|  |
| --- |
| Report |

Computer networking –

TFTP server and client

*Laboration 3*

|  |
| --- |
| *Author:* Christofer Nguyen & Jonathan Walkden  *Subject:* Computer network  *Corusecode:* IDV701 |



# Introduction

The report will be containing the task for laboration 3. Laboration 3 goes through the process of creating a TFTP server, Trivial File Transport Protocol that is a smaller version of FTP. Trivial file transfer protocol is a simple transfer protocol that does not use any authentication unlike FTP. It is less reliable, in transport layer it uses UDP and all packages are ACK separately. There is a startup code for this laboration, that includes handling one or multiple connections in threads and all the method where defined for TFTP. The basic functionality of transferring and receiving acknowledgement are to be implemented in the methods that are defined.

The purpose of this laboration is to understand the TFTP and how it works.

# Partner Participation

In this section the participated work will be shown in percentage in relation to the time spend in this laboration.

People involved in this work:

1. Christofer Nguyen – 50%
2. Jonathan Walkden – 50%

In this project both team members participated in code segments of all task. All code was handled and discussed together. The members in the team worked part wise for each code segement because the recommended approach to solving this laboration was to implement each part separately until they worked. The step by step approach meant that each of the team members had to work on one code segment per day. Resulting in each member having to be as much engaged in the coding, giving a complete understanding for the whole work.

Christofer worked on the RRQ, WRQ and error handling while Jonathan worked on setup, block and timeout handling. NOTE! This only consider the majority part almost all of these code segments where implemented with the help of each team member. Each team member also did do research of the particular area resulting in having a easier time implementing the code together(having a common overview).

Final parts of work was the report and fine tuning the code for easier readability, whereas Christofer did the most of the report and Jonathan the fine tuning and commenting of the code.

# Assignment Task

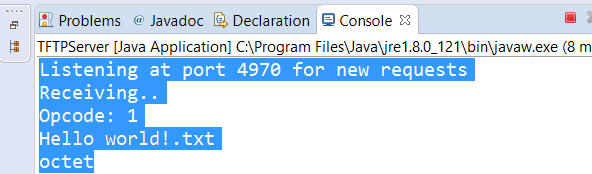
In this chapter the laboration task will be presented together with picture to illustrate and describe the solutions. The code that are implemented are mostly functionality to the empty predefined methods included in the startup code. The simulated client that is utilized in this laboration is a application called tftpd64. The transfer mode used for all task is the “octet” or binary transferring.

## Task 1

In this task we are to implement the basic functionality of reading a request from the client.

**Solution:**

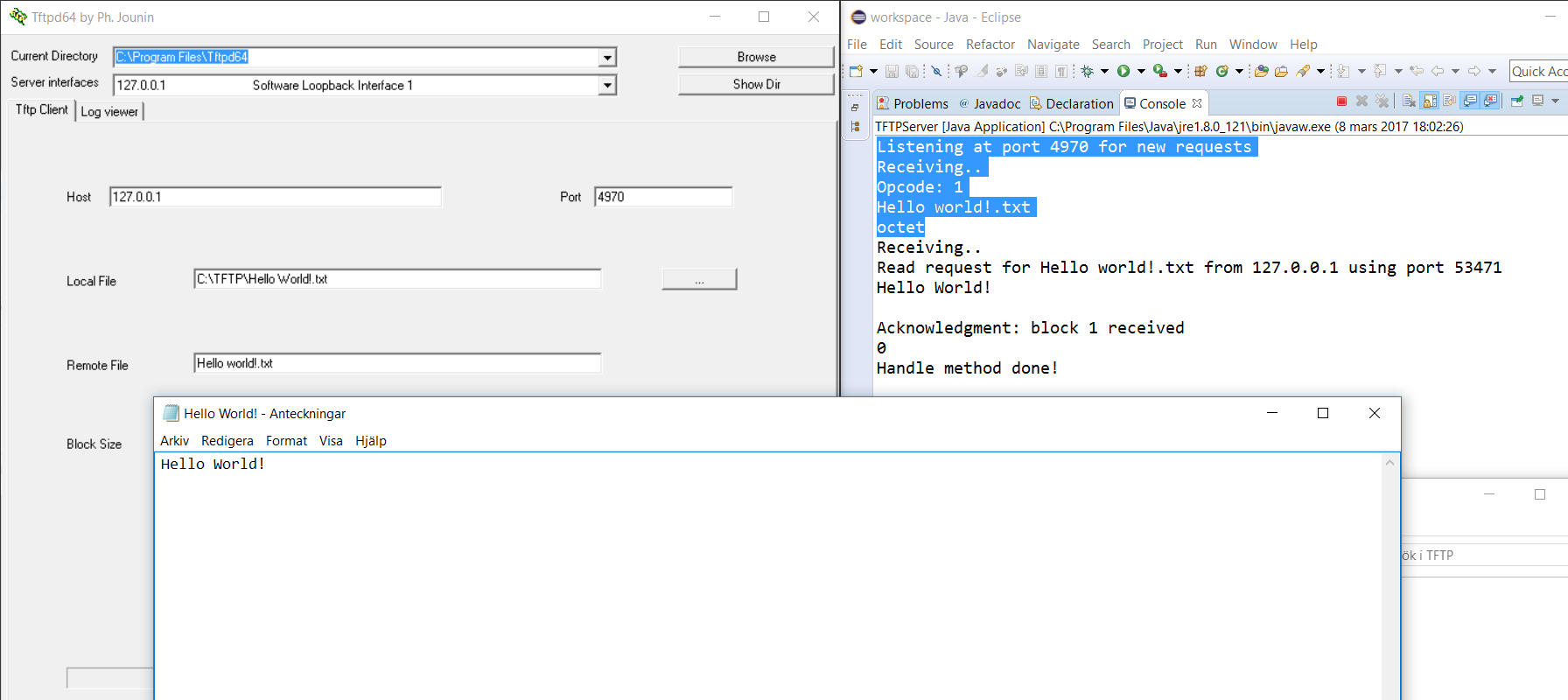
The first step is to establish a connection and accept a request from the client. We then take this information to parse and read the request that was send from the client. That is shown below in the console after requesting to read a file from the server.



The picture above shows the client requesting a read request from the opcode 1, the filename “Hello world!.txt” and the transfer mode which is binary. The same method is used later on to parse a write request with the opcode 2. The first two bytes in the request contains the opcode.

Now that we have information about what request the client wants, we can use this to create a response. The response in a read request means that we are sending data to the client for them to view. According to how the RFC1350 protocol works is that after each package we are to have an acknowledgement. In the read request we are getting the ACK from the client after they have successfully obtained the data.

Here is a illustration of the actual runtime requesting a file from the server shown in the picture below.



The client tftpd64 must know the ip address of the server and the remote file that wants to be requested. The local file is used here in order to read the data received from the server into the local file else there is no way to display the data received. The above picture also showcases blocks handled but they are not covered in this task.

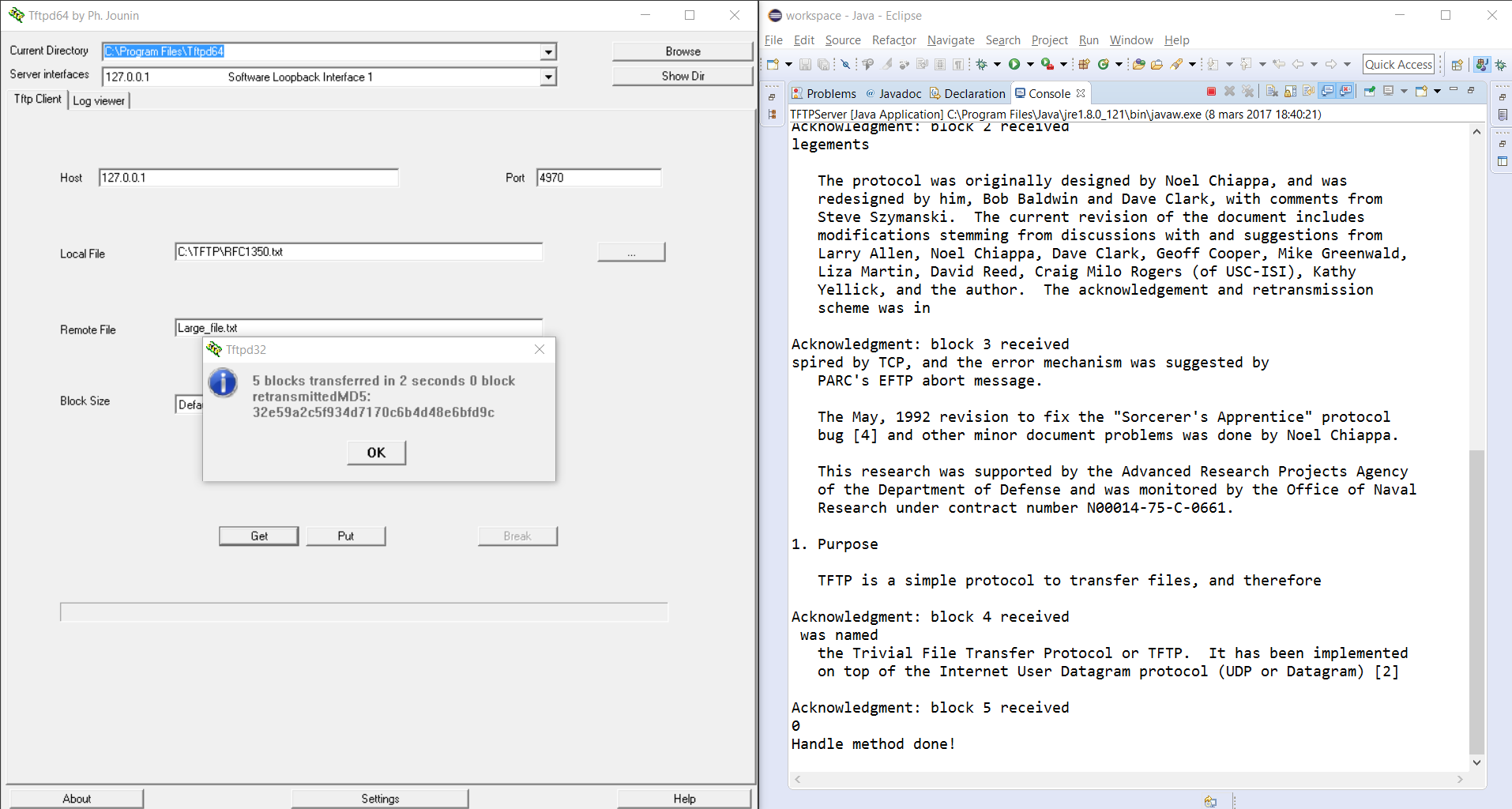
In the connection we are using two different sockets one for listening to the connections and another to handle the requests from the client. In this particular case it is socket for awaiting connections and sendsocket for handling the request send from the tftpd64.

## Task 2

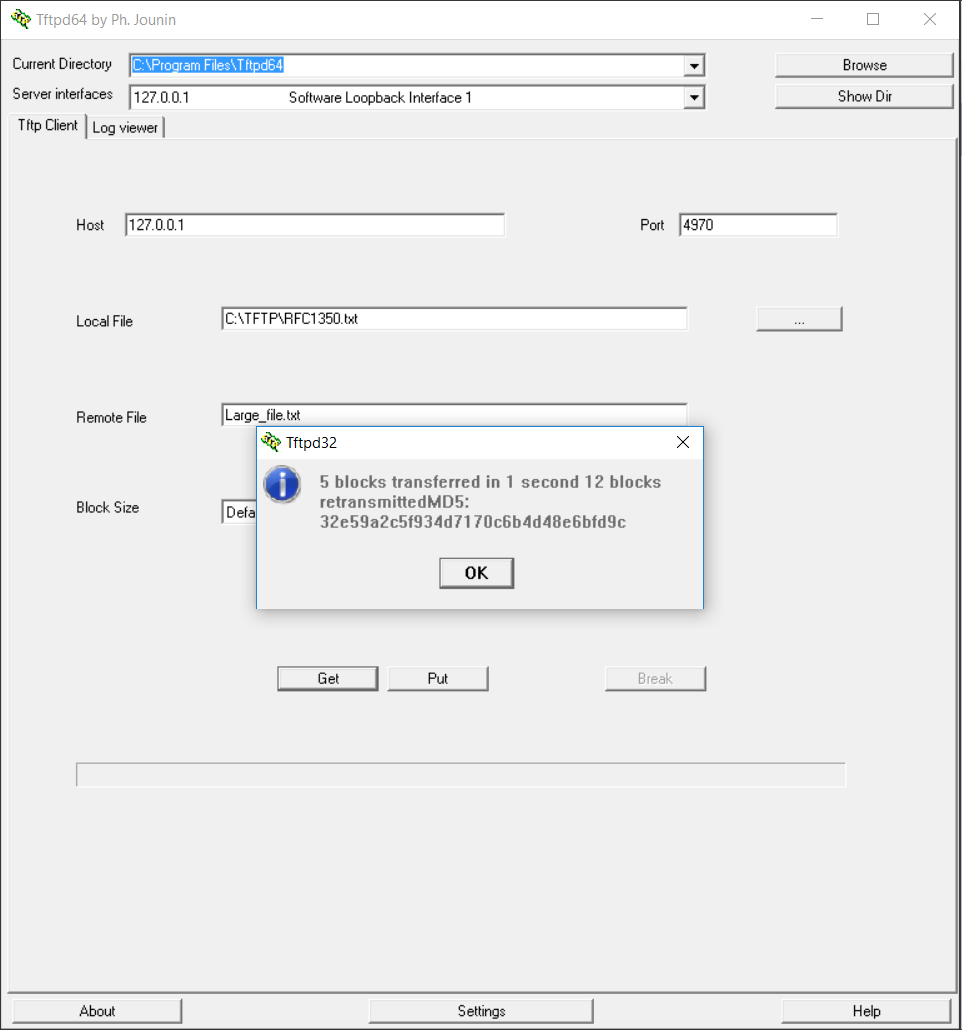
Here we are to implement block functionality to the specified RRQ in order to handle files bigger than 512 bytes. We are to also implement a timeout function for the request, this timeout is for when we are not receiving a ACK within the time range.

**Solution:**

The buffersize that are used in the code is 516 but the 4 bytes is allocated for the opcode and blocknumber of each data package, the rest is allocated for the data itself.



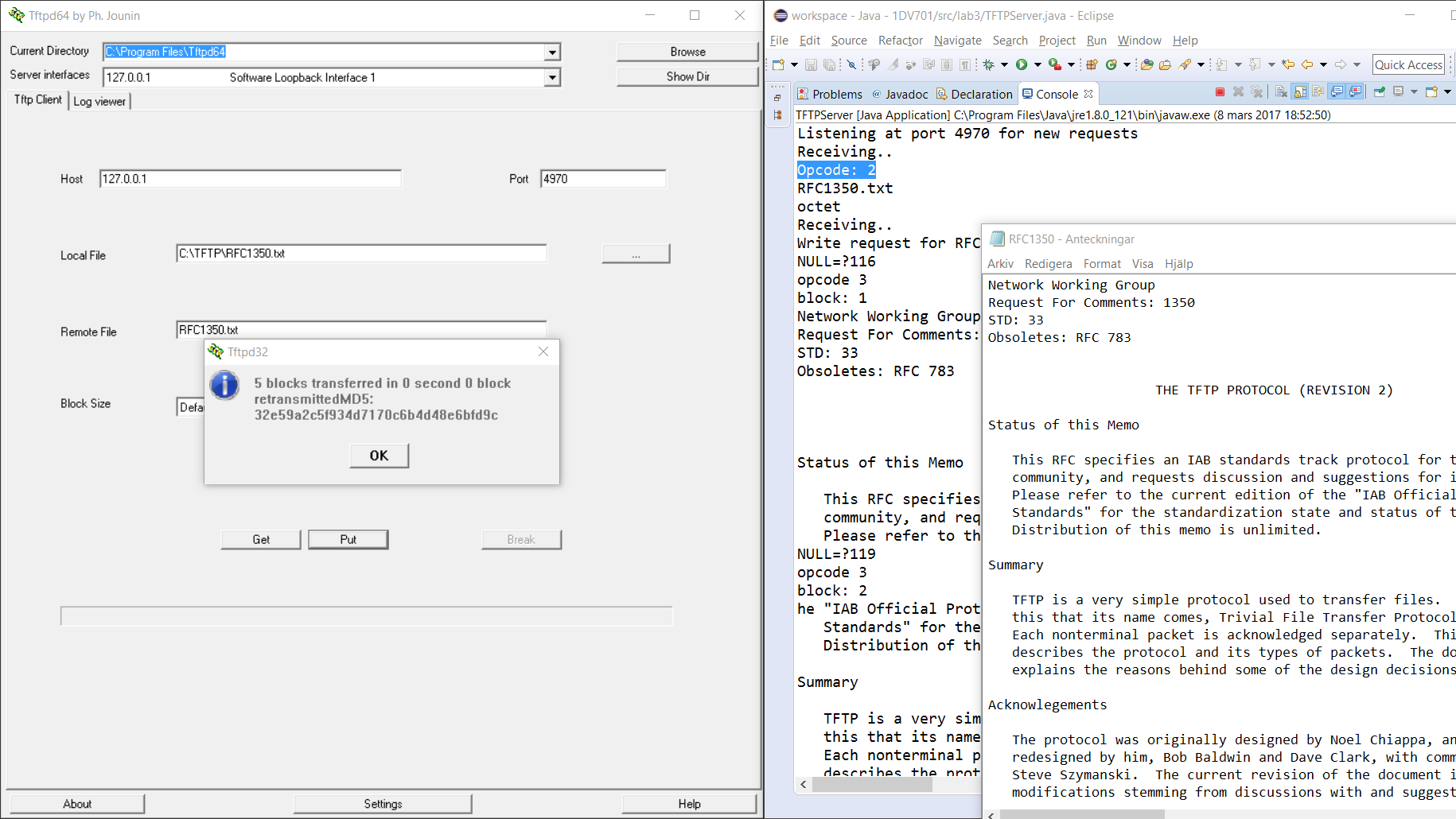
The picture above shows a actual runtime with the block functionality implemented. Each block is 512 bytes of data information, for a block to be successfully transferred to the client we are to get a ACK after each package has been received. If a block is not ACK we are to retransmit the current block, this is done when the timeout has been reached. If a package is retransmitted one to many times we end the communication with a error package to the client. In this particular example it took 5 blocks to send all information and no block had to be retransmitted.



The picture example shown above we have changed the timeout to be 1 millisecond resulting in some retransmission. The blocks are retransmitted but the same block has never been retransmitted twice. In this case the client does not receive a error package.

For the next segment of this task we are to implement a write request response. The write request that are send from the client means that we are allowing them to write data to our server. That means we are creating the file in a local directory that is held by the server.

The picture below explains and illustrate the actual runtime for a WRQ.



As seen above the communication was successful! The server received data for the RFC1350 text file. The opcode is 2 as seen above and the created file is right next to the console in the designated write directory for the server. In case the client tries to write to the same file again there will be a error package saying the file already exist!

## Task 3

In the last task in this laboration we are to implement a error handling for the following errors:

0 Not defined, see error message (if any).

1 File not found.

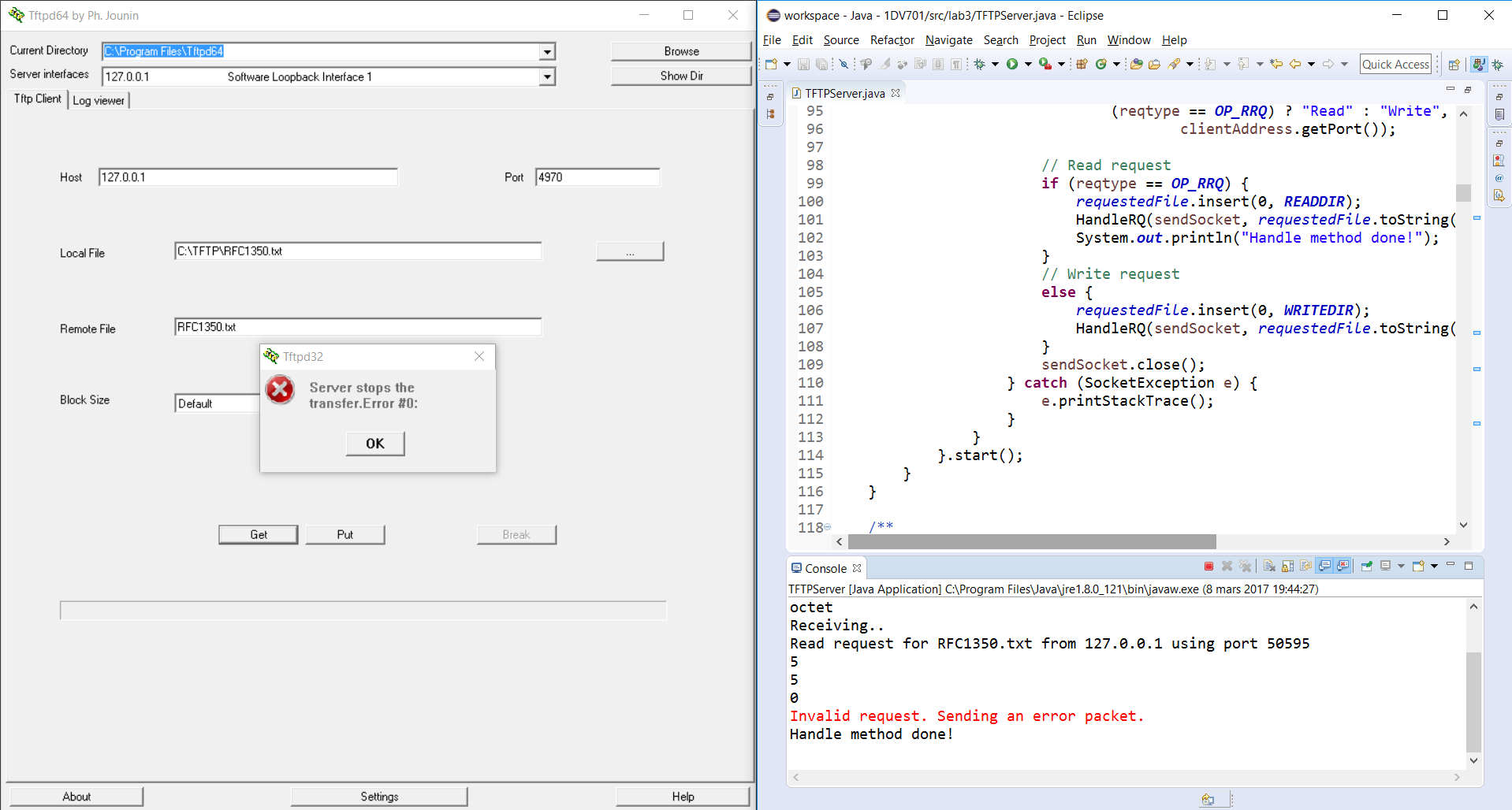
2 Access violation.

6 File already exists.

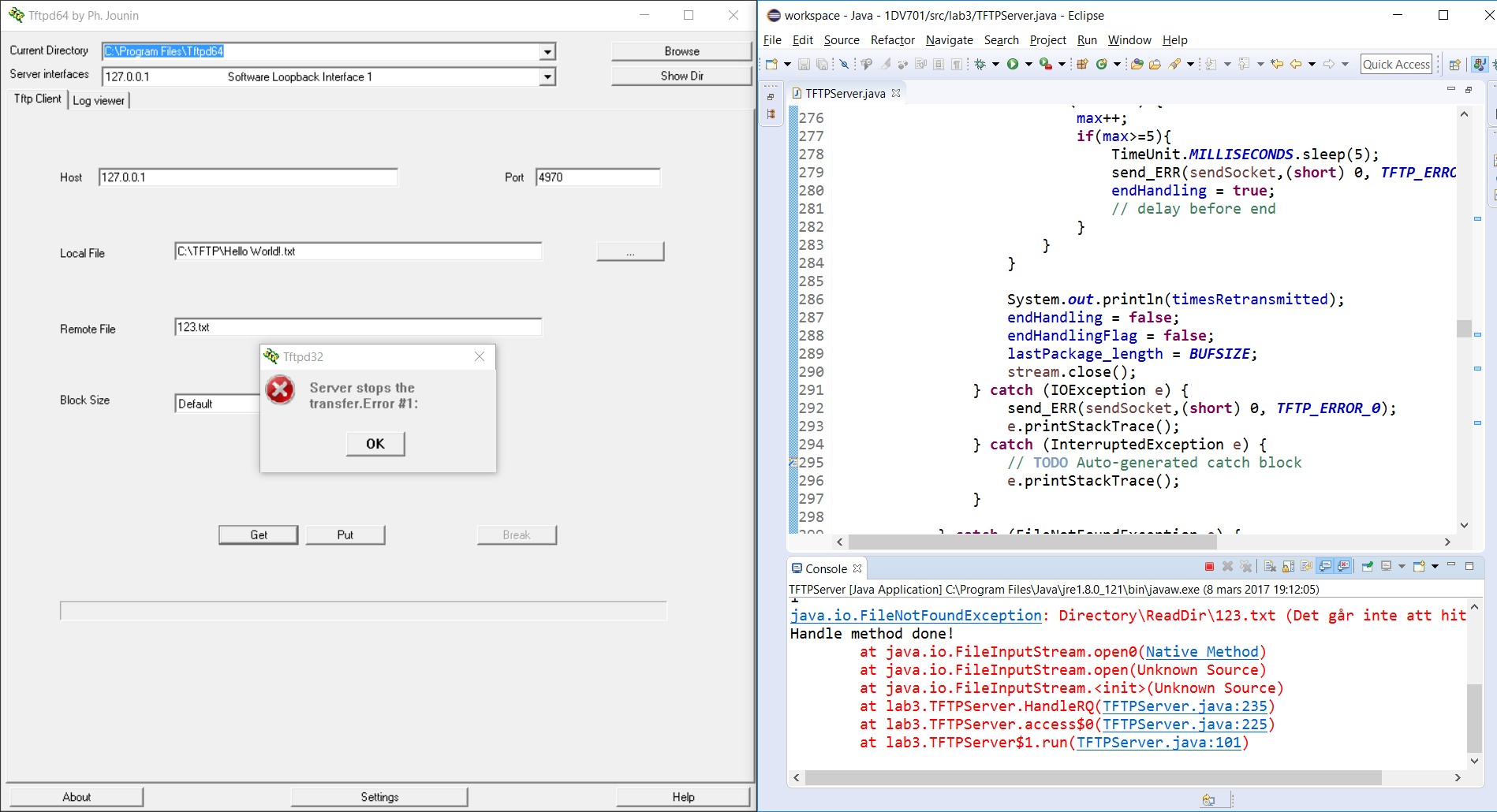
**Solution:**

In general the 2 error will be sent each time anything goes wrong in the RRQ and WRQ that is not error 1 or 6. After each error the server will terminate the connection with the client but there will be a small delay before termination. This is to allow the last package to get retransmitted incase it was lost, the last package being a error package if the termination is not a normal termination that is signified by a data package being smaller than 0-511 bytes.

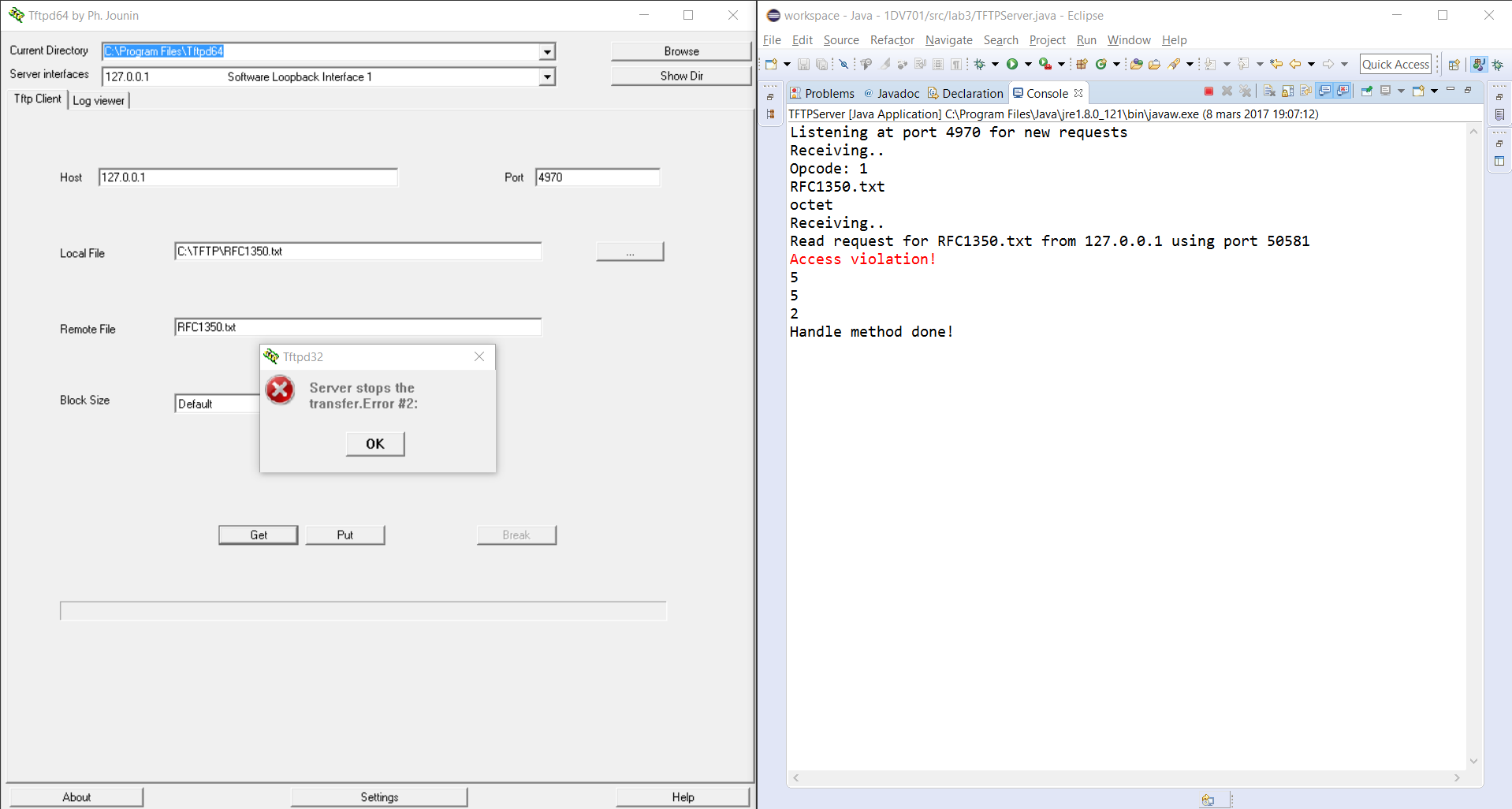
Error 0 from the server is shown below. Error 0 is simulated through giving the server a opcode it can not recognize resulting in a request not recognized. When there is no request for a read or a write we send a error 0 not defined package.



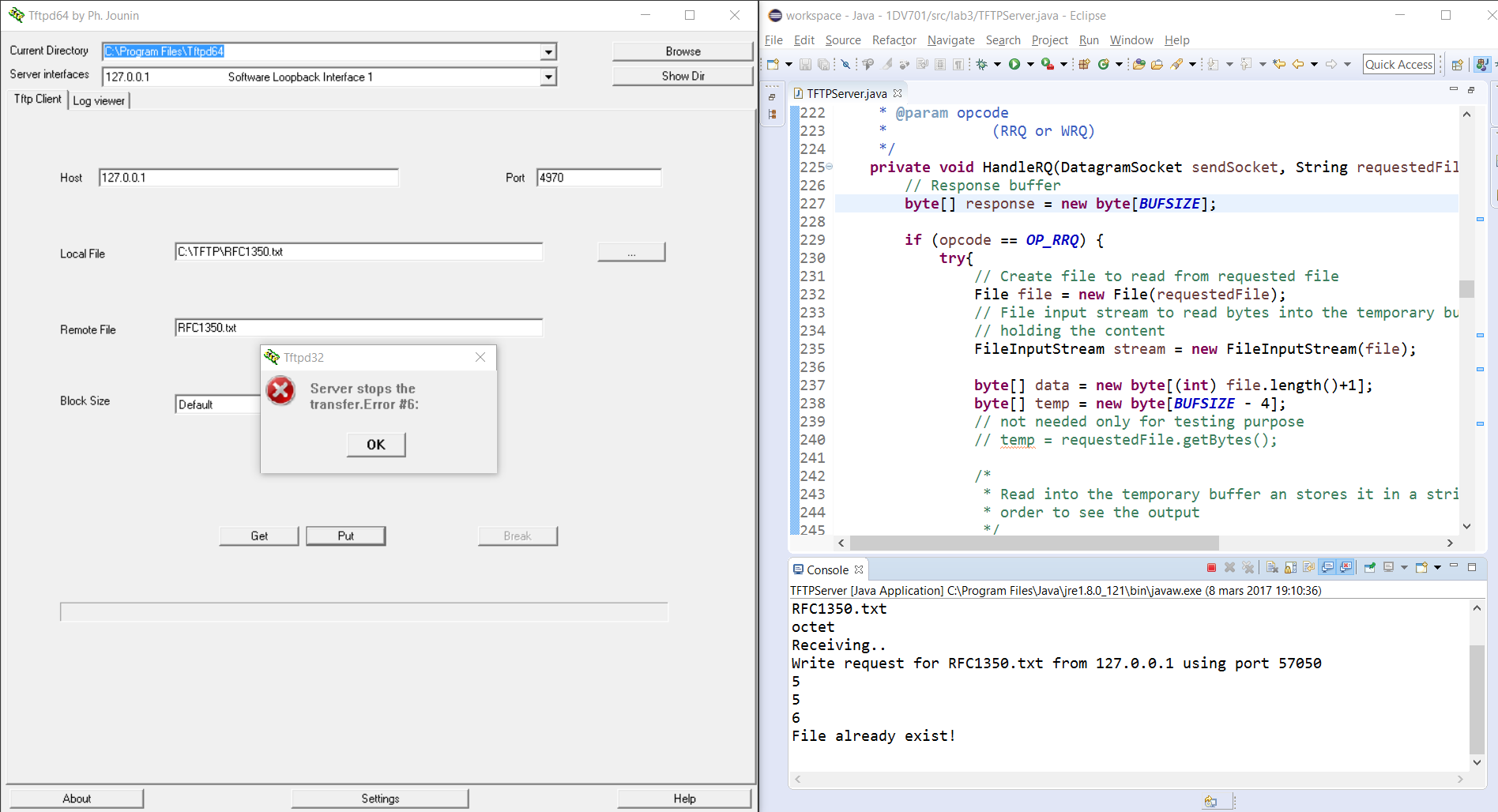
Error 1 from the server is shown below. Error 1 occurs when the client tries to write to a file that does not exist in the server, resulting in a file not found exception.



Error 2 from the server is shown below. In the task it is stated that this error should be created on a generic IOException response.



Error 6 from the server is shown below. This last error occurs when the client wants to write to a file that already exist.



If during a request anything other than the opcode associated with that request is received then the program will treat the opcode as an error an terminate handling of the request.