KaZaA*.

First, we showed them how it worked very briefly, then let them have a try by themselves. Their first reactions, while trying the program for the very first time, were very good as they did found out the user interface intuitive enough. Then, after a few minutes of testing, they told us how they felt about our program.

Most of them told us we did a great work, which is always nice to hear. Fortunately, they also gave us some things that they disliked.

One person out of the ten we had showed it thought that we should not switch automatically to the transfer panel when starting a download from the search panel. We decided to not change that however, as she was the only one to have the remark done, and because we think it is more annoying to not know if the download started after we have double clicked on the file, for example.

Almost half the people we questioned though told us they found it strange that we could not share a single file, instead of a directory and all the files it contains. This is an interesting point as we have thought about it before and concluded that sharing a single file was useless.

Maybe was that because we were already heavy users of peer-to-peers networks and we were used to sharing large numbers of files. But indeed, people discovering peer-to-peer systems might want to share just some specific files. However, we would have changed it so we could share files along the directories, but we did not have enough time left.

That was really the two only negative points they gave us, the rest being more like compliments about what we have done. Especially, it seems they have liked the user interface enough to not wonder where to click to do what they wanted to, and some details like having the audio tags in the search panel when searching for audio files impressed them.

Finally, the validation exercise was an interesting one. This was the first time we conducted one, and hearing feelings from people that have not worked on the application for hundreds of hours before testing it is quite different from what we really expected from them. It appeared too that they gave us some suggestions that we would have added if we had time, thus making us think that we maybe should have gone through this validation process sooner in the project development.

Conclusions and project evaluation

Evaluation of the product

Although we have not really reached our primary goal, which was to create a semi-centralized network, we are still quite happy with our result. Not long before the end of the development, we had enough time to either transform our network into a semi-centralized one, or to really enhance the application we already had to make it really robust and complete, but not both. We have chosen the second solution, and we think that was the best thing to do. Our final product is well achieved, and we would rather have a finished centralized network than a semi-finished semi-centralized network...

When looking back at it, the implementation of the programs went pretty smoothly. We had only two big bugs, the rest of the coding was quite linear and did not give us too much trouble. We must say, we were quite used to programming in JAVA, used to using the javadoc and we were not too bad programmers to begin with. We think the implementation itself is not bad, even though it could be probably better in several ways. For example, we used many "techniques" that are faster but not the best looking ones, like implementing *ActionListeners* directly in most of the user interface classes, or using a lot of internal classes.

Evaluation of the project development

Although we are satisfied with our final product, we are not too happy with how we conducted the development of the application. We worked way harder on the second half of the period than on the first half, which is a sign of a bad organisation, or so we think. We obviously underestimated the time taken by some parts of the development. But even now, we are still surprised by the time it took for some modules that we would have think done in a few hours when it took us really a few days. On the other hand, we thought the whole implementation for multisourcing would be a pain while it almost worked on our first try. Well, even if we blocked on the very single problem for almost a week! Now we know how hard it is to estimate how long the development of a given class will take.

As far as our planning is concerned, well, we did not really have a full planning established from the beginning, so we cannot say we followed it or we did not. But not having a clear planning established did not annoy us, and we do not think we would have done better with one. We still believe planning is important but in our case, the coding was linear enough so we did not need one. Though we knew, even if we did not formally write it down, what had to be done and we knew a bit about how hard it would be.

Also, we implemented a lot as our ideas came, meaning we did not write down on paper how classes should interact together, nor we truly thought about how to solve a specific problem. Usually, we had a vague idea of the solution and as we began implementing it, it appears to be working probably as well as if we had worked it on paper before. But again, this worked because this was still a quite simple application and we are aware of the necessity to not rush coding on larger projects.

Conclusion

We liked this project a lot. The subject itself was interesting, and implementing our solution learned us many things about JAVA, which is probably going to be one of the main programming languages we will use in our career; we also learned a lot about software engineering in general. Finally, working exclusively on the same project for months gave us a more precise idea of an actual project development, like in real life, which is quite different from working on a project at the same time than having regular courses, as we always did until now.

We believe this project has been very useful for us and that it will serve us in our soon-to-begin career...

APPENDICES

Appendices Contents

APPENDIX A

Testing, first wave: files transfers in a mono-threaded environment

Set of tests

In all these cases, we will refer to the client downloading the file as the “client”, and to the client uploading the file as the “peer”.
The UNIX *diff* command mentioned here takes two files and compare them byte per byte to tell if they differ or not.

Transferring a file in less than a block ☑

Client can receive a file smaller than the size of blocks transferred from the peer. ☑
The file received is the same than the file sent (use UNIX *diff* command). ☑

Transferring a file in a single block exactly ☑

Client can receive a file of the size of blocks transferred from the peer. ☑
The file received is the same than the file sent (use UNIX *diff* command). ☑

Transferring a file in several blocks ☑

Client can receive a file larger than the size of blocks transferred from the peer. ☑
The file received is the same than the file sent (use UNIX *diff* command). ☑

Transferring several files at the same time ☑

Client can receive two different files simultaneously from the same peer. ☑
The files received are the same than the files sent (use UNIX *diff* command). ☑

Full test of a transfer in a mono-threaded context

We began testing with transferring a file not larger than the size of a block; the transfers worked well. Then we tried transferring a larger file, large of about 15 MB, which represents around 30 blocks.

We had inserted console printing at almost every step to clearly see what was happening. It appeared clearly there was a problem of communication between the two clients when one blocked on sending data and the other crashed on reading data. As you can see on fig. a (see below, left), this client, which is trying to get the file, crashed because he tried to create a byte array of over 1GB. Therefore the problem occurred when reading the integer supposed to give the size of the buffer to create. We can see (although it is not clear with mixed lines due to threads running randomly) that the client was able to download the first part of the file... while the other peer did not finish sending it! Actually, the peer (fig. a, right) was blocked and would never quit "sending the buffer..."

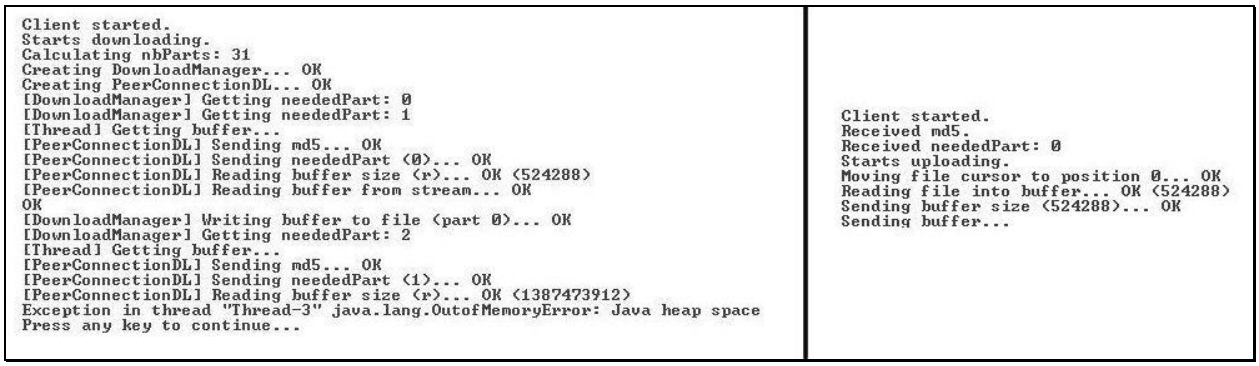


Fig. a: Console outputs of client (downloading, left) and peer (uploading, right)

So what happened there? We tried to modify the size of transferred blocks to see if anything would happen. And indeed, it appeared that it worked perfectly well with blocks smaller than... 3 bytes (instead of 512 kilobytes). It even worked with blocks of 3kb but only from say computer A to computer B, not from computer B to computer A. At first, it seemed to be random crashes, so it made us think there was a problem with the thread managing the download. Then, after more testing, it was clear that it would always work with blocks of a given size and never work with larger blocks. But we could not see where the problem was. We reviewed the code several times, ran it on paper, and everything was supposed to work well.

In such a case, the best thing we have found to do was to read (once more...) the *javadoc* of all the methods we used. Then we found out. Client downloading the file used the *read* method of *DataInputStream* instead of the *readFully* method to read the buffer sent by the peer. Both methods frankly have the same documentation, just written differently. We tried using *readFully* instead of *read* however, and the problem was solved.

Our explanation of the problem is the following: the peer sends the integer 524288 to the client using the *writeInt* method while the client uses the *readInt* method to retrieve it. In that case, the client does know that 4 bytes are to be read from the stream, and therefore will block until 4 bytes are available for input. Then the peer sends the byte array of 524288, i.e. it sends 524288 bytes to the data stream. Client reads the integer correctly and just after that, reads the stream into a buffer. Using *read*, it only reads the bytes available in the stream at the moment the method is called. It does not block until there are enough bytes in the stream to fill the buffer up. So client reads the stream and reads the very beginning of what the peer is sending at the same time – that was only 1 to 3 bytes on our computers, it would depend of the computer speed and the connection speed. Client writes this few data to the file and then goes for next part of the file. It reads the integer from the stream using *readInt*, so it reads 4 bytes of data – but they are actually 4 of the bytes from the buffer the peer is still sending. These fully depends of the file being transferred, and most likely the integer will be very big either in the positive or in the negative. If positive, client will crash when trying to create a buffer of such a size, and if negative, it will crash trying to create an array of a negative size. Peer will never stop sending the data because the stream is now broken. All that is resumed in the scheme below (fig. b).

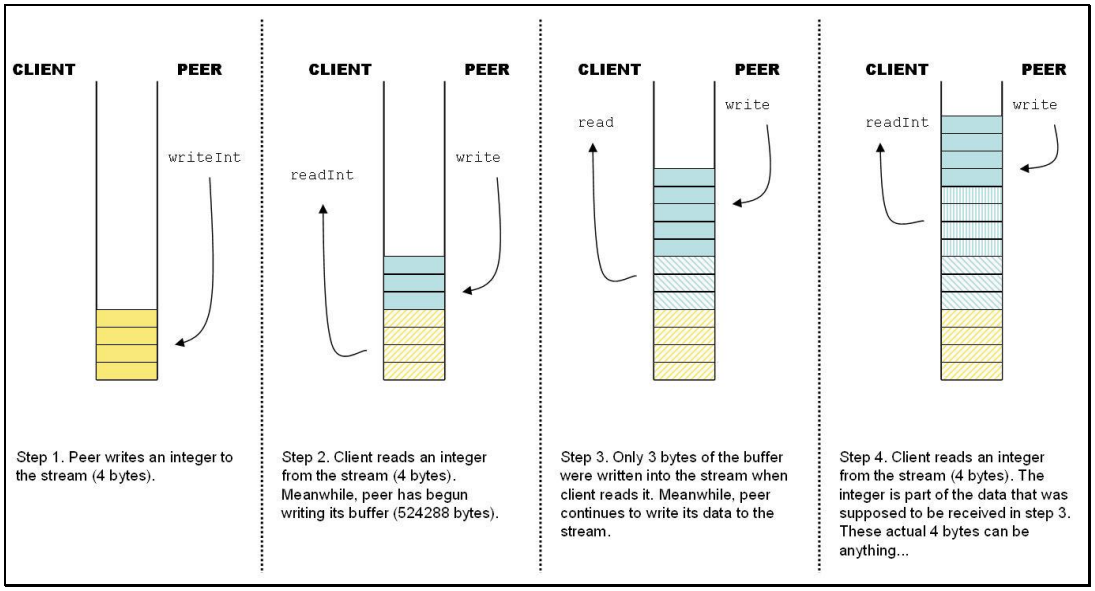


Fig. b: Miscommunication between 2 clients; explanation of the problem

APPENDIX B

Testing, second wave: files transfers in a multi-threaded environment

Set of tests

In all these cases, we will refer to the client downloading the file as the "client", and to the clients uploading the file as the "peers".
The UNIX *diff* command mentioned here takes two files and compare them byte per byte to tell if they differ or not.

Transferring a file in less than a block ☑
Client can initiate a download of a very small file by several peers simultaneously. ☑
The file received is the same than the file sent (use UNIX *diff* command). ☑

Transferring a file in a single block exactly ☑
Client can receive a file of the size of blocks transferred from several peers. ☑
The file received is the same than the file sent (use UNIX *diff* command). ☑

Transferring a file in several blocks ☑
Client can receive a file larger than the size of blocks transferred from several peers. ☑
The file received is the same than the file sent (use UNIX *diff* command). ☑

Transferring several files at the same time (one peer per file) ☑
Client can receive two different files simultaneously from several peers (1 per file). ☑
The files received are the same than the files sent (use UNIX *diff* command). ☑

Transferring several files at the same time (multiple peers per file) ☑
Client can receive two different files simultaneously from several peers (2 per file). ☑
The files received are the same than the files sent (use UNIX *diff* command). ☑

Transferring files in two ways simultaneously ☑
Client can receive files from a peer while sending other files to this peer. ☑
The files received are the same than the files sent (use UNIX *diff* command). ☑

Full test of a transfer in a multi-threaded context

The code used for transferring files from multiple peers to a client is the same than the code used for transfers from a single peer to a client (see appendix A, page 42). Therefore, if there were going to be errors, it would necessarily be due to the addition of multiple threads.

And there was an error. Transfers worked well; but when comparing the received file with the original file, it appeared that they differed. We first tried with small files, of the size of a few blocks. It usually worked well enough and the files were the same. But still, sometimes they differed. It was obvious, after more testing, that the bigger the file was (i.e. the larger number of parts there were to transfer), the greater were the chance that they differ. Although we knew they differed, it was still hard to know exactly in what they did. The UNIX *diff* commands does tell where they do but it is really too hard to establish a relationship between a number of offsets in the binary file and our programming.

The first thing we tested was to make sure the peers were uploading the correct blocks, i.e. the blocks asked by the client. We printed the md5 hash code of the blocks sent by the peers and compared them to the md5 hash code of the original blocks (we had the file split into parts of 512kb using the UNIX *split* command). It appeared that the blocks sent by the peers (fig. c) were the same blocks than the original file (fig. d, left). Then we splitted the received file into parts of 512kb and checked the md5 hash code of each part to determine where the problem occurred (see fig. d).

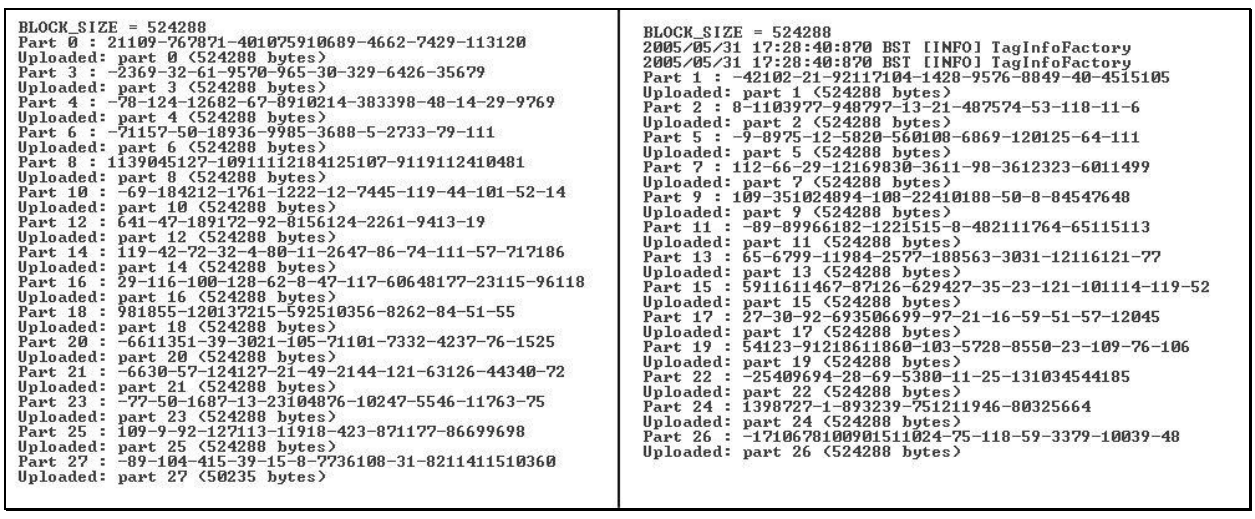


Fig. c: md5 hash code of blocks sent by the peers

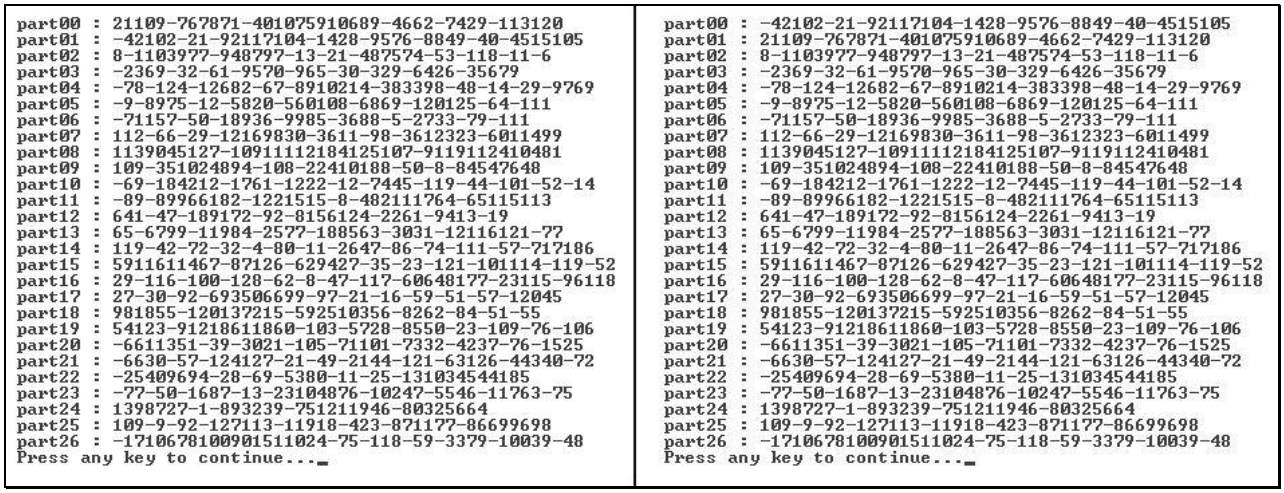


Fig. d: comparison between original blocks (left) and received blocks (right)

What appeared clearly was that the first two blocks have been swapped. Then we knew the problem was during the writing-to-the-file process. Consequently, we thought there was a concurrent access to some data by the multiple threads. We thought obviously of the *RandomAccessFile* object and the *part* variable as they were the two objects used when writing to the file. Thus, we tried synchronizing threads and accesses to these two variables, but it did not help. Then, it suddenly came to us: the two instructions using the *RandomAccessFile*, *seek* and *write* (the first to move the cursor at the right place in the file, the second to actually write to the file) had to be critical, as they should always be executed together and not interrupted by the processor to let another thread modify the instance of *RandomAccessFile* during the process. Otherwise, a thread could move the cursor to a place in the file, then another thread could move it to another place and the first thread would then write the data at the wrong place (see scheme below, fig. e, explaining how the blocks inversion mentioned above happened).

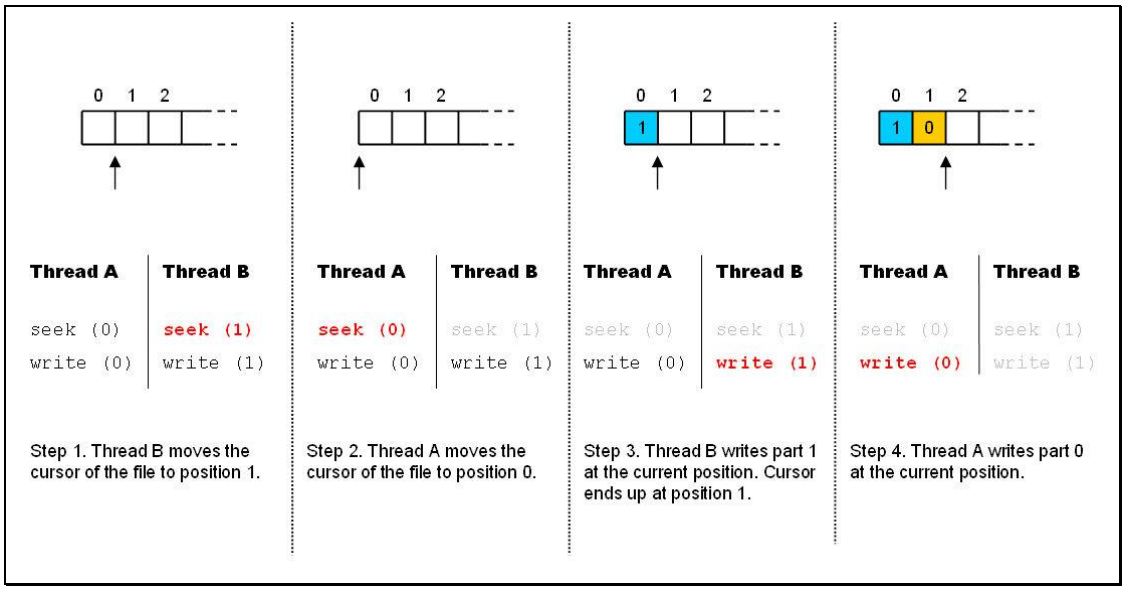


Fig. e: concurrent access to a file in a multi-threaded environment; example of what can happen

APPENDIX C

Files transfers, speed comparisons

Performances tests

Here you will find measures we have taken in order to have some general ideas about how long a transfer of a given file should take, and how long it actually takes using our application.

We used these tests for determining what would be the best size for blocks to be transferred, and using this size, to compare the time taken by our application with the time taken by operating systems – which we consider minimal – for a single file transfer.

The files transferred weighted 50 MB and 700 MB. In all tests, we are skipping the connection (between client and peer) time, which is totally dependent of the system (can be up to 15 seconds on a Windows system, and instantaneous on a UNIX system).

	File size : 50 MB		File size : 700 MB	
Block size	Duration (s)	Avg speed (MB/s)	Duration (s)	Avg speed (MB/s)
256 kb	40	1.3	540	1.3
512 kb	30	1.7	430	1.7
1024 kb	20	2.6	280	2.6
2048 kb	13	4	210	3.4
50 MB	10	5.1	125	5.7
Using Windows Explorer	5	10	70	10

Fig. f: speeds of transfers depending of blocks size

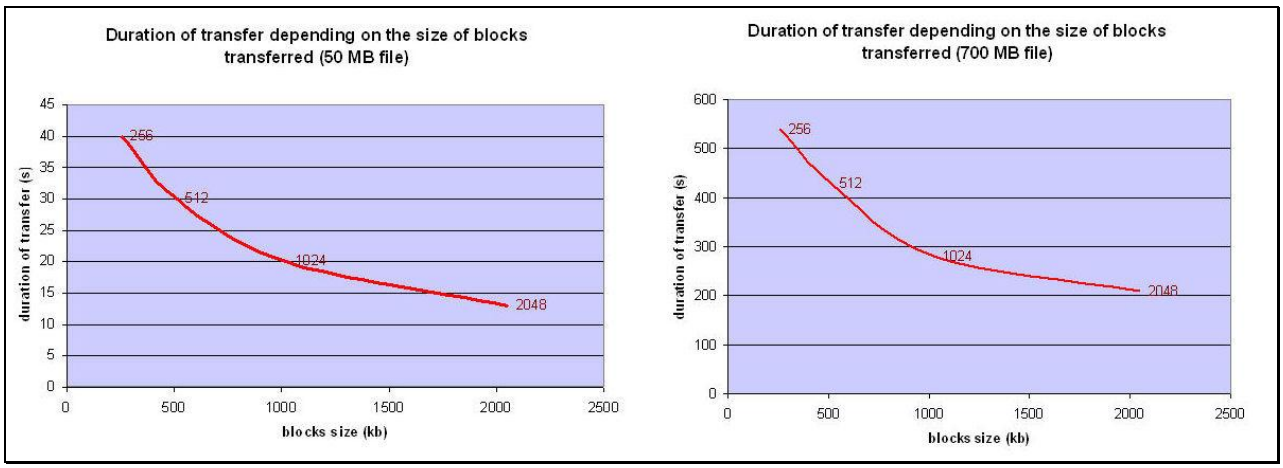


Fig. g: graphs of duration of transfers depending of blocks size

APPENDIX D

Database tables creation scripts

Audio table creation script

```
create table audio
(
     id             integer not null references others(id),
     bitrate        integer,
     artist         text,
     title          text,
     length         integer,
     genre          text
);
```

Video table creation script

```
create table video
(
     id             integer not null references others(id),
     width          integer,
     height         integer,
     length         integer,
     codec          text
);
```

Images table creation script

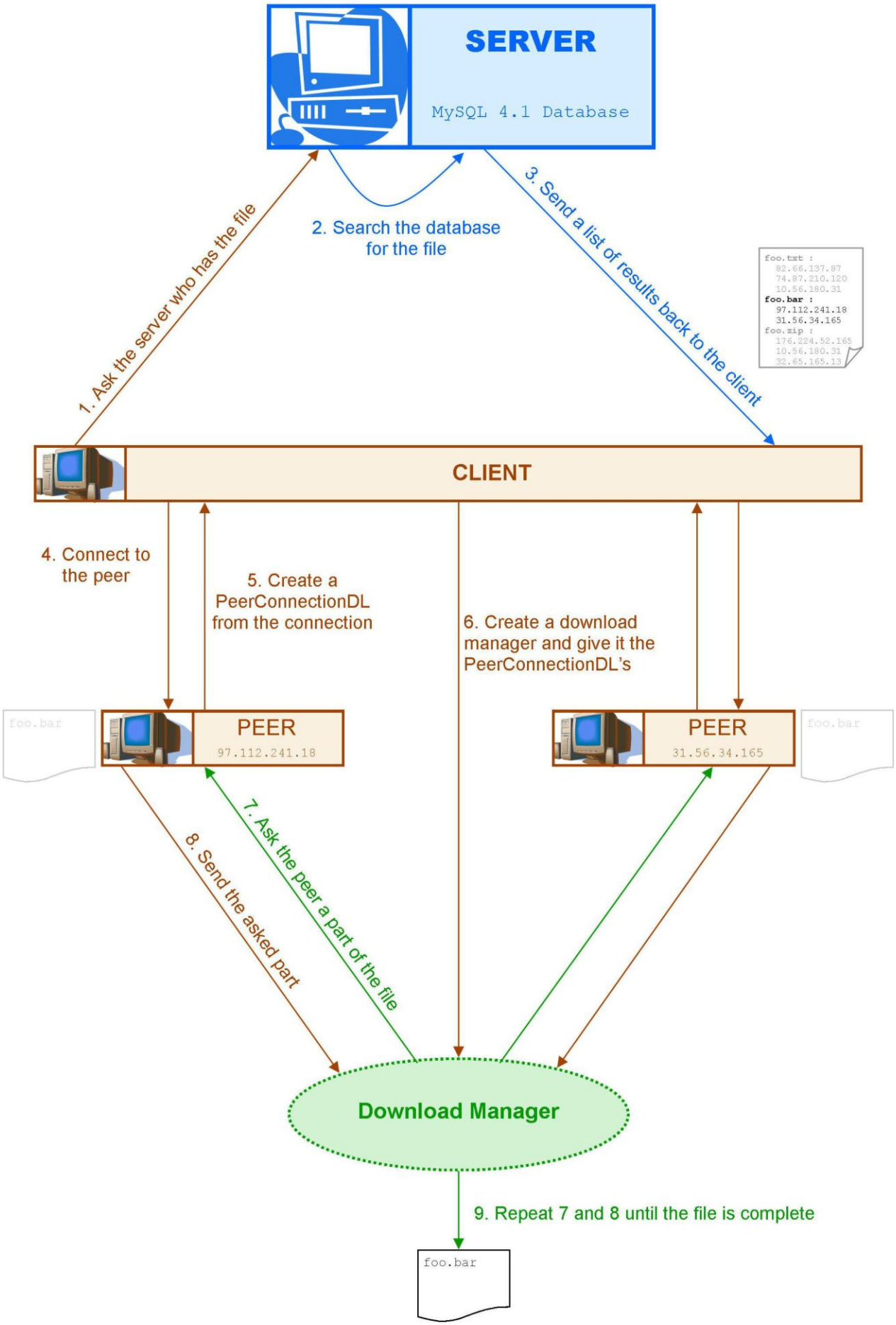
```
create table images
(
     id             integer not null references others(id),
     width          integer,
     height         integer
);
```

Others table creation script

```
create table others
(
     id             integer not null,
     kind           integer,
     filename       text,
     ip             text,
     file_id        text,
     file_size      integer,
     primary key (id)
);
```

APPENDIX E

General downloading process


SERVER
MySQL 4.1 Database
1. Ask the server who has the file
2. Search the database for the file
3. Send a list of results back to the client
foo.txt :
82.66.137.87
74.87.210.120
10.56.180.31
foo.bar :
97.112.241.18
31.56.34.165
foo.zip :
176.224.52.165
10.56.180.31
32.65.165.13
CLIENT
4. Connect to the peer
5. Create a PeerConnectionDL from the connection
6. Create a download manager and give it the PeerConnectionDL's
foo.bar
PEER
97.112.241.18
PEER
31.56.34.165
foo.bar
7. Ask the peer a part of the file
8. Send the asked part
Download Manager
9. Repeat 7 and 8 until the file is complete
foo.bar

APPENDIX F

Saving users' settings

Saving settings on disk

Sometimes, the program stores information about what is going on in a *Properties* object. This is basically a *HashMap* associating to *String* key another *String*, said value. When the application is closed, the contents of this *Properties* object is written into a file on the disk so the program can read them and load them again at next start-up. We will see here what is saved in that file and the format of the *Strings* in there. See below for an example of what you would find in that file:

```
#Fri Jun 10 02:49:31 CEST 2005
download_dir=C\:\\Documents and Settings\\Nicobest\\Bureau
shared_dirs=C\:\\My Music|D\:\\Download
auto_connect=false
server_ip=192.168.0.10\:1337
downloads=113-82-2046-122102-602261264512212-35-106-11|1111110111|
113-82-2046-122102-602261264512212-35-106-11=foo.bar|90|4834120|10
```

The first line is a commentary, telling when the last time the file has been updated was. The second line saved the path of the directory where users want to save their downloaded files. The third line saved users' shared folders' paths. Paths are separated by the "|" character. Fourth line says whether users want to auto-connect at start-up or not. The value here is fairly obvious. Fifth line stores the server's IP address.

The next lines are used to store downloads' progresses, and keep information used to resume them next time the application will be launched. Sixth line saves the actual progress of all the downloads. The value *String* is formatted as follow:

`md5_1 | bitset_1 | md5_2 | bitset_2 | etc.`

Bitset's 1 mean the part has been downloaded while bitset's 0 mean the part is still to download. In the example above, all parts have been downloaded up to the 10^{th} but the 7^{th} is still to be downloaded.

All the other lines are used to resume the download and update the GUI accordingly to the download's state. There is one line per download, thus the key is the md5 hash code of the file being downloaded. For each md5, the value *String* is formatted as follow:

`filename | progress | filesize | nbParts`

where *progress* is the download advancement from 0 to 100 and *nbParts* the number of parts in the file.

APPENDIX G

Technical documentation

Classes overview

AudioTag

AudioTag is used for reading tags from audio files of mp3 and ogg formats. It used an external library for that purpose.

See code listing, page 65

ClientImplementation

This is the main code for the client. This object is instanced when the client connects to the server. Once launched, a loop listens to other clients trying to connect. It also contains all the methods to communicate with the server, such as methods for sharing files or searching for files. Downloads are started from this class too, as well as uploads.

See code listing, page 68

ClientInterface

This is the interface that *ClientImplementation* implements.

See code listing, page 87

DownloadManager

DownloadManager is instanced once per download. It is added *PeerConnectionDL* objects and it manages the download. It also updates the GUI as the download progresses. It writes downloaded parts to files on the disk too.

See code listing, page 89

JPanelLibrary

JPanelLibrary is a sort of *JPanel* and represents the library panel in the GUI. The list used to store paths of shared folders uses a default list model. Internal class *CustomButtonListener* manages actions and calls methods from *ClientImplementation* to communicate with the server.

See code listing, page 98

JPanelSearch

JPanelSearch is a sort of *JPanel* and represents the search panel in the GUI. Each of the four tables uses its own model: *TableAllModel, TableAudioModel, TableVideoModel* and *TableImageModel*. *JPanelSearch* implements *ActionListener* and manages its own actions. These call appropriate methods (mainly *searchByFilename*) in *ClientImplementation*. Also, *JPanelSearch* extracts information from *RequestResults* objects in order to display them into the tables.

See code listing, page 105

JPanelTransfers

JPanelTransfers is a sort of *JPanel* and represents the transfers panel in the GUI. Each of the two tables uses its own model, declared as internal classes *TableUpModel* and *TableDownModel*. The last internal class, *ProgressBarRenderer*, is a custom cell renderer for the download table. *JPanelTransfers* contains methods for stopping, resuming and cancelling downloads, these being called by the instance of *P2PGUI*.

See code listing, page 114

MD5

This class contains static methods for calculating the md5 hash code of a file.

See code listing, page 125

P2P

P2P is the class with the main method. All it actually does is to create an instance of *P2PGUI*.
See code listing, page 127

P2PGUI

This is the main class for the Graphical User Interface. Besides managing the window and the panels that are contained in it, it is also able to save and load user's settings. As far as the GUI is concerned, it also manages the buttons on the south of the panel, calling methods in the appropriate panels (*JPanelLibrary* or *JPanelTransfers*). It has internal classes for listening to events: *MenuListener* and *ButtonListener*, their roles being fairly obvious.
See code listing, page 128

PeerConnectionDL

This class is instanced by the *download* method of *ClientImplementation* when connecting to other peers. Basically, *PeerConnectionDL* represents that connection. It contains a method for downloading a part of a file from the client it is connected to. Instances of this class are added to instances of *DownloadManager*.
See code listing, page 143

PreferencesWindow

PreferencesWindow is the window showing user's settings such as server's address. It is a sort of *JDialog* and it implements *ActionListener* to handle actions by itself.
See code listing, page 146

RequestResult

This is the object returned by the server to the client after that latter searched for a file. This object thus contains information about all files, such as their md5 or their size. It only contains information common to all the files.

See code listing, page 150

RequestResultAudio

It extends *RequestResult* and add to the list of information stored the additional information about audio files.

See code listing, page 153

RequestResultImages

It extends *RequestResult* and add to the list of information stored the additional information about image files.

See code listing, page 156

RequestResultVideo

It extends *RequestResult* and add to the list of information stored the additional information about video files.

See code listing, page 158

ServerImplementation

This is the whole code for the server. Thus it contains everything you would expect from it, from the loop listening to the clients to the methods sending requests to the database.

See code listing, page 161

ServerInterface

This is the interface implemented by *ServerImplementation*.

See code listing, page 175

TableAllModel

This is the model used by one of the *JTable* of *JPanelSearch*. It extends *MyTableModel* and only redefines the column names and their kind.

See code listing, page 176

TableAudioModel

This is the model used by one of the *JTable* of *JPanelSearch*. It extends *MyTableModel* and only redefines the column names and their kind.

See code listing, page 177

TableImageModel

This is the model used by one of the *JTable* of *JPanelSearch*. It extends *MyTableModel* and only redefines the column names and their kind.

See code listing, page 178

TableVideoModel

This is the model used by one of the *JTable* of *JPanelSearch*. It extends *MyTableModel* and only redefines the column names and their kind.

See code listing, page 179

MyTableModel

It is the model of table used by tables of *JPanelSearch*. It extends *AbstractTableModel* and redefines the methods quite normally. It adds a method for setting the values of the whole table.

See code listing, page 180

References

Websites

About peer-to-peer in general:

Freepedia: http://fr.freepedia.org/index.php?title=P2P
Wikipedia: http://wikipedia.org
WinMxHelp: http://winmx.2038.net/winmx/fr-wpn.html
NTRG: http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/Intro.html
LimeWire: http://www.limewire.org
JXTA: http://www.jxta.org
Gnutella: http://www.gnucleus.com/gwebcache

About development in JAVA:

Sourceforge: http://sourceforge.net
Javazoom: http://www.javazoom.net
IBM: http://www.alphaworks.ibm.com/tech/tk4mpeg4
Developpez.com: http://www.developpez.com
MySQL: http://dev.mysql.com

PowerPoint Presentations

Concept-Based P2P Search by Ingmar Weber, Max-Planck-Institute for Computer Science:
http://www.mpi-sb.mpg.de/~iweber/peer-to-peer/concept-based%20P2P%20Search.ppt
Protocoles de communication en Peer-to-Peer by Marc CALVISI, M2PGI :
http://www-adele.imag.fr/~donsez/ujf/easrr0405/p2p/p2p.ppt