

**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)

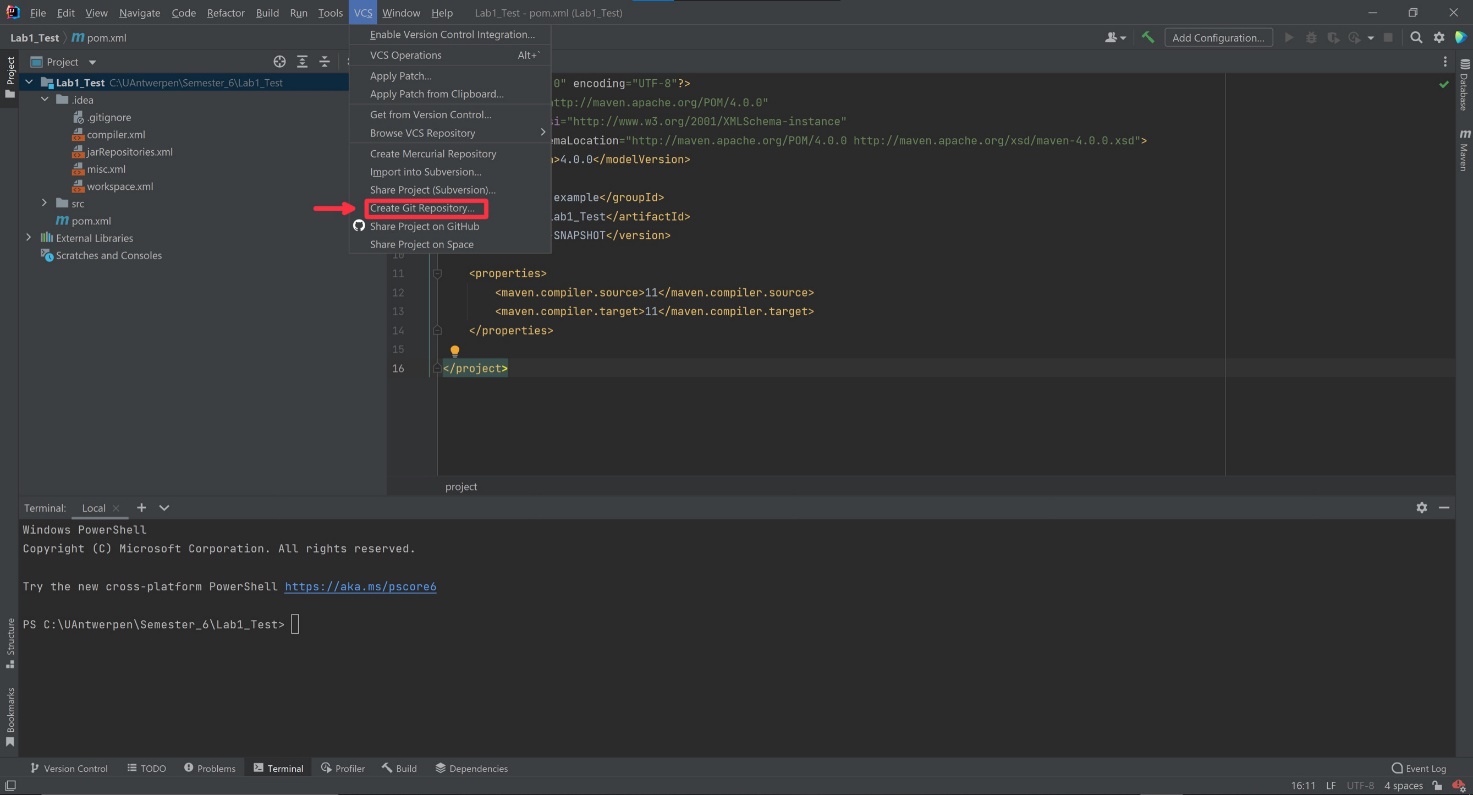
[2.1 Filesharing Application using TCP 6](#_Toc98789687)

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

[2.3 Enabling Multithreading 6](#_Toc98789689)

[2.4 Conclusions 8](#_Toc98789690)

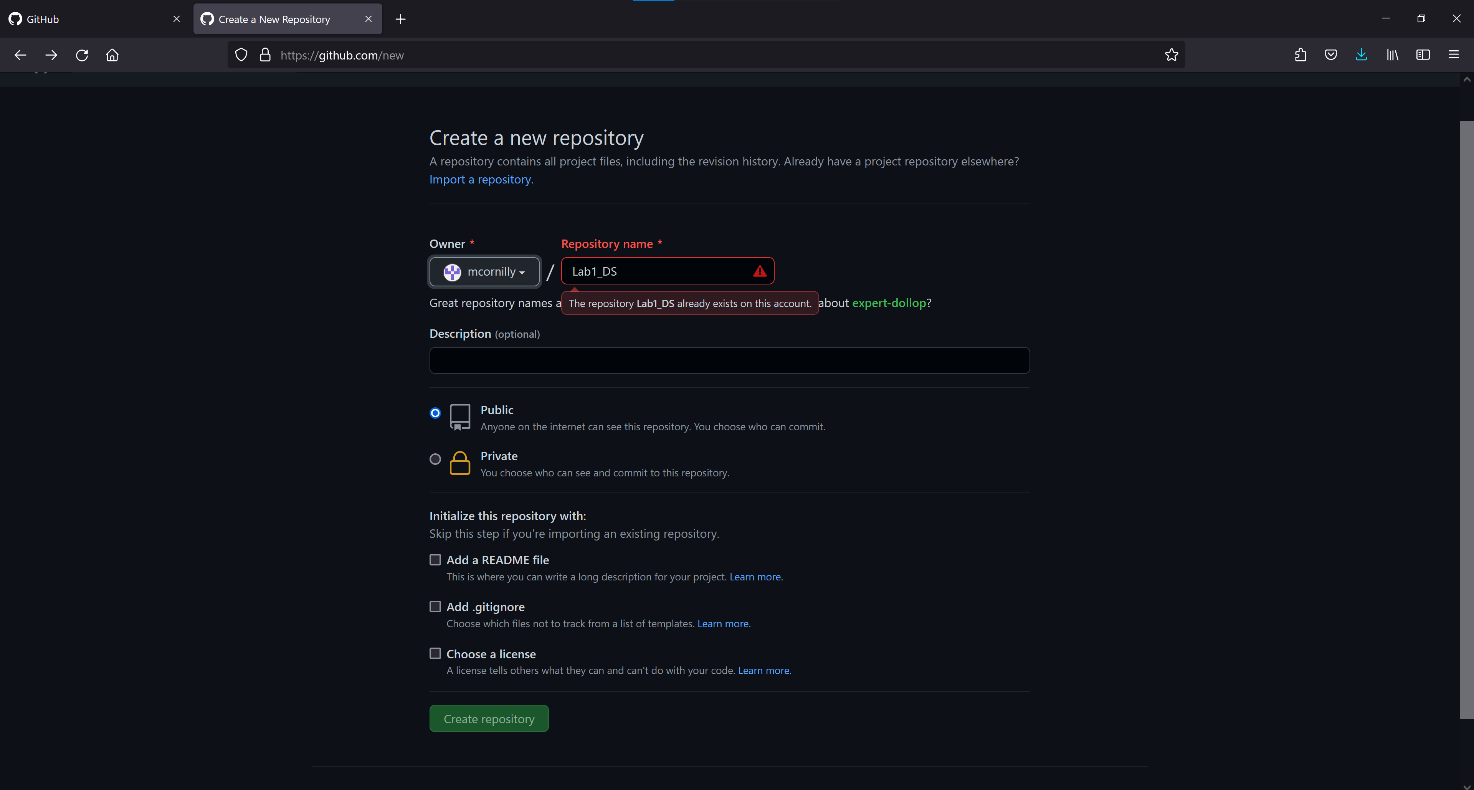
1. Git
   1. Create local repository



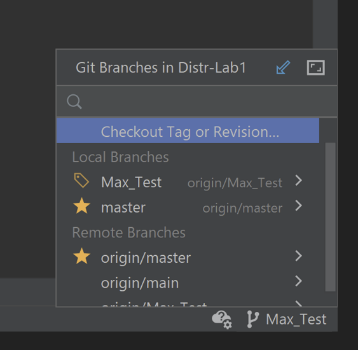
*Figure 1.1: Create Git repository*

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

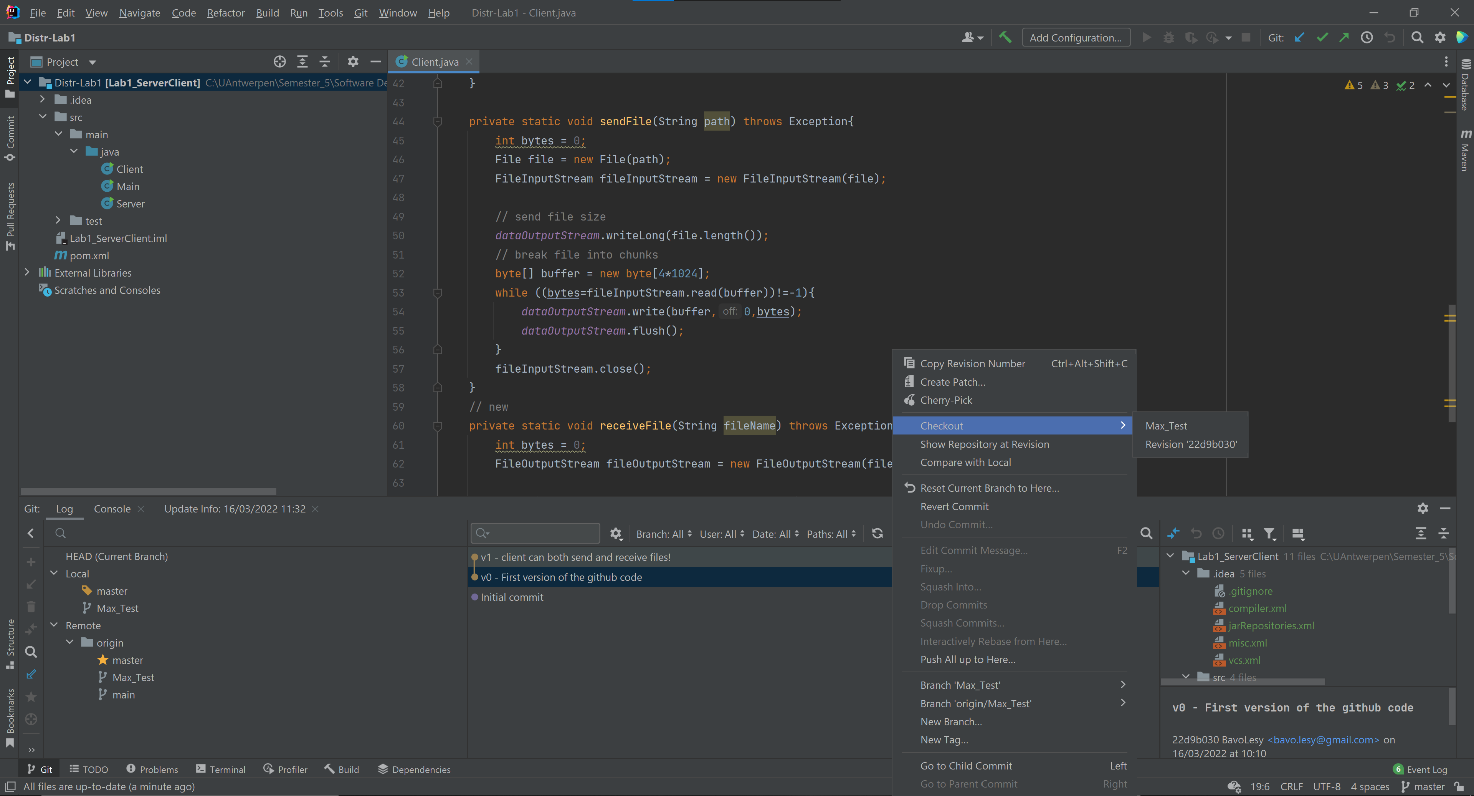
*Figure 1.3: Branches*

Afbeelding met tekst, schermafbeelding, monitor

Automatisch gegenereerde beschrijving

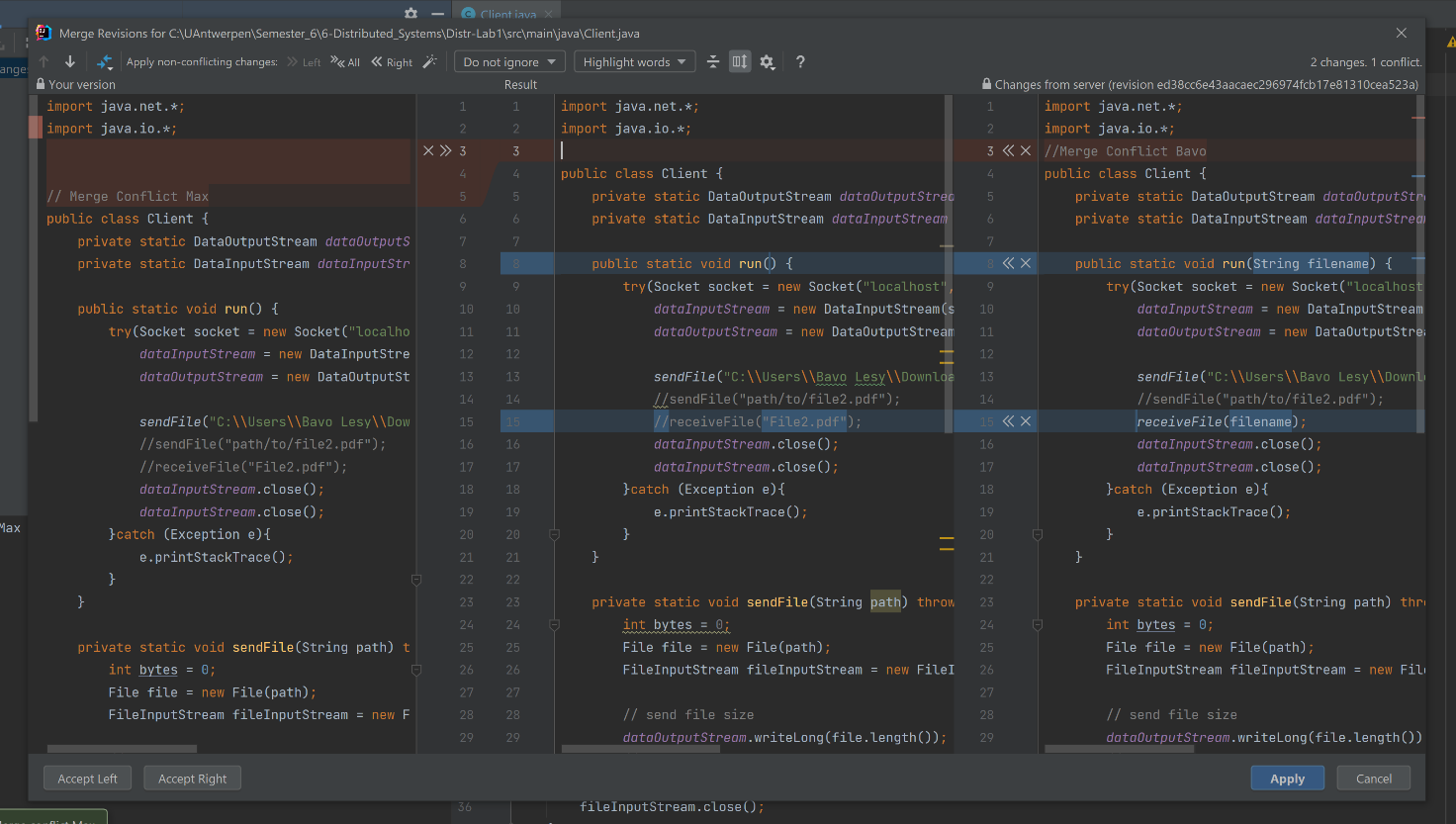
*Figure 1.4: Push code*

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 UDP

UDP is very different from TCP, the main difference being that TCP is a connection-oriented protocol. This means that a connection must be established between the two entities before data can be transferred. TCP guarantees that data is delivered in the correct order without errors. This makes TCP a reliable protocol. But it may take longer compared to UDP. UDP is a connectionless protocol, making it a faster but less reliable protocol. With UDP, data is continuously sent to the receiver whether they receive it or not.

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.

To send the file, we read it from the local directory and open a FileInputStream. Then we write the contents of the file to a buffer and use that buffer to create a DatagramPacket. Then we send the packet using DatagramSocket.send(DatagramPacket) . On the receiving end, we create an empty DatagramPacket and use DatagramSocket.receive(DatagramPacket) to retrieve the packet. Then we use getData() to read the content and write it to a buffer and get the size of the packet using getLength() . Finally, we use FileOutputStream.write(buffer, 0, size) to write the content to the file.

* 1. Filesharing Application using TCP

We created a filesharing application using TCP protocol to transfer files between server and client. To implement this application, we used the java.net framework. We used this framework to open a TCP server socket using java.net.ServerSocket and opened a client socket using java.net.Socket. We used these two sockets to establish a connection between the two.

Now that we have a connection between the two, we can start sharing files. We created our application so that clients can receive and send files from the server. We used DataInputStream, DataOutputStream, Files, FileInputStream and FileOutputstream all from the java.io library. First step was to create a DataInputStream and DataOutputStream on both server and client. So they can both send and receive data.

To send the file, we read it from the local directory and get a FileInputStream. Then we write the contents of the file to the DataOutputStream. After that, we use the flush() method to clear the stream. On the receiver side, we get the FileOutputStream and write() the contents of the DataInputStream into it. After the interaction is complete, we close the data stream and close() the socket.

* 1. Enabling Multithreading

Multithreading means that our server can serve multiple clients simultaneously on the same socket using different threads. When a client sends a request to send or receive a file, a thread is spawned that allows the client to communicate with the server. Multithreading may seem daunting at first, but it really isn't. We just moved the server's file sharing chores into the ServerThread class. The TCP server establishes a connection with the client, then starts a new ServerThread object. This object is responsible for file sharing and client-side processing. So when two clients connect to the same socket, they each instantiate a different ServerThread that can work on the same socket in parallel. For a UDP server, it's pretty much the same, just no connection to start. The server constantly listens on the socket and creates a different UDPServerThread object for each client sending 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 2.2: 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