

**Distributed Systems: Lab 1**

Introductory lab & TCP/UDP communications

Max Cornilly

Group 3: re-ordering

Industrial Engineering: Electronics - ICT

Third Bachelor

Academic Year 2021 - 2022

Table of contents

[Introductory lab & TCP/UDP communications 1](https://uantwerpen-my.sharepoint.com/personal/s0190345_ad_ua_ac_be/Documents/UA/6e%20Semester/6-Distributed%20Systems/Lab1.docx#_Toc98789678)

[Table of contents 2](#_Toc98789679)

[1 Git 3](#_Toc98789680)

[1.1 Create local repository 3](#_Toc98789681)

[1.2 Create a public repository on GitHub 3](#_Toc98789682)

[1.3 Push code to public repository 4](#_Toc98789683)

[1.4 Roll back to previous commits 5](#_Toc98789684)

[1.5 Induce conflicts on the remote repository and merge the conflicts 5](#_Toc98789685)

[2 TCP/UDP comms: 6](#_Toc98789686)

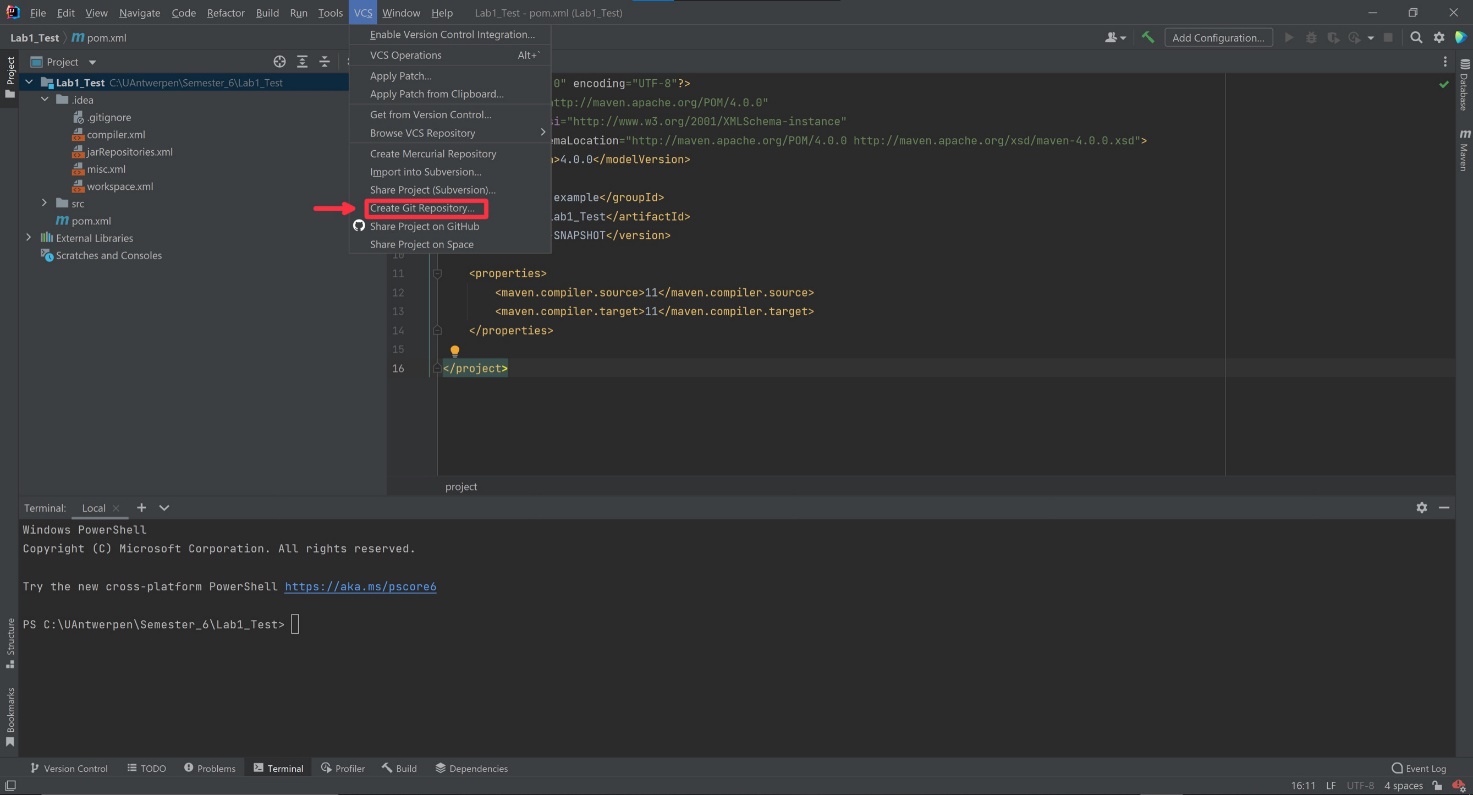
[2.1 Filesharing Application using TCP 6](#_Toc98789687)

[2.2 Filesharing Application using UDP 6](#_Toc98789688)

[2.3 Enabling Multithreading 6](#_Toc98789689)

[2.4 Conclusions 8](#_Toc98789690)

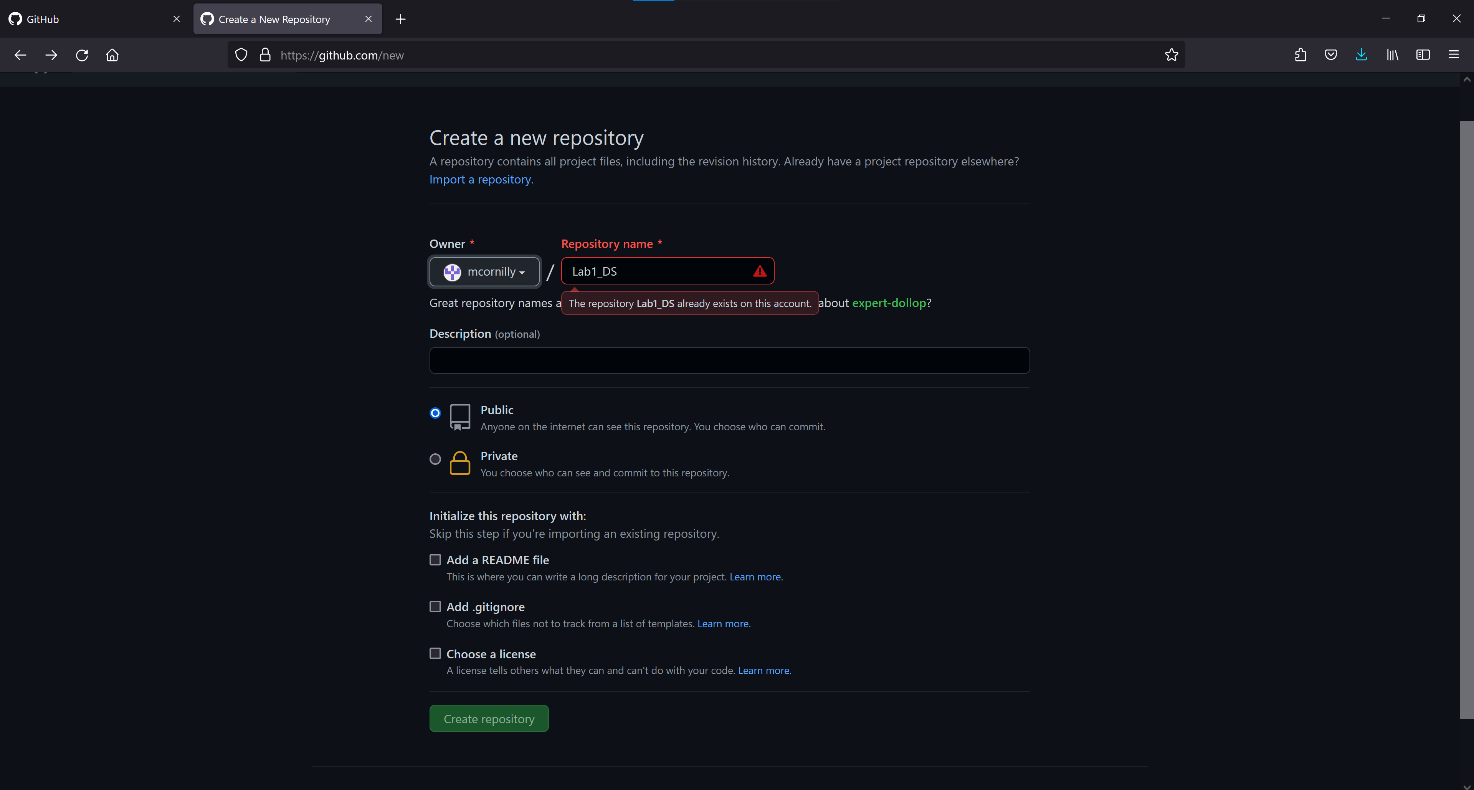
1. Git
   1. Create local repository



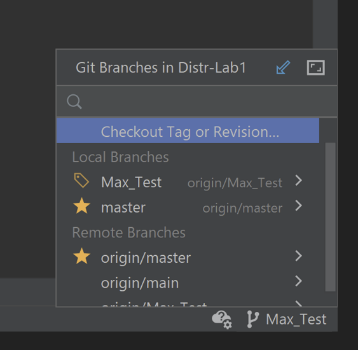
*Figure 1.1: Create Git repository*

To create a local repository in IntelliJ, click on VCS then create Git repository. This is the repository that is stored locally on my computer.

* 1. Create a public repository on GitHub



*Figure 1.2: Create public Git repository*

We created a public repository so that all the group members have access to our code.

To test our repository, we created different branches and switched between these branches. Afterwards we pushed our code to the public repository so that we can pull the code of our teammates. Then we tested the rollback feature to get code from a previous commit.

* 1. Push code to public repository

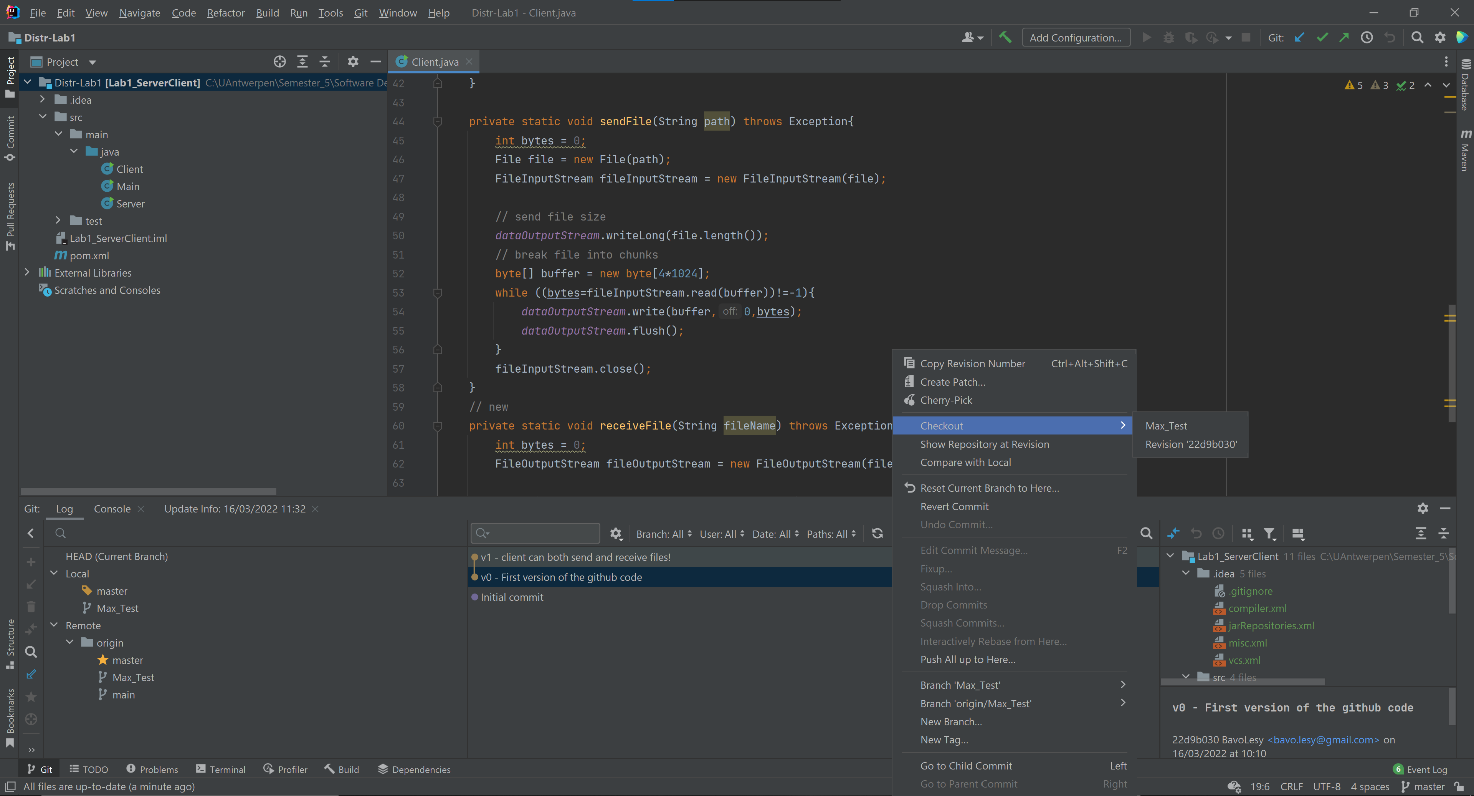
*Figure 1.3: Branches*

Afbeelding met tekst, schermafbeelding, monitor

Automatisch gegenereerde beschrijving

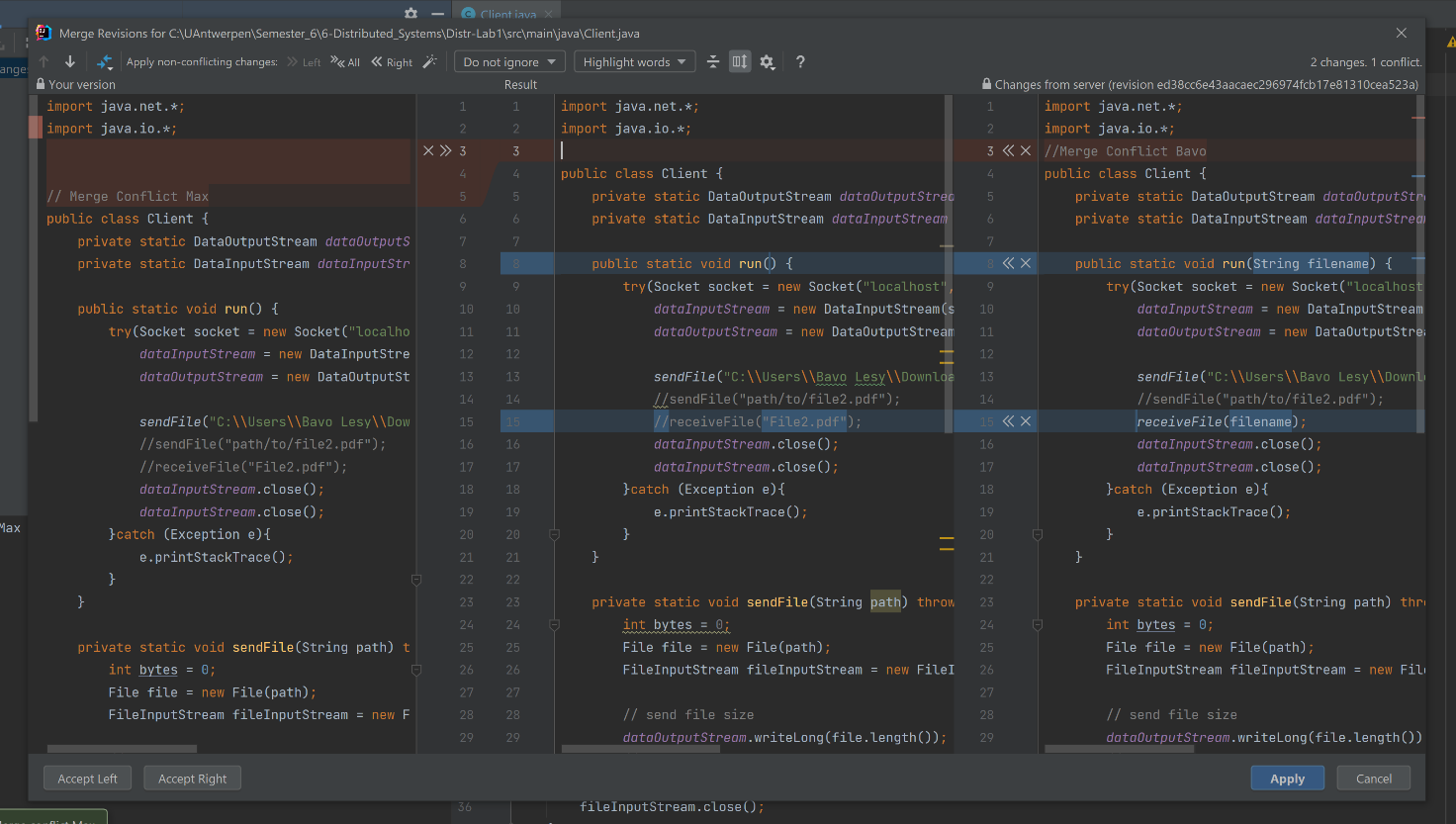
*Figure 1.4: Push code*

To push your code to the public repository you just press the push button in IntelliJ and confirm. Now the most recent version of your code is pushed on the repository.

* 1. Roll back to previous commits

*Figure 1.4: Push code*

If you made a mistake in the commit you did there is a way to roll back to previous commits. As you can see in figure 4 you can go back to a different version of your code. Each commit has a number and the possibility to add a comment to your commit.

* 1. Induce conflicts on the remote repository and merge the conflicts

*Figure 1.5: Merging conflicts*

We both worked on the same piece of code but did some things differently. If we both commit our code, we induce a conflict. If there is a conflict, we can choose which version of the code we can accept and which one to discard.

1. TCP/UDP comms:
   1. Filesharing Application using TCP

Firstly, we developed a filesharing application that uses TCP communication to transfer files between the server and the client. We used the java.net framework to implement the application. With this framework we can use java.net.ServerSocket to open a TCP socket for the server and we can use java.net.Socket to open a TCP socket for the client and to establish a connection.

Once we’ve established a connection, it is time to start our filesharing. The unique part about our application is that clients can both send files to and receive files from the server. To implement the filesharing, we used the java.io library. Thanks to this library, we can use the DataInputStream, DataOutputStream, File, FileInputStream and FileOutputstream. We start of by creating a DataInputStream and DataOutputstream on both the Server and Client side so they can send and receive data.

For sending a file, we read it from our local directory and get the FileInputStream. Then we write() the contents of the file into the DataOutputStream After that we use the flush() method to clear the stream.. On the receiving end, we get the FileOutputStream and write() the contents from the DataInputStream into it.

If the interaction is done we close() the datastreams and we close() the socket.

* 1. Filesharing Application using UDP

UDP is quite different from TCP, the biggest difference is that TCP is connection-oriented protocol. This means that before there can be data transmitted there must be a connection between the two entities. TCP guarantees that the data will be delivered in the correct order without errors. This makes TCP a reliable protocol. But it can take more time in comparison with UDP. UDP is a connection less protocol which makes it a faster protocol but less reliable. In UDP data is continuously sent to the recipient, whether or not they receive it.

Since we don’t need to establish a connection, we just send our data through the socket and assume there is someone listening on the other end. To implement UDP connection, we firstly open a DatagramSocket (java.net framework) and then we use DatagramPacket to send / receive packets through this socket.

For sending a file, we read it from our local directory and open the FileInputStream. Then we write the contents of the file into a buffer, and we use this buffer to create a DatagramPacket. Then we use DatagramSocket.send(DatagramPacket) to send the packet. On the receiving end, we create an empty DatagramPacket and use DatagramSocket.receive(DatagramPacket) to get the packet. Then we use getData() to read the contents and write them into a buffer and getLength() to get the size of the packet. Lastly, we use FileOutputStream.write(buffer, 0, size) to write the contents into a file.

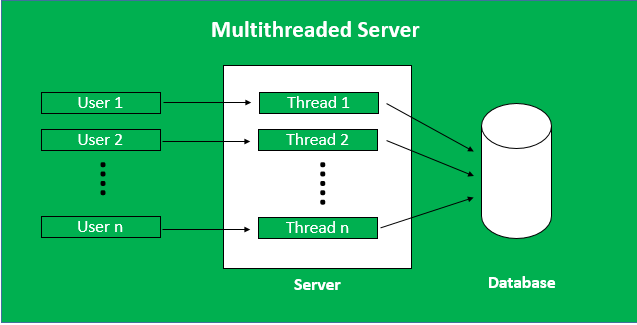
* 1. Enabling Multithreading

Multithreading means that our server can handle multiple clients at the same time through the same socket, by using different threads. When a client sends a request to send or receive a file, a thread is generated through which the client can communicate with the server.

Multithreading seemed daunting at first, but in reality it is not. We just moved the filesharing duties of the server into a ServerThread Class. The TCP server sets up a connection with the client and then starts a new ServerThread object. This object is responsible for filesharing and handling the client. So if two clients connect to the same socket they each instantiate a different ServerThread that can work in parallel on the same socket.

For the UDP server it is mostly the same, except for the fact that there is no connection to start. The server is constantly listening to the socket and also makes a different UDPServerThread object for every client that sends a request.

Multithreaded servers have a lot of advantages. To start off they respond efficiently and quickly to the increasing client queries quickly. If u use a multithreaded server, the waiting time for the user decreases. All users can get a single response at a single time, so no user has to wait for other processes to finish. All the threads are independent of each other. Therefore, an issue in one thread doesn’t affect other threads.



*Figure 2.1: Multithreaded server*

*(source:https://www.geeksforgeeks.org/multithreaded-servers-in-java/)*

MultiThreaded Server Programming

*Figure 2.2: Multithreaded server*

*(source:https://www.geeksforgeeks.org/multithreaded-servers-in-java/)*

* 1. Conclusions

We are very happy with how our filesharing server turned out. Using TCP communication, we can send and receive large files like pdf’s and videos. Our UDP application is limited to sending and receiving small files that don’t exceed the max packet size. This is because we did not implement a way to divide the files into multiple packets. With UDP you don’t know which packets get received first or if some packets get lost, and this can lead to corrupt files.

* 1. GitHub Link

https://github.com/BavoLesy/Distr-Lab1