Université de Liège

Ingénieur Civil - 3ème BAC

December 20, 2019

Introduction to computer networking

Second part of the assignment

SIBOYABASORE Cédric 20175202
BULUT Stephan 20172244

1 Software architecture

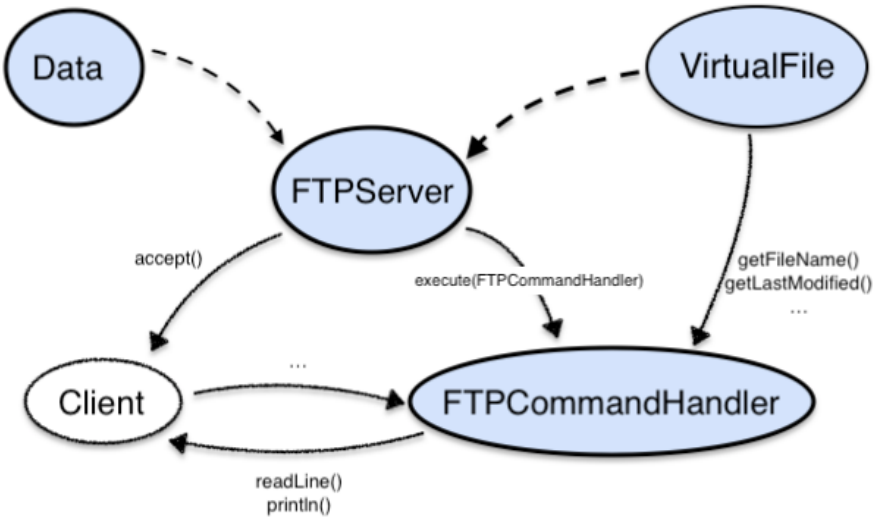
We have broken down the problem into 4 classes that interact in a object-oriented way:

`FTPServer.java` denotes the FTP Server class. It implements the main void which creates the FTP server socket that accepts the client sockets. Once a client is connected to the server he is redirected to a personal *FTPCommandHandler* thread.
The FTPServer class initializes two *VirtualFile* list structures in which the initial files (mytext.txt, myimage.bmp, secret.txt) are added. They are thus only added at server startup. The data from these files are taken from the Data class . It also initializes the currDir *String* list (whose latest element is always the current directory).

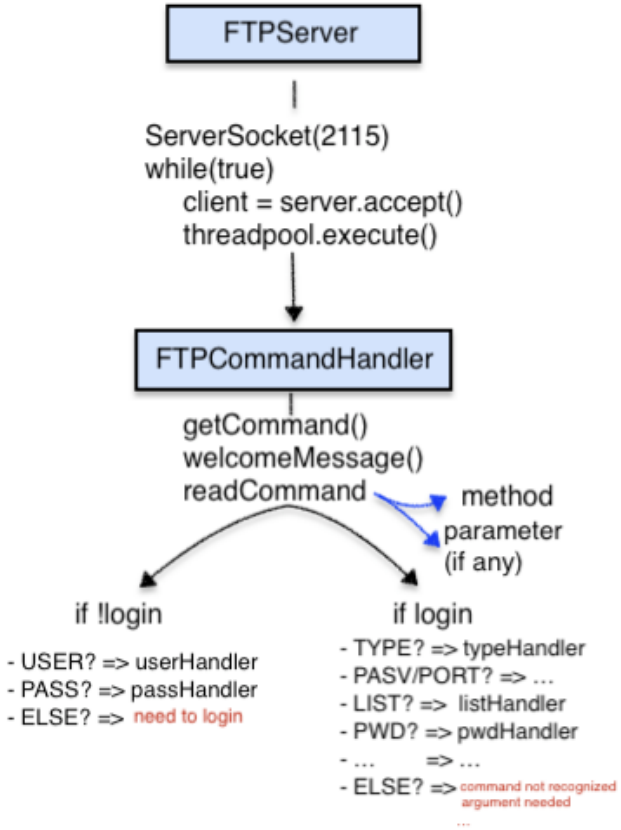
`FTPCommandHandler.java` is the thread class that handles all the requests that the considered client might send to the server. The thread is instantiated with the considered client, the two *VirtualFile* list structures and the currDir list. It reads the FTP commands that the client sends in the stream then it sends a customized three-digit FTP response to the client. FTPCommand-Handler implements a personal method for each FTP request that the server can handle in order for the code to be more modular. For the encapsulation aspect of the object-oriented coding of the class we group together as set all the private variables that every method can modify (command handling).

`Data.java` is the class containing the data of the initial virtual files and folders ("Irasshaimase" in mytext.txt,"UPUPDOWNDOWNLEFTRIGHTLEFTRIGHTBASTART" in secret.txt and the image in byte array).

`VirtualFile.java` is the class that allows us to virtually represent files and folders (by their names, their size, their content..). Each object of this class is virtual (stored in the RAM) and doesn't actually "exist". We made a distinction between image files and the other types of file (text..) : image files must enter a byte array as data in their VirtualFile constructor and text files have String data. A few more methods (`getFileName(), getFileSize(), rename(newname), ...`) are implemented in this class to modify the virtual files/folders attributes which are private.



2 Program Logic



In `FTPServer.java` in Lign 51 the server is created with the method `ServerSocket` and listens on port 2115. We then create the thread pool with a maximal number of threads *maxThreads*. In a while(true) loop, we accept new client connection with the `accept()` method called on ServerSocket(Lign 62). Thanks the setSoTimeOut called on client (Lign 63) and the try/catch block that catches an IOExceotion, the server will disconnect from the client if the client hasn't sent a command in a timespan of 15 minutes. A new `FTPCommandHandler` thread is created for the current client (Lign 64) then it is executed (Lign 65).
In `FTPCommandHandler.java` (in *run* method) we get the command sent by the client (Lign 62) through the *getCommand* method (Lign 300). That method instantiates a BufferedReader to be able to later read the command sent by the client. In the *run* method, we create a PrintWriter (Lign 64) that will be used to send messages to the client. Then we welcome the client with a warm message (Lign 312) then we process the client requests in a while(true) loop (Lign 67) so that the server can process multiple requests from the same client. We read the command BufferedReader with *readLine* and place the returned string in a readCommand variable (Ligne 68). We split readCommand into an array string (Ligne 78) and the first element of the array corresponds to the method of the command. If the array length is 2, the command takes an extra parameter and the second element of the array corresponds to the parameter.If the login boolean variable is false, the client could only have sent a USER, a PASS or an invalid command (login is set to true in PASS so client cannot make any other commands unless he sends a valid user and password). We check those cases via switch. If the client has sent a valid username (via USER) and a valid password (via PASS) the boolean variable login is set to true and the client can send

other commands. Each command is handled in its own method. The commands that take extra parameters have their personal methods called with the parameter as arguments. If client sends an unrecognized command (boolean recognized = false) or sends a command with a parameter (when the command doesn't take an extra parameter, boolean noArgument = false) or a command with no parameter (when the command takes an extra parameter, boolean validArgument = false), a customized error message is sent (Ligns 274-285).

3 FTP protocol

We implemented the operations that FileZilla demands. All the commands we implemented are in RFC 959 (except MDTM. We precise that in the FEAT response). We didn't implement the creation and deletion of folders (and we forbid folder renaming).

Requests & reply codes	
USER	331 Password required 430 Invalid username
PASS	230 Successfully connected 430 Wrong password, retype username and password 530 Username not valid
SYST	215 UNIX Type: L8
FEAT	211-Features MDTM 211 End
PWD	257 ".." created
CWD	250 Working directory changed. 500 Impossible to access to this parent directory/The directory mentioned doesn't exist
CDUP	200 Changed to parent directory. 500 The directory has no parent.
TYPE	200 Transfert mode set to A/I 501 Unrecognized parameter
PASV	227 Entering Passive Mode ("ipAddress","x","y")
EPSV	229 Entering Extended Passive Mode (|||" port "|)
PORT	200 Active data connection created 504 Port number must be greater than 1023
EPRT	200 Extended active data connection created 504 Port number must be greater than 1023
LIST	125 Listing initation... 250 Listing ok. 503 Data connection needs to be opened first 504 Folder specified doesn't exist
MDTM	(file last modification) 550 File not found in current directory
RNFR	350 Requested file action pending further information 550 File "filename" doesn't exist in current directory 504 Server cannot rename folders.
RNTO	250 File renamed 503 You need to specify which file you want to rename (use RNFR).
RETR	226 Download successful 550 File "+filename+" doesn't exist in current directory
DELE	250 File deleted 550 No file named "..." in current directory
STOR	150 File downloading 226 Upload successful. Closing data connection
- boolean -	530 Need to log in first 501 Command needs parameters 502 Command not recognized 503 Data connection mus be opened first (for LIST, RETR & STOR) 504 Command doesn't take extra parameters

4 TCP stream

In order to respect the stream-oriented property of TCP, the requests from the TCP Streams are read this way :

```
InputStream in = client.getInputStream();
BufferedReader read = new BufferedReader(new InputStreamReader(in));
String readCommand = read.readLine();
```

We implemented a `getCommand()` that performs the two first lines (for better readability). The `readCommand` string contains the command that the client sent thanks to the `readLine()` method that performs efficient character-input stream reading.
The BufferedReader `read` reads line inside a `while(true)` in order to wait messages from the client.
We use

```
OutputStream out = client.getOuputStream();
PrintWriter controlResponse = new PrintWriter(out, true);
```

and the `controlResponse.println()` method to send a response to the client via the control channel. The autoflush variable of the Printwriter constructor is set to True to avoid having to flush our response after every println, it will automically be sent to the client after calling `controlResponse.println()`.

5 Multi-thread organisation

In order to avoid too many simultaneous connections to the Server, we set a threadpool thanks to `Executors.newFixedThreadPool(`*maxThreads*`)` from `java.util.concurrent.*` where *maxThreads* is an argument that denotes the maximum number of FTPCommander threads that can run simultaneously. The server will handle *maxThreads* clients and their requests and when the server's done with a client it handles another one that was waiting in the pool. This way we can avoid DDos attacks that could potentially crash the server. It might decrease speed performance but it makes the server more robust.

```
ExecutorService threadPool = null;
try{
        threadPool = Executors.newFixedThreadPool(maxThreads);
}
catch(IllegalArgumentException i){
        System.out.println("Maximum number of threads cannot be negative !");
        System.exit(1);
}
while (true) {
        Socket client = server.accept();
        client.setSoTimeout(15*60000);  // Time-out of 15 minutes
        FTPCommandHandler r =
        new FTPCommandHandler(client,fileList,folderList,currDir);
        threadPool.execute(r);          //Execute thread
}
```

We create the thread pool in a try/catch block to catch an IllegalArgumentException if server was invoked with maxThreads ≤ 0. We then execute the *FTPCommandHandler* thread (subclass of Thread) for a client from the thread pool via the `execute` method from the

ExecutorService class.

If multiple clients are simultaneously connected to the server (multiple threads are running) they might simultaneously attempt to modify (rename,delete,...) the shared server virtual files which would lead to corrupting the server.
This is why we included the *synchronized* attribute in methods definitions that handle commands that modify the server's state (because we implemented a method for each command):

-public *synchronized* void **listHandler**(String filename) that handles the LIST command.
-public *synchronized* void **deleHandler**(String filename) because deleting a file named filename from the server modifies the server's state.
-public *synchronized* void **rnfrHandler**(String oldname) and public *synchronized* void **rntoHandler**(String newname) because renaming a file that's on the server modifies the server's state.
-public *synchronized* void **retrHandler**(String filename) because downloading a file filename from the server modifies the server's state.
-public *synchronized* void **storHandler**(String filename) because uploading a file filename to the server modifies the server's state (more files on the server)

6 Robustness

In `FTPServer.java` we check if the additional argument *maxThread* is a number. If not then we throw a NumberFormatException. We also check if *maxThread* is greater than 0. If not then a IllegalArgumentException is thrown. As mentioned previously, we handle possible multithreading-attacks from malware thanks to the thread pool.
We also implemented the program in order to handle every valid command and return error messages to the others. We manage commands authorizations through **boolean variables**.

We divided the commands in two types : no-argument commands & single-argument commands *(commands which have more than 1 argument are prohibited)*. In every single-argument command, we firstly check the validity of the argument. Then, we try to proceed with the command for both types. If an error is detected, we send to the client a customised error message.

The first command needs to be a valid `USER` command *(i.e "Sam" or "anonymous")*(boolean **validUser true**) followed by a valid `PASS` command *("123456" for Sam)* to allow other commands. If one of the two previous commands is not valid, the FTPServer doesn't accept other commands until valid username and password are submitted (boolean **login true**).

Once that is done, a couple of commands are sent by the client (`SYST, FEAT, PWD`) which are handled standardly.

Commands which establish a Data Connection (such as `PASV, PORT, EPSV & EPRT`) are inside try and catch blocks in order to avoid Data Connection errors *(which throws IO Exceptions)* .If the command and the data connection is open then we set the boolean **dataConnexion true**).

Commands which deal directly with directories (such as `CWD, CDUP, LIST`) are checked meticulously. We generally check what is the current directory and then we check the validity of the operation. For CWD, we ensure that the given working directory exists. As well, for CDUP, we check if the current directory has a parent. Likewise, for LIST, we check if the folder exists

(and try and catch data-connexion operations).

Similarly, commands which deal directly with files (such as `MDTM, RNFR, RNTO, DELE, RETR & STOR`) are also checked meticulously. We check the current directory and we don't allow operations on file if we are currently in the wrong one. For RNFR & RNTO, we check if the file to rename exists and if it is in the current directory. We check this condition thanks to a function called *existsInDir* that we implemented.We check the same condition for MTDM, DELE.
The data connection must firstly be open if the client wants to execute the LIST, RETR and STOR commands (we close it at the end of the command). If the data connection is closed when entering the method (dataConnection = false) then we send a 503 error message (cfr Tab in FTP Protocol section). In RETR & STOR, in addition to check same conditions, we added a deeper verification : we check file extensions before downloading the file (to check if it is indeed an image in binary that we want to download) and we convert UpperCase extension (i.e. "BMP","JPEG", ...) into LowerCase extension to extend the image extension definition ("BMP" vs "bmp"...).