INFO0010-4: Project 2 - Report

BARE Alexandre - s172388
RANSY Bastien - s174413

December 20, 2019

1 Software architecture

We divided our code into 7 classes. They are represented in Figure 1 :
FTPServer.java is the main class which initiates the sockets and the virtual directory, and delegates the tasks to the threads available in the thread pool when a client wants to connect (cfr. *FTPServerThread.java*). When a connection is made, the thread handles the requests (according to the RFC standards) coming from the client by replying with a successful message if the request has been handled without any problem, or with a negative response in case of a fail during the process. Those exchanges take place in the control channel, which is initiated each time a new control connection is made.
When it comes to data transfers (for downloading, uploading a file or listing the content of a directory), the data channel is opened to complete the requested process.
The files handled on the server are virtual : our class in *VirtualDirectory.java* creates the virtual directory on the RAM and manipulates virtual files (as implemented in *VirtualFile.java*). The initial content of the files and directories is in the class in *VirtualDirectoryContent.java*. *VirtualDirectory.java* is responsible for the navigation through the directories and for the uploading, downloading, renaming and general manipulation of the virtual files in the directory.
Finally, we made a in *InvalidStringFormatException.java*, a class which extends the *Exception* class of Java and therefore throws exception messages when the format of a string (mostly used for filenames) is not valid. And in the same way, we made a class in *TransferSizeExceededException.java* that handles the exceptions linked with too bulky transfers of data.

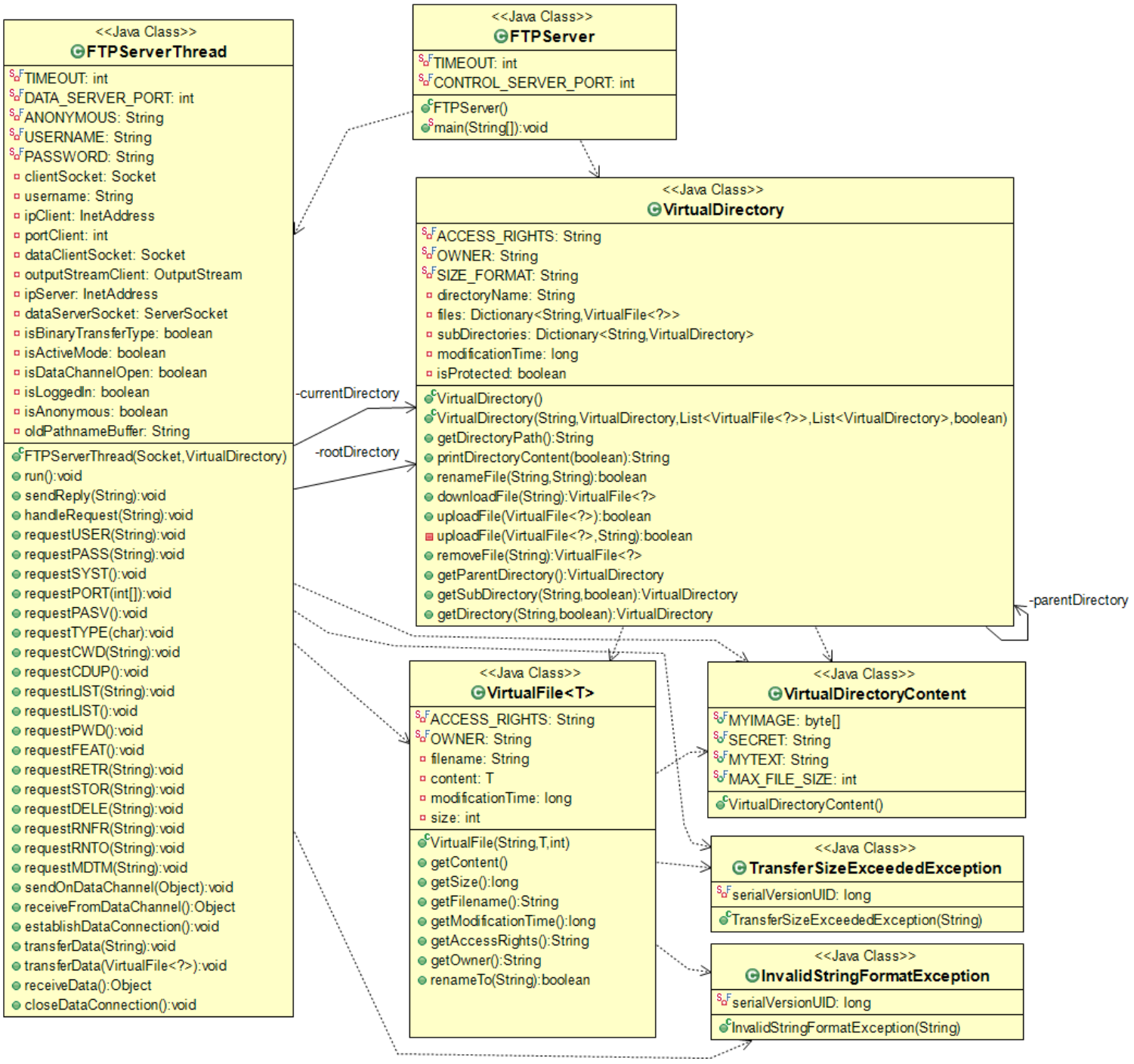

Figure 1: Diagram of classes

2 Program Logic

Our server works the following way :

- First of all, in *FTPServer.java*, the server socket is created as well as the thread pool and the virtual directory. Then the socket time out is set, and the server listens on the port 2151 (control connection). The virtual root directory and its sub-directory are loaded in the RAM. The root directory contains the virtual files *"mytext.txt", "myimage.bmp"* and the folder *"private"* which contains the virtual file *"secret.txt"*. The class *VirtualDirectory.java* handles the loading of the directories and delegates the loading of the content of each file to *VirtualFile.java*. (*FTPServer.java - l.18 to 22*)

- *FTPServer.java* listens continuously for an incoming control connection. When a client finally connects to the server, a new thread is created to delegate the requests handling to *FTPServerThread.java*. (*FTPServer.java - l.26 to 30*)

- The (thread) server asks for the username, and then listens for requests. (*FTPServerThread.java - l.74 to 84*)

- When a request is received, the code identifies its content by dividing it into 2 Strings separated by a space : the first one corresponds to the command, the second to the argument if there is one. The argument is also split into smaller strings separated by a comma (which will be useful to identify the different arguments of a PORT request. In this assignement, it is the only request which works with multiple arguments but the code is ready for further commands implementation of such type). (*FTPServerThread.java - l.131 to 136*)

- The server then performs a switch-case sequence, associating each command to a function for specific request handling. If there is no match, a message specifying that the command hasn't been recognised is sent. (*FTPServerThread.java - l.138 to 190*)

- The command is handled by its corresponding function; the list of the handled commands and possible responses can be found in section 3. (*FTPServerThread.java - l.209 to 732*)

- When a data transfer is required, the type of data transfer is sent through the command *TYPE*. Binary and ASCII transfers are handled. The choice is up to the client. (*FTPServerThread.java - l.372 to 387*)

- The server, then, needs to establish the data connection. It depends on the data transfer mode (*FTPServerThread.java - l.834 to 848*)
  In active mode, a socket is created with the IP address and the port number of the client, which were communicated using the *PORT* command

(The first 4 numbers of the argument are the IP address, the last 2 numbers determine the port number through the formula $port = nb_5 \times 256 + nb_6$). In passive mode, the server creates a new server socket on a chosen port (2051) and listens on it. The client can initiate a connection to the server thanks to the response given by the server to the *PASV* command, which contains the IP and port of the server data socket in the same format as with *PORT*
The data is then sent on the data channel in the format specified by the type (ASCII or Binary) via the *write()* method of the *OutputStream* object for binary transfers, or of the *OutputStreamWriter* object for ASCII transfers. 3 commands can lead to a data transfer: *LIST*, *RETR*, STOR (*FTPServerThread.java - l.749 to 778*)

- When the data transfer is complete, the data connection is closed (while the control connection is kept open) : the client data socket is closed and the data server socket too if the passive mode was used. (*FTPServerThread.java - l.951 to 956*)

- *STOR* can modify the content of the directory. It creates a new VirtualFile object filled with the file content received through the data channel. (*FTPServerThread.java - l.605 and 608*)

3 FTP protocol

The requests handled by the server and their respective possible replies are :

- NB 1: All the commands that have arguments can send a reply advertising a syntax error in the arguments ("501").

- NB 2: We have arbitrarily chosen that all the commands that do not have any argument won't reply with the error code "501" but will simply not take the arguments into account. In some situations, a more detailed "501" reply could be handled (see below).

- NB 3: If the command is not recognised, the error code "500" is sent.

- **USER** : The server responds with an approbation to log in if the user is anonymous ("230") or else, with a demand for the password ("331").

- **PASS** : The server responds with an approbation to log in if the password and the username are correct ("230"), or with an error message if at least one of the two is not correct ("430") or no username has been sent ("332").

- **SYST** : The server responds with the server system ("215").

- **PORT** : The server responds with a successful message advertising the entering in active mode ("200"), or an error message if the client is not logged in ("530"), the number of arguments is not at least of 6 (only the

6 first are considered) ("501"), the IP address is not valid ("501"), the 6 first arguments are not in range [0; 255] ("501").

- **PASV** : The server responds with a successful message advertising the entering in passive mode associated with the 4 numbers of the server IP address and the 2 numbers (X and Y) to compute the client port number: X * 256 + Y ("200"), or an error message if the client is not logged in ("530").

- **TYPE** : The server responds with a successful message advertising the change of transfer type - Binary or ASCII ("200"), or an error message if the parameter is not a unique character ("501"), if the client is not logged in ("530"), or if the parameter is invalid - neither 'I' (for Binary), nor 'A' (for ASCII) ("504").

- **CWD** : The server responds with a successful message advertising the change of working directory ("250"), or an error message if the client is not logged in ("530"), or if the directory can't be found with the path given in argument ("550").

- **CDUP** : The server responds with a successful message advertising the change of working directory ("200"), or an error message if the client is not logged in ("530") or there is no parent directory ("550").

- **LIST** : The server responds with a successful message advertising that the directory has been found (if there was an argument) and that the server is about to open the data connection ("150"), and/or the data was correctly transferred ("226"); or an error message if the client is not logged in ("530"), the directory can't be found with the path given in argument (if there was an argument) ("451"), the data connection can't be opened ("425"), the transfer is aborted because of an error when writing in the data channel ("426"), the transfer is aborted due to either the server or the client data socket that has timed out ("426").

- **PWD** : The server responds with a successful message advertising the path of the current directory ("257"), or an error message if no directory has been loaded (yet) ("550").

- **FEAT** : The server responds with a multi-line message where on each new line, is specified an additional feature implemented in the server that goes beyond those defined in RFC959 ("211").

- **RETR** : The server responds with a successful message advertising that the file has been found and that the server is about to open the data connection ("150"), and/or the data was correctly transferred ("226"); or an error message if the client is not logged in ("530"), the file can't be found ("550"), the data connection can't be opened ("425"), the transfer is aborted because of an error when writing in the data channel ("426"),

the transfer is aborted due to either the server or the client data socket that has timed out ("426").

- **STOR** : The server responds with a successful message advertising that the filename is valid and that the server is about to open the data connection ("150"), or the data was correctly uploaded ("226"); or an error message if the client is not logged in ("530"), the filename is not allowed ("553"), the data connection can't be opened ("425"), the transfer is aborted because of an error when writing in the data channel ("426"), the transfer is aborted due to either the server or the client data socket that has timed out ("426"), the data to receive has exceeded the maximum transfer size ("452") or there was a processing error in the uploading of virtual files ("451").

- **DELE** : The server responds with a successful message advertising that the file whose filename was given in argument is deleted ("250"), or an error message if the client is not logged in ("530"), or if the file can't be found ("550").

- **RNFR** : The server responds with a successful message advertising that the file has been found and that the server is waiting for further information to rename the file ("350"), or an error message if the client is not logged in ("530"), or if the file can't be found ("550").

- **RNTO** : The server responds with a successful message advertising that the file has been renamed ("250"), or an error message if the client is not logged in ("530"), the new filename is not allowed (only ASCII characters without '/' are allowed) ("553") or an "RNFR" request has not been done previously ("503").

- **MDTM** : The server responds with a successful message advertising the last time the file was modified in milliseconds since 1st January 1970 ("213"), or an error message if the file can't be found ("550").

4 TCP stream

The requests are read via the *readLine()* method of the *BufferedReader* object, which is constructed with an *InputStreamReader* object wich is himself built with the *InputStream* of the Socket *clientSocket*. The request reads are then looped in a while loop :

```
InputStream in = clientSocket.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(in));
String request = br.readLine();
try {
while(request != null) {
```

```
        if (request.length() > 0) {
                handleRequest(request);
        }
        request = br.readLine();
}
}finally {
    br.close();
    clientSocket.close();
}
```

In this way, the number of I/O operations is minimised because a buffer stores packs of characters, from which the reader reads until a "\r\n" character is encountered. Multiple reads are then taken into account and we can't lose information.

5 Multi-thread organisation

We create a new thread each time a client wants to connect with the server in *FTPServer.java*. Once this process is achieved, *FTPServer.java* delegates the task to the thread in *FTPServerThread* and listens for other connections. To prevent the flooding of the server if too much threads are running at the same time, we created a thread pool which limits the amount of threads running concurrently. Indeed, if the pool is full, the client that is willing to connect to the server will have to wait that a thread has finished its task. The code responsible for this is:

```
ExecutorService threadPool = Executors.newFixedThreadPool(
    maxThreads);

        try {
                while(true) {
                        Socket clientSocket = serverSocket.accept();
                        clientSocket.setSoTimeout(CLIENT_TIMEOUT);
                        clientSocket.setTcpNoDelay(true);
                        FTPServerThread serverThread = new
                            FTPServerThread(clientSocket);
                        threadPool.execute(serverThread); // if a thread
                            is available in the thread pool, assign to this
                            thread the work of serverThread
```

where *maxThreads* (the maximum number of concurrent threads in the pool) is specified in argument when launching the main server.
As the common data shared by the different threads is the virtual directories and files, we "synchronized" the methods of the respective classes (*VirtualDirectory.java* and *VirtualFile.java*) that manipulate the shared data. Each lock is

thus associated to one of the 2 classes. It is, thus, impossible for a thread to modify or access the content of a virtual directory (directory name, a reference to the parent directory, the reference to the files or the sub-directories, the last time the directory was modified, ...) (resp. the content of a virtual file (the last time a file was modified, the size of a file, the filename, the content of the file, ...)) if another thread has previously acquired the lock to the corresponding class without having released it yet.

6 Robustness

At each stage of the execution, we try to manage as much predictable exceptions as possible.
In *FTPServer.java* main() method, we catch exceptions of class:

- IllegalArgumentException in case the command line argument (the maximum number of concurrent threads) is invalid (either 0 or a non-integer number or not a number)
- SocketTimeoutException in case the server times out
- UnknownHostException in case the server IP address could not be found
- Exception in case of unforeseen exceptions and to handle the very high number of possible exceptions. It notably handles exceptions due to unforeseen problems at the initialisation of the virtual files and directories at the beginning of the execution

In *FTPServerThread.java* run() method, we catch exceptions of class:

- SocketTimeoutException in case the client response times out
- Exception in case of unforeseen exceptions and to handle the very high number of possible exceptions

These exceptions would lead to the server dying.

But other exceptions are caught in the different methods to send an error reply to the client and prevent the server from dying.

We made our server robust by making sure that every possible request is handled by the server: if a command that was not implemented is requested, the server replies that the command wasn't recognised.

Furthermore, for every command handled that takes at least one argument, an error message is sent each time the request has not the expected format. The expected format is mentioned in the message so that the user knows the origin of the problem and how it can modify its requests.

There are also a lot of type conversions in this project; we then throw an error when it fails, specifying why (e.g. if it tries to convert an integer which is too big to be represented with 8 bits, such as for the arguments of PORT).

We also set socket timeouts to prevent a malevolent user from flooding the server by maintaining connections open indefinitely.

We also check that the objects passed in method arguments are not null to prevent an exception of class *NullPointerException* to be thrown.

We also implemented our own exception classes to handle these 2 specific situations:

We set a maximum file transfer size (to 2^{16} bytes) to prevent too big file transfers from blocking the server threads.

We ensure that a user can't rename files with forbidden characters such as non-ASCII characters or "/", which is reserved for separating the directories (potentially also from a file) in a path.