Mandatory 1

62577: Datakommunikation

14. Marts 2019

GRUPPE 20

Anders Wagner
AHW
s185113

Frederik Reitzel
FMR
s185111

David Fager
DF
s185120

Erlend Tyrmi
ET
s132962

Justus Gammelgaard
JAAG
s185088

Max Gammelgaard
MPG
s185085

https://github.com/DePloda/DatakommunikationMandatory1.git

Table of contents

1 Introduction

The assignment was to create a FTP-Client and use it to download 2 files from an FTP-server, and upload a file if possible. The program was made without the use of the FTP-client in Java's API, but had to use a socket. The anonymous FTP server used for this program is ftp.cs.brown.edu which has options to upload files as well as download.

2 Implementation

We have built a very basic FTP client with a TUI menu. The program lets you choose a file to download by entering the path, and stores the file as a type of the user's own choosing. The connection is made using a Socket-object for sending commands to the server, and one or multiple socket-objects to receive data, depending on how many files that are transferred to and from the server. This is because the server closes the socket connection, when a file has been successfully transferred.

To be able to download a file, it was necessary to set the connection to passive mode, to avoid the connection being caught by the firewall. (In active mode the client sends a port number to the server for receiving data, but when the server tries to connect to the client, this is caught by the firewall as a foreign client trying to connect.)

2.1 Sockets and streams

```
25
26      private void prepareControlChannel() throws Exception {
27          try {
28              System.out.print("LOCAL:\tOpening socket to " + STD_URL + " on port " + 21 + " (control channel) ... ");
29              controlSocket = new Socket(STD_URL,  port: 21);
30              System.out.print("SUCCESS\n");
31
32              System.out.print("LOCAL:\tInitializing control socket streams ... ");
33              serverWriter = new BufferedWriter(new OutputStreamWriter(controlSocket.getOutputStream()));
34              serverReader = new BufferedReader(new InputStreamReader(controlSocket.getInputStream()));
35              System.out.print("SUCCESS\n");
36              serverReply( expectedReplies: 5);
37
38              System.out.print("LOCAL:\tLogging in to FTP server ... ");
39              serverCommand("USER " + STD_USERNAME);
40              System.out.print("SUCCESS\n");
41              serverReply( expectedReplies: 1);
42          } catch (Exception e) {
43              e.printStackTrace();
44              throw new Exception("!!! - Failed control channel connection");
45          }
46      }
```

Appendix 1.1 | DatakommunikationMandatory1 | G20FTPClient.java, preparedControlChannel

In appendix 1.1 it can be seen that a new socket is created which connects through port 21 on the PC, and connects to the STD_URL which is brown's FTP server. Two channels is used for the connection. The one seen here is the control channel, where the communication between the server and the client happens. The server is always listening on the control channel for a command for which it will give a reply. The prepared channels use two different streams. A bufferedwriter that writes to the outputstream on the socket, which is connected to the server, and a bufferedreader, which reads from the socket's inputstream.

2.2 Downloading files

```
180     private void downloadFile(boolean testmode) throws Exception {
181         String filepath;
182         String fileName;
183         String[] sizeReply;
184         try {
185             //Opening the dataSocket. Sender automatically closes after file transfer.
186             System.out.print("LOCAL:\tOpening socket to " + STD_URL + " on port " + dataPort + " (data channel) ... ");
187             dataSocket = new Socket(STD_URL, dataPort);
188             System.out.print("SUCCESS\n");
189             System.out.print("LOCAL:\tInitializing data socket streams ... ");
190             bufferedInputStream = new BufferedInputStream(dataSocket.getInputStream());
191             System.out.print("SUCCESS\n");
192 
193             //Asks user to specify server filepath for file to download if test is not being run
194             System.out.println("LOCAL:\tEnter the filepath for the file on the server (E.g.: /pub/README)");
195             if (testmode) {
196                 if (firstDataConnection) {
197                     filepath = "/pub/README";
198                 } else {
199                     filepath = "/u/dg/Collision.java";
200                 }
201             } else {
202                 filepath = handleUserInput();
203             }
204             if (filepath.equals("")) {
205                 filepath = "/pub/README"; //Default
206                 System.out.println("LOCAL:\tUsing default: " + filepath);
207             }
208 
209             //Ask server for the size of the file to download, to ensure the file even exists
210             serverCommand("SIZE " + filepath);
211             sizeReply = serverReply( expectedReplies: 1).split( regex: "\\s+");
212             if (sizeReply.length > 3) {
213                 throw new Exception("!!! - File not found on server");
214             }
215 
216             //Asks the user where to download the file to
217             System.out.println("LOCAL:\tEnter designated file name (E.g. MyFile.txt)");
218             if (testmode) {
219                 if (firstDataConnection) {
220                     fileName = "TestOver1KB";
221                 } else {
222                     fileName = "TestUnder1KB";
223                 }
```

Appendix 1.2 | DatakommunikationMandatory1 | G20FTPClient.java, downloadFile

To download a file from the FTP-server we have created a method called *downloadFile()* that creates a socket, which connects to the server on its data port and a BufferedInputStream, which reads the input from socket that is connected as a data channel. It has a test mode, that makes the program download two predefined files from the FTP-server in two different directories on the server. Otherwise the user can specify a file path to download from. It always downloads the file to the root of the project. The user can define the name of the file.

One of the assignments was to print the first kilobyte of a downloaded file. The method checks whether or not the file is bigger than 1kB and if that is the case it counts the first 1024 bytes of the file and prints them. If the file is smaller than 1kB it prints the whole file.

3 Work distribution

Description of the work	Responsible
Research and implementation of the FTP client in java.	David Fager & Erlend Tyrmi
Writing the implementation & test chapters of the report.	Anders Wagner & Justus Gammelgaard
Writing the remaining chapters of the report.	Max Gammelgaard & Frederik Reitzel

4 FTP commands

FTP-Commands used in the program	
USER [userName]	Sends login username to server.
RETR [yourFilePath]	Retrieve file from server on data channel
TYPE I	Sets servers transfer type to binary
PASV	Server enters passive mode
CWD	Change working directory
STOR	Stores a file on the server
SIZE	Server sends size of given file
Other useful commands	
LIST	Lists files and folders in current directory
PASS	Sends login password to server
PORT	Specifies a port (in active mode) to transfer data to.
DELE	Delete remote file

4.1 Minimum command usage

When transferring a file from an FTP server to the local machine running the client, the client is as a minimum required to use these commands, depending on the conditions.

Some FTP servers require a login, and then the raw FTP command: **USER** [userName] needs to be used. Some of those FTP servers also require a password, and then the command: **PASS** [password] follows the USER command.

When the FTP server has accepted the user, it has to either be set to passive mode, so the user can connect to the returned port or the server has to know which port on the users machine, that is shall connect to. These commands are **PASV** for passive mode or **PORT** followed by the IPv4 and port numbers.

Then the server needs to know how to transfer the file, that is what structure type of transfer it should do. For this the command **TYPE** [structure-character] is used, where TYPE I is binary for instance.

The last step is telling the server which file it should send on the data channel, by using the **RETR** [filePath] command, where the file's path on the server is specified. This command makes the server transfer the file on the data channel which is either connected by the passive port on the server or specified user port and then the goal is achieved.

4.2 Other commands

Should the client have been more comprehensive, then commands like **LIST** and **CWD** would have done great deeds. This is because it would have allowed the user to navigate through the FTP servers directories, and then been able to find files and directories to download and upload, instead of having to rely on the web interface, of the server.

Furthermore, if the FTP server had not been anonymous, and therefore restricted in what connections are allowed to do, then **DELE** would also have been an important command, so that the client could delete unwanted or unused files from the server.

5 Tests

Three tests were done.

1. The first one for downloading a file with less than 1 kb. That file should be printed immediately in the program and saved locally.
2. The second test was for a file larger than 1kb, which should be saved locally and only the first kB printed. The files where downloaded from different directories on the server.
3. Third, a file was uploaded to the 'incoming' folder, and the upload was verified by visiting the FTP server via a web browser.

6 Conclusion

A lot of research was necessary for this project. Finding out how to read from the data port and write from a binary stream to a file was the main bottleneck. Reading the replies on the control channel from the FTP server was also more difficult than expected, as there was a recurring problem that the connection froze as soon as the program checked if there was a new line to read. The assignment put great demands on the groups ability to read documentation.

Although the challenge to overcome this task was steep, the process of implementing a simple FTP-client gave us a greater insight in how client-server communication is handled. Furthermore, it was discovered that RFC's, although heavy with text, can be useful for understanding how protocols like the FTP work. On top of general understanding of the File Transfer Protocol, the Request for Comments for the FTP (RFC 959) contains commands needed to utilize the FTP in our client.

All in all we felt that the result was satisfactory, as the program passed our manual tests.