



CURS 2021-2022

DUCHNET – A PEER-TO-PEER FILE
SHARING SYSTEM IMPLEMENTED
USING JAVA RMI
DISTRIBUTED COMPUTING – PROJECT01

JOEL AUMEDES SERRANO I PAU ESCOLÀ BARRAGÁN
UNIVERSITAT DE LLEIDA



Index

Introduction	2
How does it work?	2
Startup	3
Contents and metadata	4
Request information from the whole network	4
Downloading a file	5
Use case example	9
Conclusion	12
Appendix	13
Appendix 1: GlobalQueueThreads	13
Appendix 2: ContentManager cache	13

Introduction

The Duchnet is a peer-to-peer file sharing network implemented using Java RMI. By the definition of peer-to-peer, all nodes must provide client and server functionality, so all nodes execute the same code, implemented in the peer package. Interfaces Peer and Manager are used to define which functions can be used locally and remotely, and which ones can only be used locally.

A node consists of a Peer (implemented in PeerImp), a Manager (implemented in ContentManager) and an RMI registry, where both objects are binded under the names “peer” and “manager”, so that we can refer to a node by the IP of the machine running it and the port where the registry is running. A combination of a Peer and its Manager is referred to as a node’s components, and a combination of an IP and a port running a Registry is referred to as a PeerInfo. More information can be seen in the project’s Javadoc documentation, and all the code can be found in its GitHub repository:

<https://github.com/Galahad3x/Duchnet>

How does it work?

The Duchnet CLI has many functions that the user can execute. By executing the “help” command, a message printing all the commands and a short description of what they do will be displayed. The more important commands will be explained with more detail in the following sections. To see in more detail what is happening behind the curtain, the “debug” command can be used to toggle between INFO and WARNING debug levels. WARNING is the default and recommended option.

Startup

To start a node, PeerProgram must be executed with the information of a node in the network, or without if we want to create a new network.

Arguments: [Local Registry Port] [Remote IP] [Remote Port]

If a port is not specified, the default 1099 will be used.

Creating a new network will create an isolated node. Connecting to another node will add the components of each node to the other to maintain a strongly connected connection graph, as seen in Image 1. If this is not done, the node that started initially will not be able to see the second node in the network, while the second one will see the original.

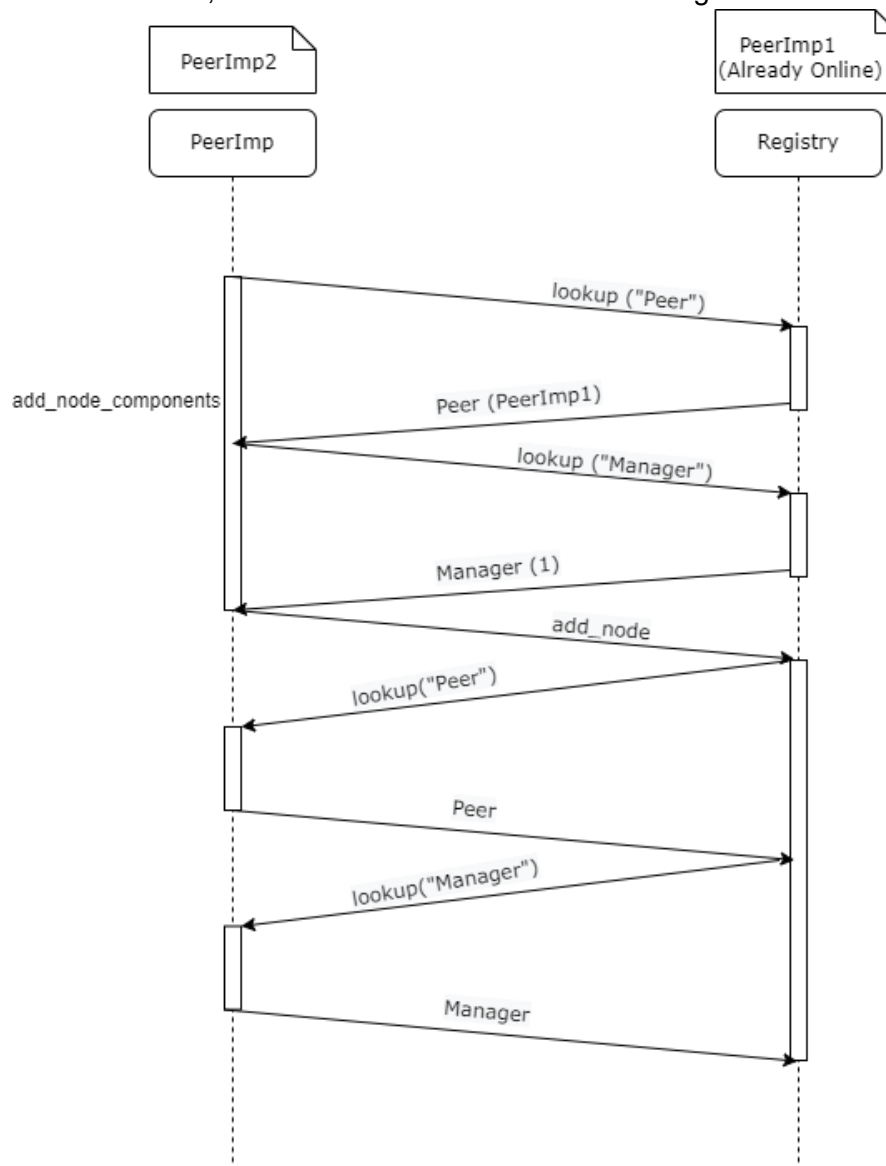


Image 1 - Starting up a node

Before any connection is done though, the user must specify the folder route of the contents, the maximum allowed number of parallel download threads, upload threads, and file threads. How these threads are used will be explained in more detail in the Download a File section.

Contents and metadata

Everything related to a content file is encapsulated in a Content object, which uses the CRC32 hash function implemented in HashCalculator to differentiate between contents. A Content object saves the different filenames used to refer to the same file across the network, the descriptions and the tags associated with this content. Using the XMLDatabase class, the descriptions and the tags associated with a hash are saved and restored in an xml file between sessions. The user can add descriptions and tags to a content with the “modify” command. These descriptions and tags will be added locally, and then shared with the other nodes when they request it.

Request information from the whole network

When a node needs some information from the whole network, it requests it from all the Peers it knows, which in turn request it to all the Peers they know, until all Peers in the network have responded to the request. To avoid sending requests in a loop, these functions use a visited peers list with all the PeerInfos of the Peers that have already answered this request.

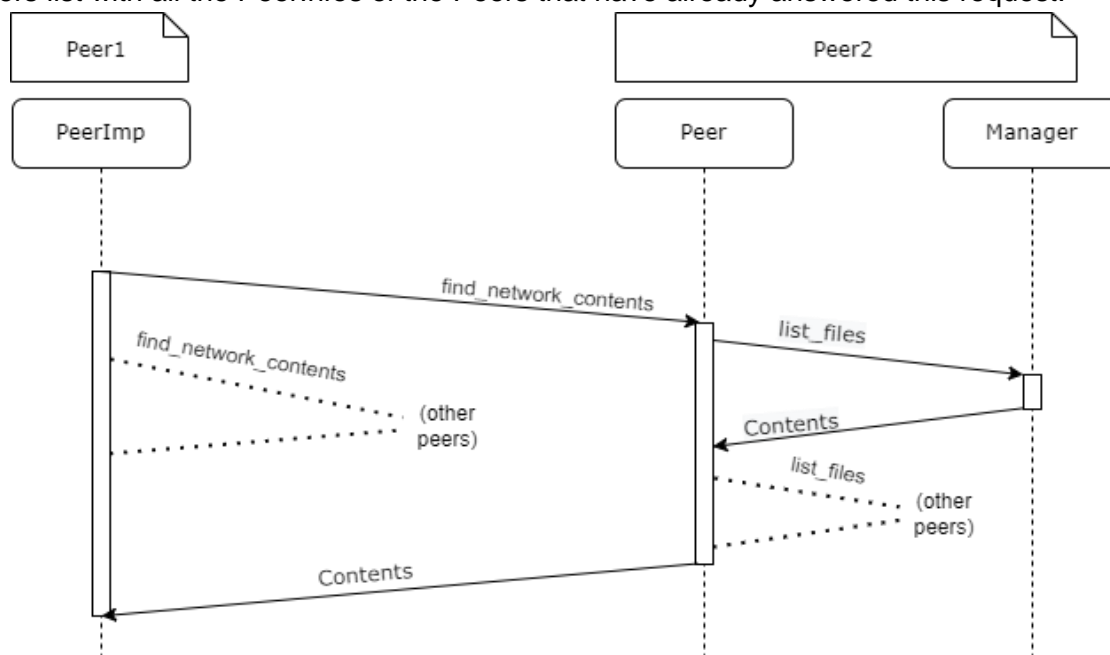


Image 2 - List all files in the network

This process is used mainly when requesting all files in the network (which can be done with the “list all” command), searching for a file to download (using the “download” command) and searching for seeders¹ for a file.

¹ A seeder is a node that owns a file we want, and that we can request it from

Downloading a file

To download a file, the user must use the “download” command. This will start a download process, which will guide the user through all the steps. When searching for content to download, the user can select a filter for file name, description, or tag. Searching by file name is the recommended method, since it saves the user some time by skipping the hash calculation of large files that don’t meet the criteria.

Image 3 is the sequence diagram for the first version that allowed the Duchnet to transfer a file.

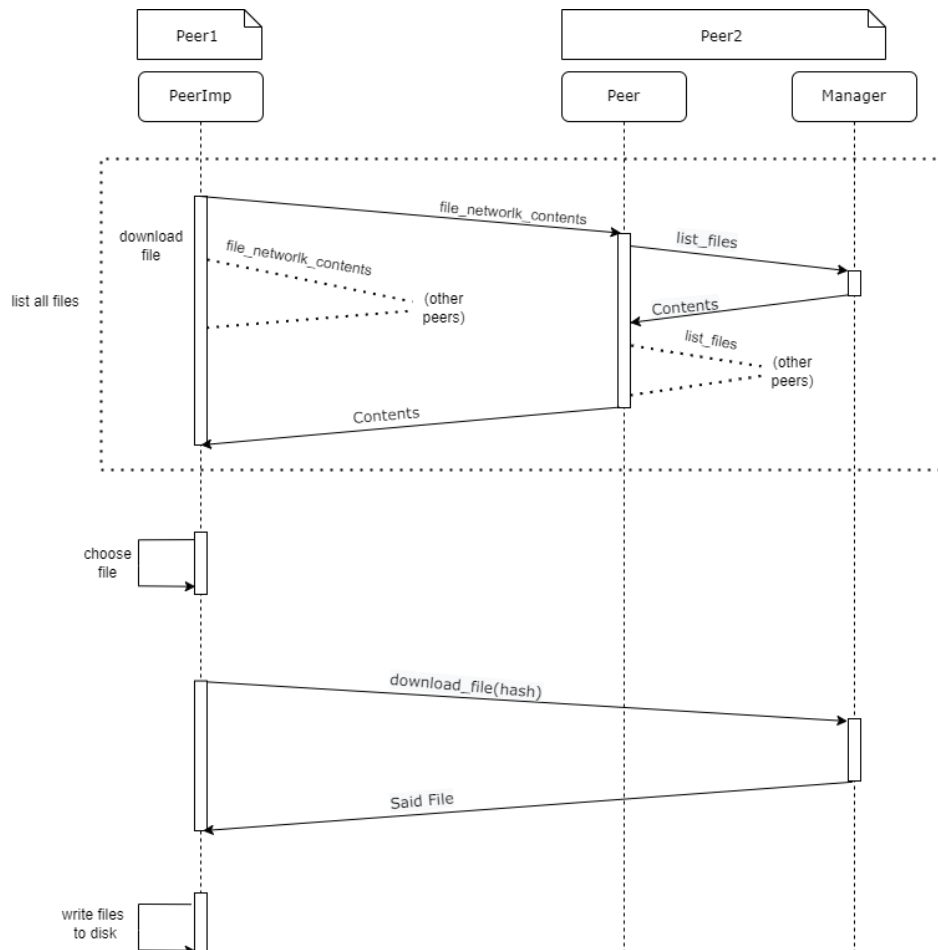


Image 3 - First version of the download process

It uses the request information from the whole network flow twice, when searching for the network contents and when searching for seeders of this file. It has 5 distinct steps; List all files in the network, choose which one to download, find a seeder for this file, ask for the whole file to a single seeder, and write the whole file to the disk.

This version was upgraded by splitting the downloads into 1 MB chunks, as the diagram for the second version, Image 4, shows.

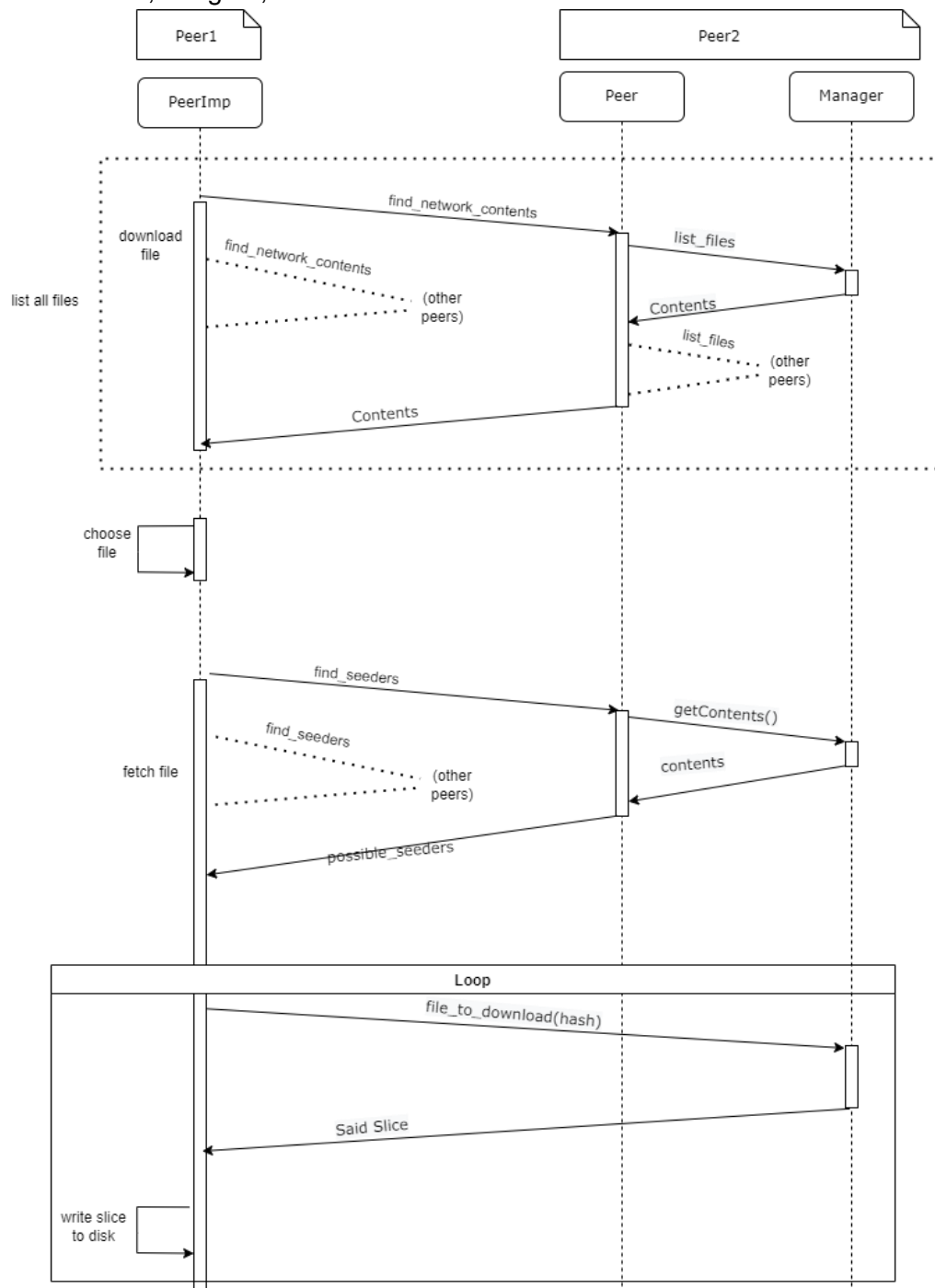


Image 4 - Second version of the download process

It has the same steps as the first version and is still sequential. The main difference is that we don't request the whole file at once, we instead ask for 1 MB chunks, which we write to the disk before asking for another chunk.

The final version has more changes related to the previous ones.

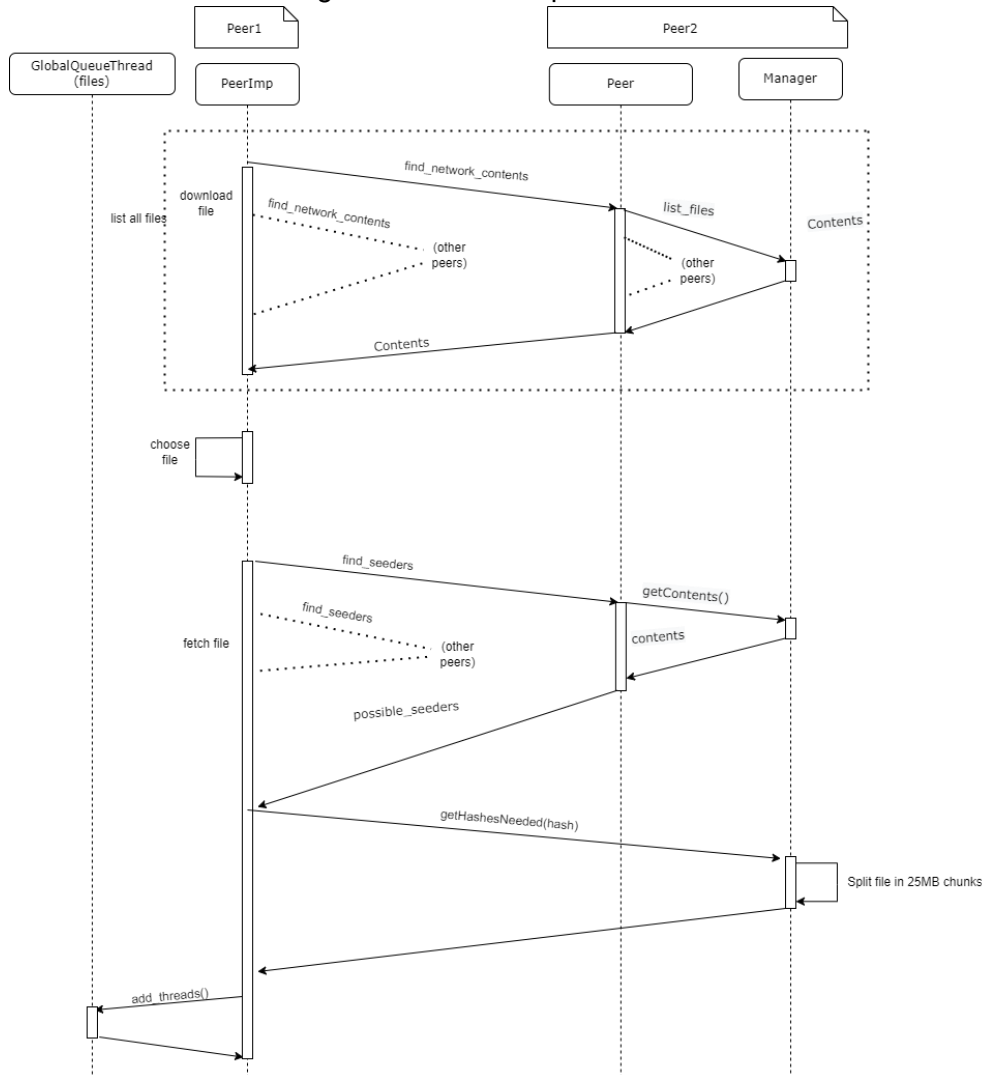


Image 5 - Final version of the download process

This version implements parallelism which allows the user to download huge files, with more than one file at once and more than one download thread at all. The process is very similar to the user, however, in this version, once the user has selected which file to download, it can go back to the main menu in the CLI and execute other commands instead of being blocked.

This version splits bigger files into 25 MB slices, and these slices into 1 MB chunks. They are downloaded separately but joined together once all of them have finished downloading. This is all done transparently to the user, which must only wait.

To handle concurrency, two threads handle a queue each, one for files and one for downloads. These threads encapsulate the starting and joining of threads in a thread-safe way, by waiting on a synchronized block and being notified when a change has been done. Then, to not flood the system with threads, the number of allowed active threads at once is limited by the user. The number of download threads sets how many threads can come out of this node to another, asking for a 1 MB slice of data. The number of upload threads sets how many incoming threads can get data from our files at once. The number of file threads sets how many 25 MB slices can be downloaded (creating download threads) at once.

The best way to explain how the download works is using an example diagram of what happens once the user has started a download, Image 6.

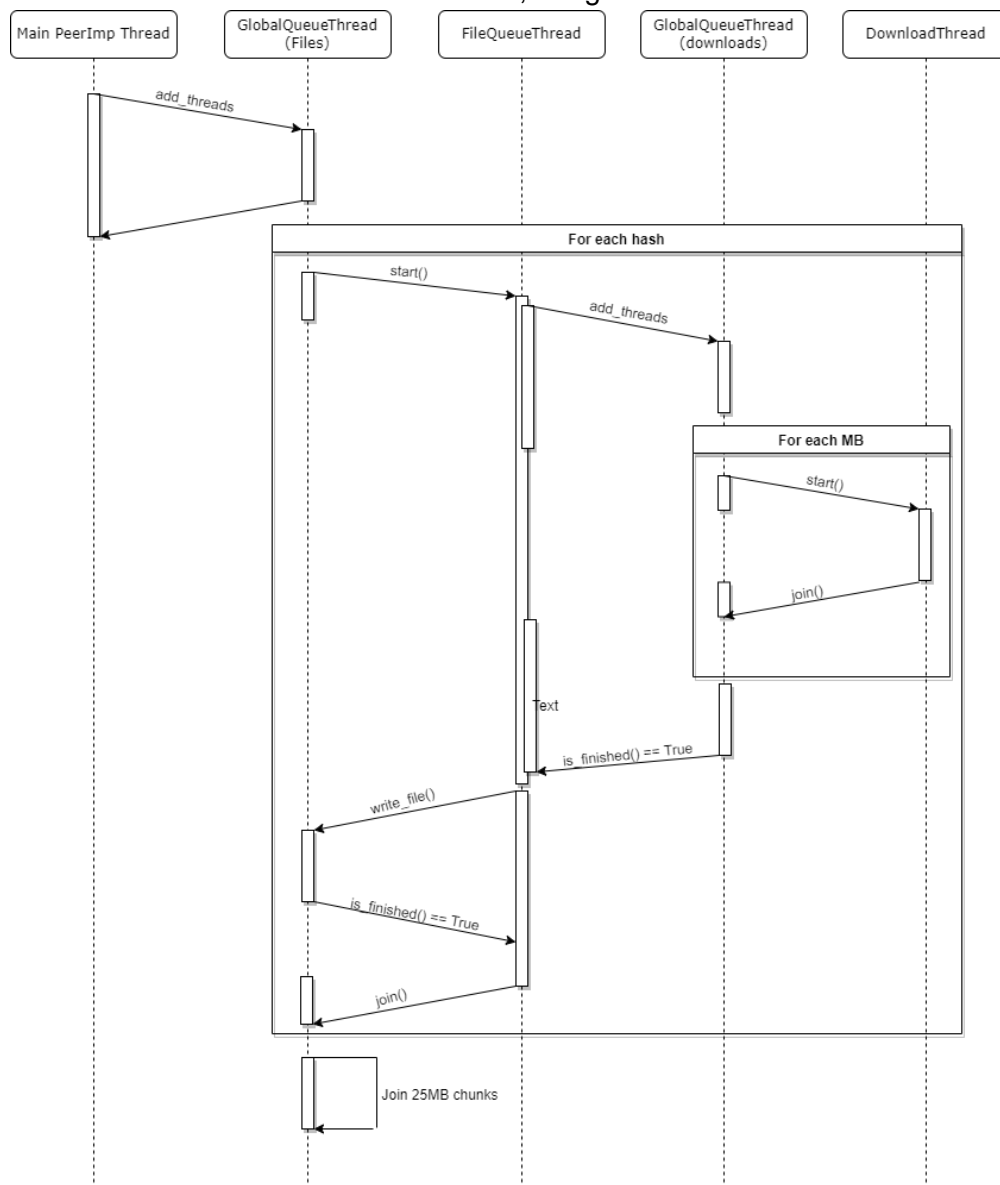


Image 6 - Diagram of the download queues

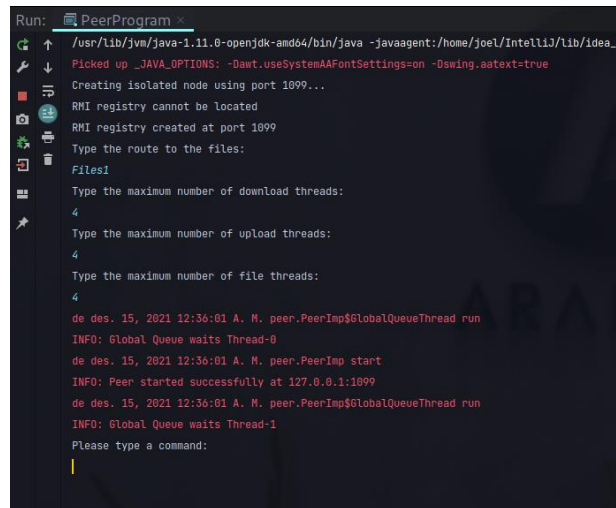
In Image 6, it is not shown how the GlobalFileThread threads work. This is explained in more detail in [Appendix 1](#).

The process starts when the main thread where all the download process has taken place adds all the necessary FileQueueThreads to the GlobalQueueThread that handles files. Remember that we split bigger files into 25 MB chunks that each have a different hash, so we add all of them.

Once a FileQueueThread is active, it adds all its necessary DownloadThreads to the GlobalQueueThread that handles downloads. Then, when possible, the GlobalQueueThread that handles downloads starts them, and they connect to a Manager to download their chunk of the file. They download it and save it on their FileQueueThread. Once a FileQueueThread has downloaded all its chunks, it writes them on a file. If it's the last slice that's part of a bigger file, it joins all slices together to rebuild the original file and deletes the remaining slices.

Use case example

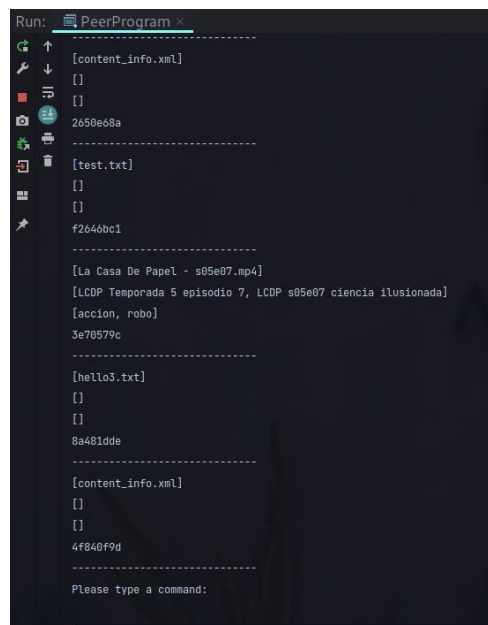
The first thing to do is start an isolated node as shown in Image 7:



```
Run: PeerProgram x
/usr/lib/jvm/java-11.0-openjdk-amd64/bin/java -javaagent:/home/joel/IntelliJ/lib/idea_r
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Creating isolated node using port 1099...
RMI registry cannot be located
RMI registry created at port 1099
Type the route to the files:
Files1
Type the maximum number of download threads:
4
Type the maximum number of upload threads:
4
Type the maximum number of file threads:
4
de des. 15, 2021 12:36:01 A. M. peer.PeerImp$GlobalQueueThread run
INFO: Global Queue waits Thread-0
de des. 15, 2021 12:36:01 A. M. peer.PeerImp start
INFO: Peer started successfully at 127.0.0.1:1099
de des. 15, 2021 12:36:01 A. M. peer.PeerImp$GlobalQueueThread run
INFO: Global Queue waits Thread-1
Please type a command:
```

Image 7 - Starting an isolated node

Now, this node can run commands locally, for example the “list” command that will list all contents in the specified folder.

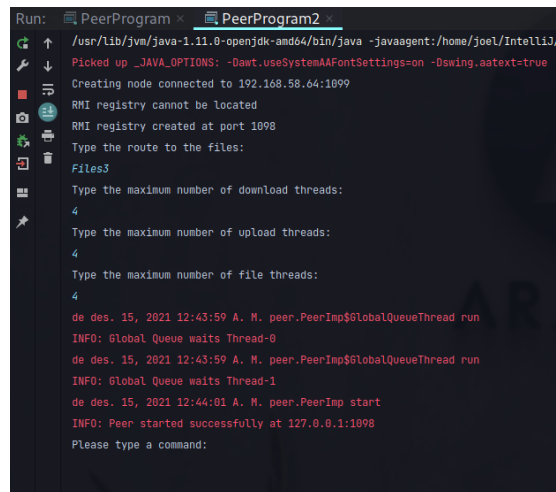


```
Run: PeerProgram x
-----
[content_info.xml]
[]
[]
2650e08a
-----
[test.txt]
[]
[]
f2646bc1
-----
[La Casa De Papel - s05e07.mp4]
[LCDP Temporada 5 episodio 7, LCDP s05e07 ciencia ilusionada]
[accion, robo]
3e70579c
-----
[hello3.txt]
[]
[]
8a481dde
-----
[content_info.xml]
[]
[]
4f840f9d
-----
Please type a command:
```

Image 8 - Listing content in an isolated node

As seen in Image 8, some contents have set descriptions and tags in the XML file, so they are shown here.

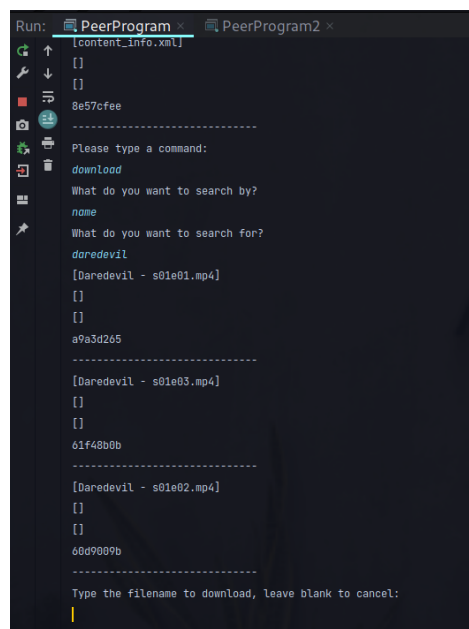
Now we can start a 2nd node connected to this original one:



```
Run: PeerProgram x PeerProgram2 x
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java -javaagent:/home/joel/IntelliJ/...
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Creating node connected to 192.168.58.64:1099
RMI registry cannot be located
RMI registry created at port 1098
Type the route to the files:
Files3
Type the maximum number of download threads:
4
Type the maximum number of upload threads:
4
Type the maximum number of file threads:
4
de des. 15, 2021 12:43:59 A. M. peer.PeerImp$GlobalQueueThread run
INFO: Global Queue waits Thread-0
de des. 15, 2021 12:43:59 A. M. peer.PeerImp$GlobalQueueThread run
INFO: Global Queue waits Thread-1
de des. 15, 2021 12:44:01 A. M. peer.PeerImp start
INFO: Peer started successfully at 127.0.0.1:1098
Please type a command:
```

Image 9 - Starting a node connected to the original one

We start the node connected to the original one by passing it as a parameter, and in this case, we run the second node in the same machine but on port 1098 instead of 1099, and with a different folder. Now, we can use network functions from any of the nodes. For example, we want to download a chapter of a TV series. We would start the process by calling the “download” command and searching by the name of the series, such as in Image 10.

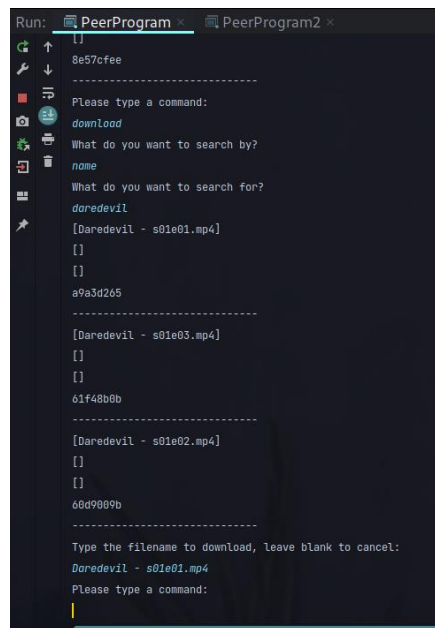


```
Run: PeerProgram x PeerProgram2 x
[content_info.xml]
[]
[]
8e57cfee
-----
Please type a command:
download
What do you want to search by?
name
What do you want to search for?
daredevil
[Daredevil - s01e01.mp4]
[]
[]
a9a3d2d5
-----
[Daredevil - s01e03.mp4]
[]
[]
61f48b0b
-----
[Daredevil - s01e02.mp4]
[]
[]
60d9009b
-----
Type the filename to download, leave blank to cancel:
```

Image 10 - Searching for "daredevil" in the network

As it has been said, the most efficient way is by name filtering. In this case, we ask the network for all files with “daredevil” in their name. This search is not case sensitive.

Since we are interested in only the first episode, we type the name of it and the process starts in the background.

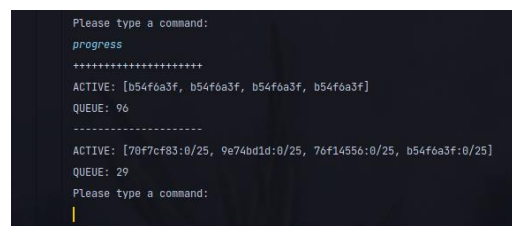


```
Run: PeerProgram PeerProgram2
[]
8e57cfee
-----
Please type a command:
download
What do you want to search by?
name
What do you want to search for?
daredevil
[Daredevil - s01e01.mp4]
[]
[]
a9a3d2d5
-----
[Daredevil - s01e03.mp4]
[]
[]
o1f48b0b
-----
[Daredevil - s01e02.mp4]
[]
[]
o6d9009b
-----
Type the filename to download, leave blank to cancel:
Daredevil - s01e01.mp4
Please type a command:

```

Image 11 - Starting a download

Now the file is downloading in the background. We can use the command “progress” to see which files and downloads are currently active, and how many are in each queue, as seen in Image 12.



```
Please type a command:
progress
+-----+
ACTIVE: [b54f6a3f, b54f6a3f, b54f6a3f, b54f6a3f]
QUEUE: 96
-----
ACTIVE: [70f7cf83:0/25, 9e74bd1d:0/25, 76f14556:0/25, b54f6a3f:0/25]
QUEUE: 29
Please type a command:

```

Image 12 - Checking the progress of the downloads

Here we can see that there are currently 96 threads in the queue (not active threads, threads waiting to get started) and 29 files in the queue, because 4 files and 4 threads are already active, corresponding to the size of the TV series chapter size of roughly 850 MB. This file will take roughly 20 minutes to download and rebuild completely.

Conclusion

Developing Duchnet was challenging, and even though we started to count when most of it was already developed it still took us more than 13 hours to code. A rough estimate of the total hours would be between 50 and 70 hours. The harder parts were defining the overall structure of the network and of a peer, since it needed to be efficient and well designed, and implementing concurrent downloads, to avoid deadlocks and to make the parallelism of downloads as transparent to the user as possible, which was finally achieved.

Appendix

Appendix 1: GlobalQueueThreads

GlobalQueueThreads are threads that have a single responsibility; handle a queue of threads. This queue is formed by MyThread objects, which is an abstract class that extends Thread, and that will be extended by FileQueueThread and by DownloadThread. This class provides extra functionalities to the thread and is used to not implement two different GlobalQueueThreads that do the exact same thing, since the file thread and the download thread work the same way, explained below.

To handle the concurrency on the queue, the GlobalQueueThread locks the queue access inside a loop inside a synchronized block, to never release it fully. Then, the GlobalQueueThread waits on this synchronized block, so the queue is free for everybody to access. Access to the queue is only allowed through functions that have a block inside, so only one thread at a time can add threads to the queue. Once the threads are added to the queue, the GlobalQueueThread is notified, which makes it regain access to the critical zone immediately instead of giving it to another random thread. The GlobalQueueThread then checks if it's possible to activate this thread, either starting it on an empty spot or by joining a thread that has already finished. Once the maximum number of threads have been started, the GlobalQueueThread goes back to sleep using the wait function. The GlobalQueueThread can also be notified by a thread that has finished, instead of only being notified when adding a new thread to the queue.

Appendix 2: ContentManager cache

In order to speed up downloads, a ContentManager uses a cache of 4 slices of 25 MB maximum each, by saving the bytes of the more recently downloaded files.

The cache is useful because one of the most time-consuming operations in a download was reading a 25 MB file from the disk just to get a 1 MB slice off of it. The cache works thanks to the principle of proximity in time by which real cache memories are based; if a memory address has been accessed recently it is likely to be accessed again soon. Since most of the 25 download threads for a slice of a file happen roughly one after the other, the cache eliminates roughly 20 times where the whole file was read from the disk, which shortened the download time drastically.