Linnéuniversitetet
Kalmar Växjö

Report

Assignment 3

1DV701

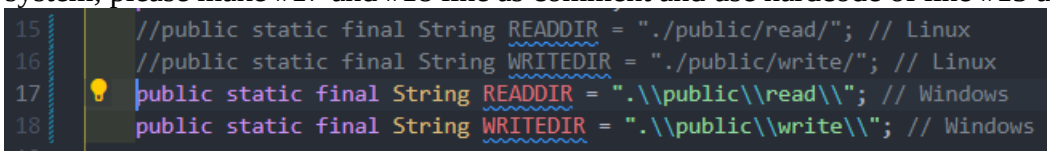
Author: Noah Carlsson,
Yishu Yang
Semester: Spring 2024
Email *nc222hx@student.lnu.se*
yy222cm@student.lnu.se

Contents

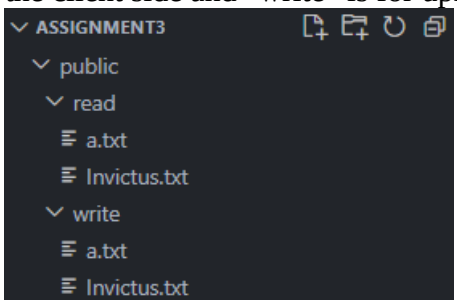
1 Problem 1

1.1 Screenshots and Explanation

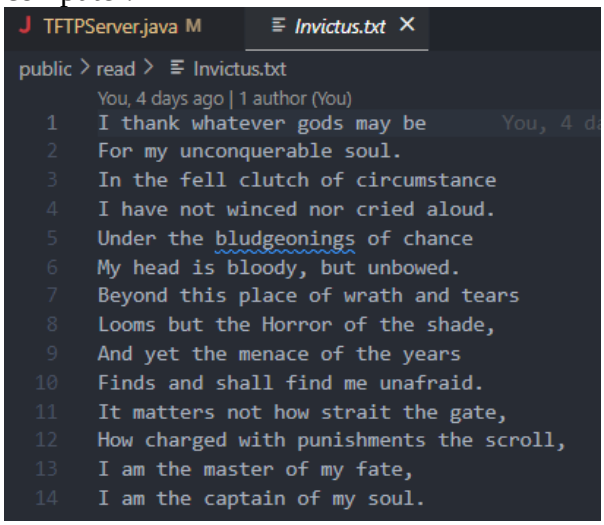
Since I use Windows System to solve this problem, I use the relative path of line #17 and #18 for hardcode in Windows. If you want to verify our work with Mac or Linux system, please make #17 and #18 line as comment and use hardcode of line #15 and #16.



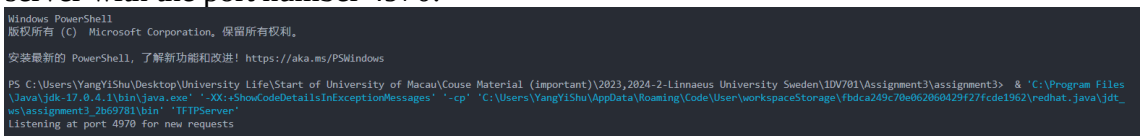
For the relative path connected to the public hierarchy just besides our TFTP server, we have two folders: “read” and “write” under the public folder, which both contains two files: “a.txt” and “Invictus.txt” for use. The folder “read” is for downloading files on to the client side and “write” is for uploading client files onto the server in “write” folder.



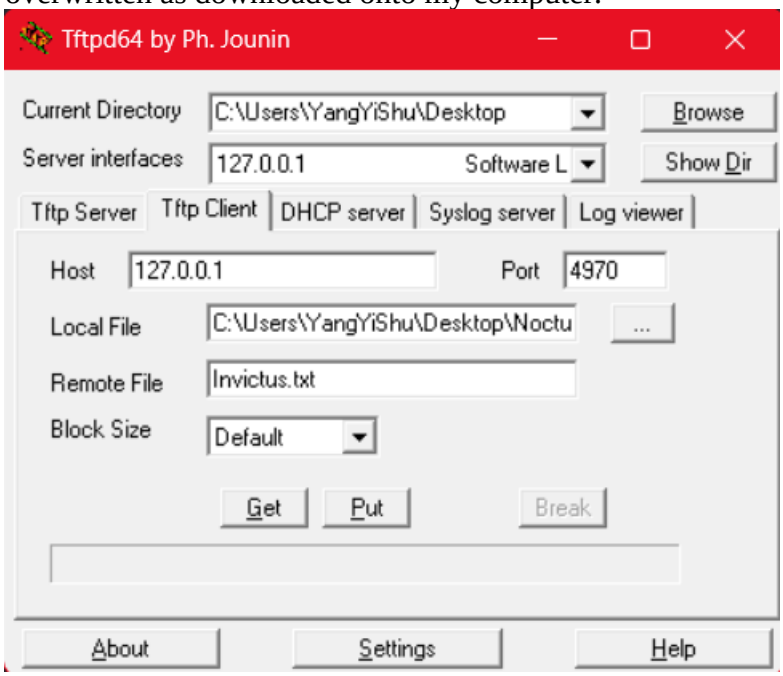
3.A.1 First we do the READ part, where error handling is also demonstrated: Firstly, we show the original file “Invictus.txt” on the server “read” folder, where its content is an English poem. It is the destinated file to be downloaded onto client computer.



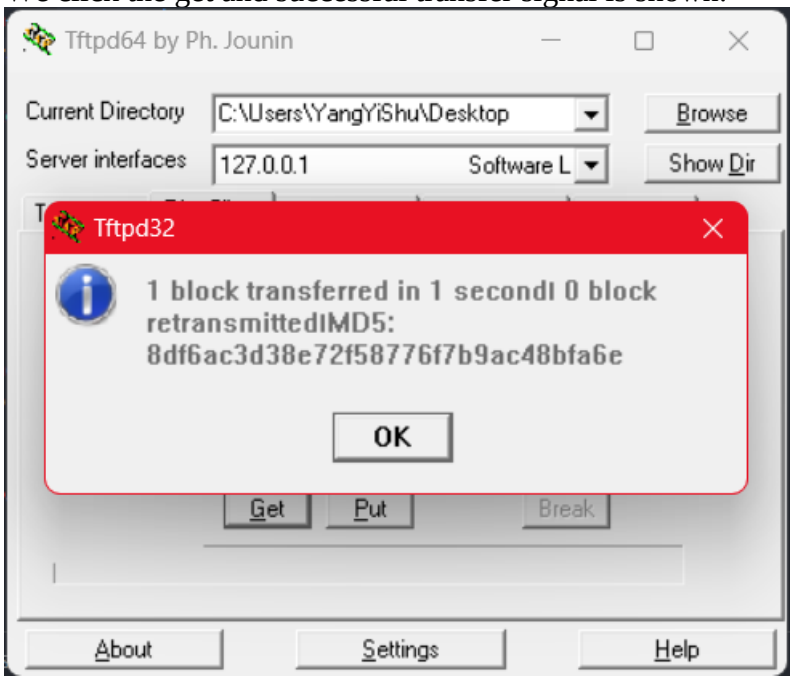
Now we just run the TFTP server code on Visual Studio Code to be connected to the server with the port number 4970:



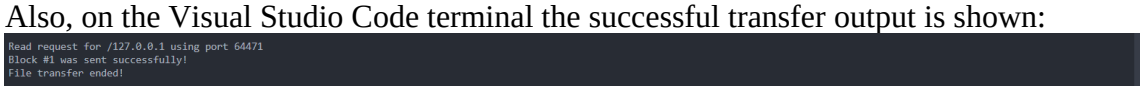
Now we open the designed Tftpd64 application for Windows to handle TFTP stuff. We choose the desktop as the Current Directory and input host as 127.0.0.1 and port number 4970. The remote file is the file we want to read onto client computer, which is “Invictus.txt”. Also we choose the path of the file “Life” on my computer to be overwritten as downloaded onto my computer.



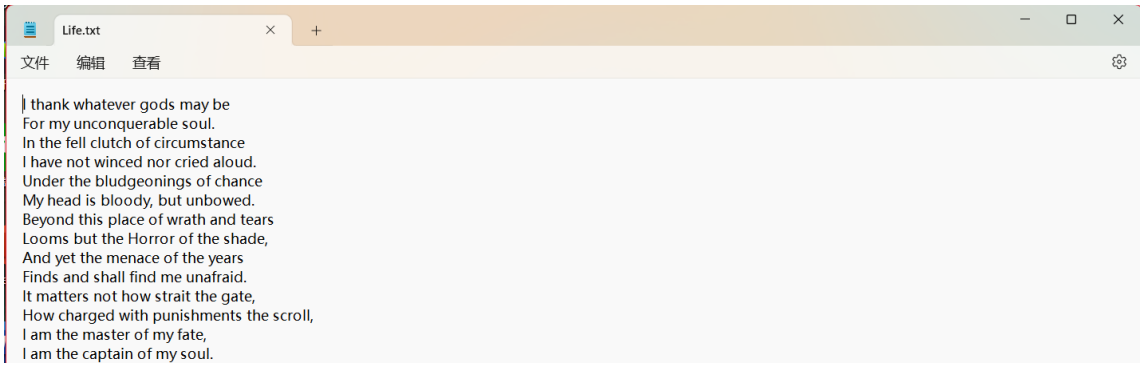
We click the get and successful transfer signal is shown:



Also, on the Visual Studio Code terminal the successful transfer output is shown:

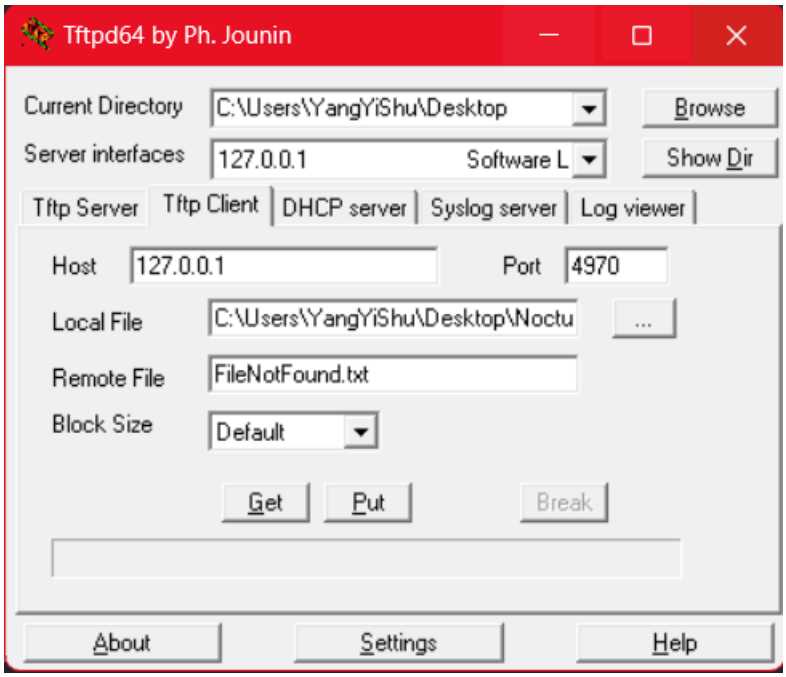


Now I go to my local computer client side, go to the folder “Nocturne” and find the file “Life.txt”, we could see that the content of this file is changed to the content of the read file “Invictus.txt” and this means read is successful.



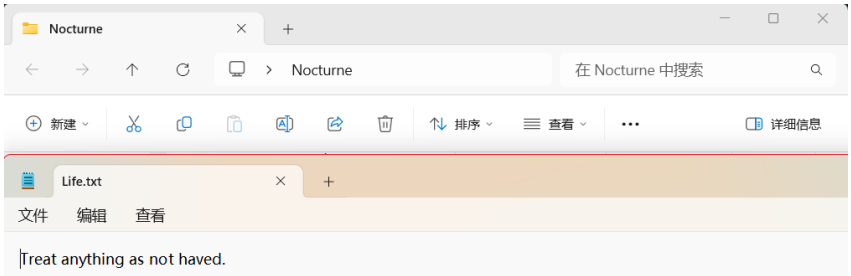
3.A.2 Secondly, we show the error handing in the READ part for file read onto the client side.

We use the above strategy again, reading files from the server onto our computer to overwrite the file “Life.txt” in the folder “Nocturne” again. But this time the remote file on the server doesn’t exist, where we input “FileNotFound.txt” in the remote file, which doesn’t exist on the folder “read”.

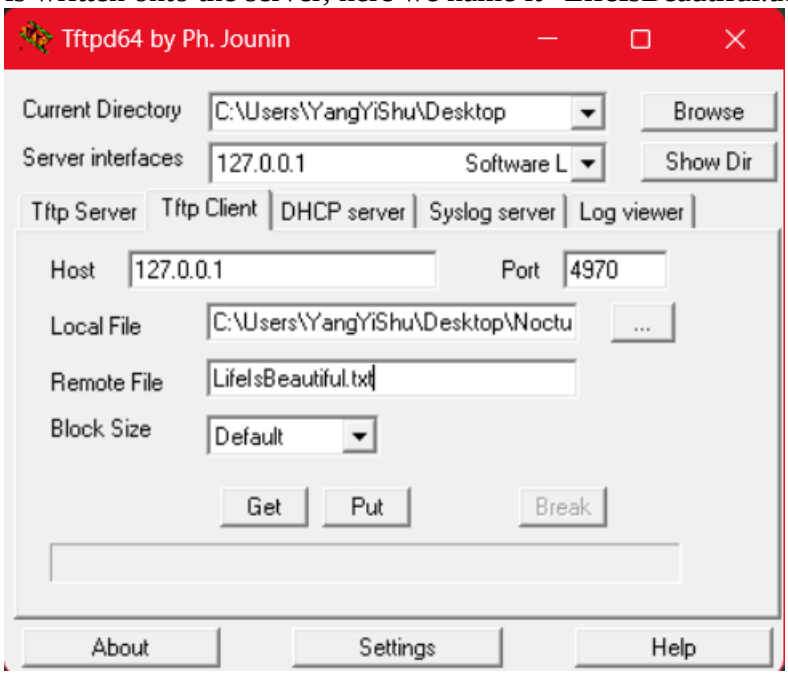


We can see the error output in the Visual Studio Code Terminal, which means that our error handling for the READ part is successful.

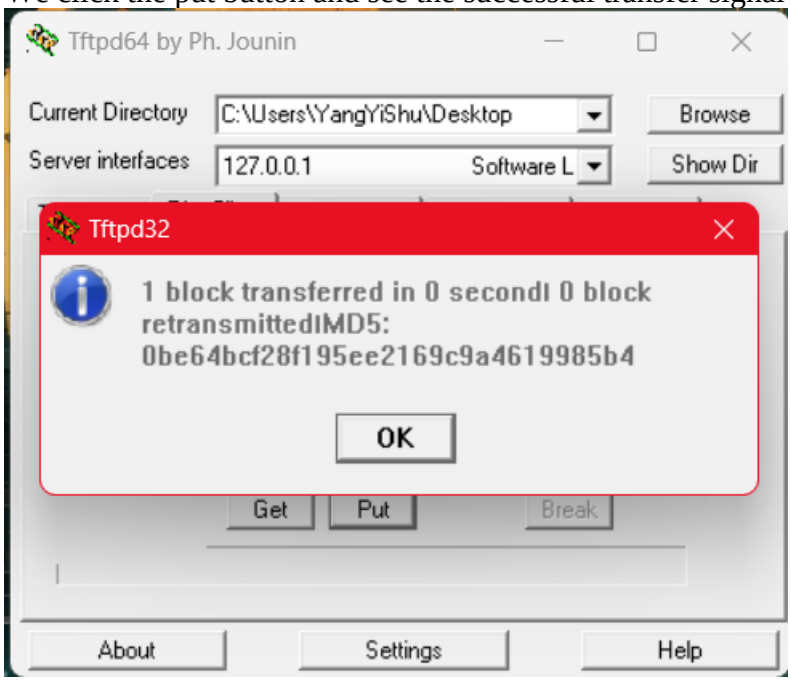
3.B.1 Then we do the WRITE part, where error handling is also demonstrated:
Firstly, we show the original file “Life.txt” which is inside my computer folder “Nocturne”, prepared to be written onto the server inside the “write” from “public” folder.



Now we input the information again. The current directory is still set to be the desktop, whereas the host is still 127.0.0.1 and port number is still 4970. Now the local file here is the file prepared to be written on to the server, so we choose the path of the file “Life.txt”. And the remote file here is the file name we would like the file have when it is written onto the server, here we name it “LifeIsBeautiful.txt”.

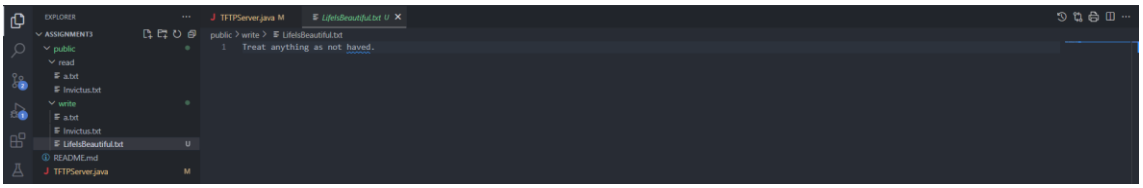


We click the put button and see the successful transfer signal for the transferred block!

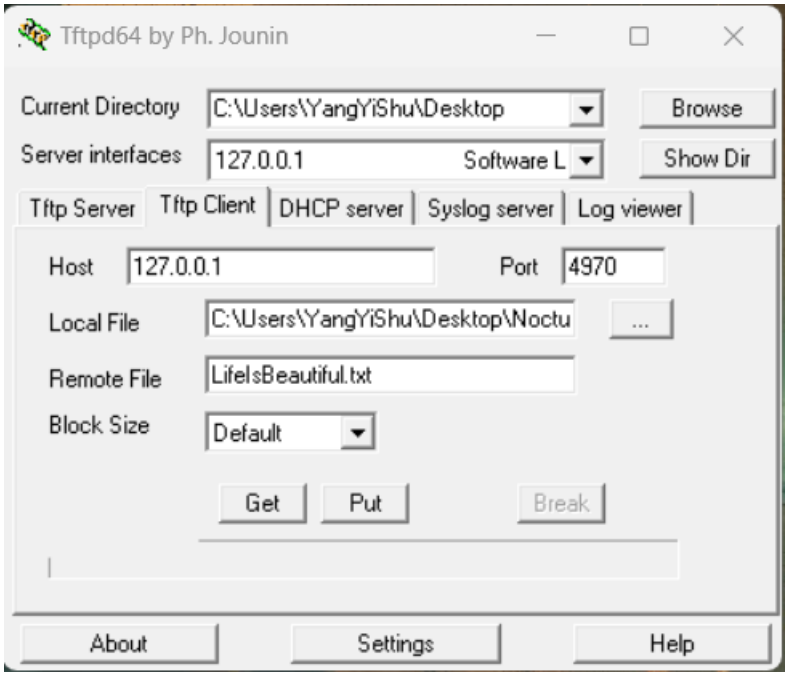


Also on Visual Studio Code, the successful output of the write request is demonstrated, which shows our success.

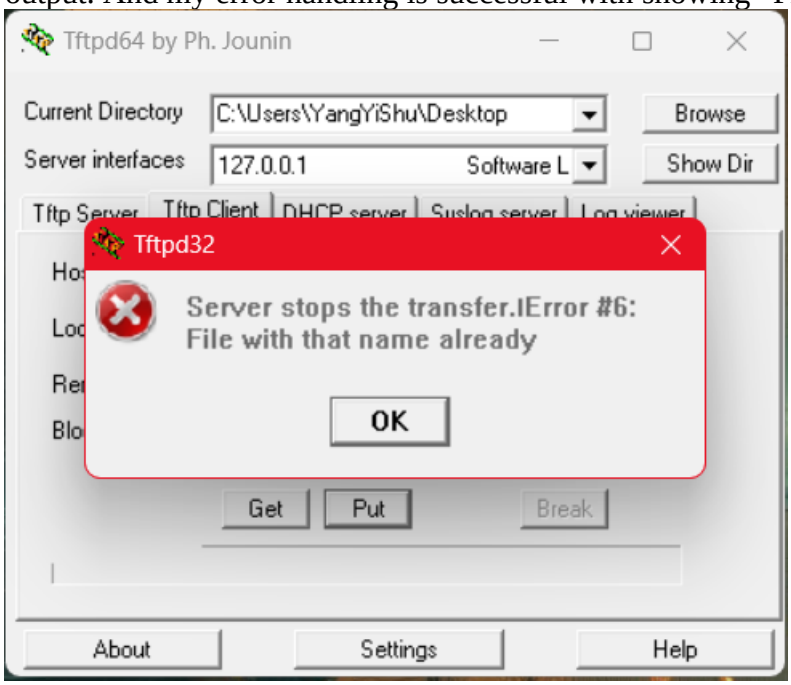
Now we open the “write” folder inside the “public” folder serving as the TFTP server, where it is clearly that the file “LifeIsBeautiful.txt” is here and its content is just the same as “Life.txt” on my computer. This is shown directly on the Visual Studio Code.



3.B.2 Then we show the error handling in the WRITE part:
Since now the “LifeIsBeautiful.txt” file is on the server inside the folder “write”, we do the WRITE part again, with all same information fulfilled and the remote file to upload is still “LifeIsBeautiful.txt” with same content in “Life.txt” on my computer “Nocturne” folder.



Click the Put button again, the error handing is shown on the exe with #Block 6 error output. And my error handling is successful with showing “File with that name already”.



Here we have one notation:
Since the difference between Linux and Windows, the error output here is not complete in terms of viewing from Linux. Actually, the error #6 here should be “File with that name already exists.” But it is not a big issue, since error message number “#6” is shown.

```
private void writeRequest(DatagramSocket sendSocket, File file) {
    if (file.exists()) { // Error Validation
        send_ERR(sendSocket, (short) 6, errorMessage: "File with that name already exists.");
        return; // Error Code 6 = "File exists"
    }
```

3.C Here I explain why there are two sockets (“socket” and “sendSocket”) in the starter code:
“socket” is the socket to be heard from, where it is connected to a local bind point to wait and listening for the request on the specific port number, so it cannot both hear the request and deal with the request. “socket” is always hearing the request on the local bind point. Whereas “sendSocket” is always in a loop to handle client requests with both Read and Write requests. So actually, they have different functions and collaborate to deal with the request package.

1.2 Discussion

Our major task is to use Java to construct a TFTP server, which can be accessed by implementing a standard TFTP client. In Problem 1 the TFTP server should be specifically implemented according to RFC1350, where only “octet” mode should be supported.

Now we will break down the process of implementation of the server into the following several steps due to its sophistication:

1. Since the TFTP Server starter code has been provided, we could look at it in detail, where those green comments with required codes to fulfil should be focused on because it helps us to understand what codes should be provided in what location. Combining TFTP specification with this, we could fully understand what we should do regarding coding, where testing script and content also help.

2. Now we need to adapt our start code to the single-read request, which also has several steps:

    A. Construct the action of listening to the specific port: Use the **“receiveFrom()”** method.

    B. Parse one single read request: Use the **“ParseRQ()”** method. ***Notation:*** The first two bytes from the message demonstrate the type of the request with opcode.

    C. The requested file should be opened now.

D. A packet regarding the response should now be created. And provide the opcode for data using the variable name: **"OP_DATA"** and a block number **"1"**.
Notation: Write unsigned shorts with the **"putShort()"** and **"getShort()"** methods when writing unsigned shorts or reading unsigned shorts is needed.

E. Only 512 bytes at maximum could be read from the file, which should accompanied by the packet and sent to the client.

F. Acknowledgement of the first sent package (ACK) should be received from the client if we successfully implemented all.

3. Several processes to check regarding the TFTP Server and provide screenshots with explanation: (This part will be shown above in section **1.1 Screenshots and Explanation**)

   A. Once everything above has been successfully settled, a **READ** request from the client should be made, where the size of the reading file should be smaller than 512 bytes. After this, every response and demonstration should be ensured to be correct by providing screenshots.

   B. Apart from reading files, the request to **WRITE** a file shorter than 512 bytes should also be checked to ensure the output is correct.

   C. Once the request has been successfully **READ**, check the TFTP server again and explain why two client sockets are used including **"socket"** and **"sendSocket"**.

4. We explain how we do this problem and find what troubles this problem below:

   1. When someone wants to perform a RRQ or WRQ they send a packet to the server containing what they want to do. The server then parses this packet to understand what it should do. The first 2 bytes are the OPCODE (RRQ/WRQ/etc.). This first request contains the file name in question and is therefore also parsed. The mode is also in this packet afterwards and the server ensures that it is "octet" (byte mode / 8 binary numbers).

   2. If the request is an RRQ, the server then starts sending back packets with the DATA OPCODE (3), an increment of the block number per packet, and the actual data in packets of size 516 (512 data + 4 bytes for the opcode and block number). For each data packet sent, the recipient will return an ACK packet with the ACK opcode (4) and the block number to ensure that it reaches them. If a different block number or opcode is returned then we know something is wrong.

   3. If the request is a WRQ, the server sends back ACK packets just like the recipient did in the RRQ. We send ACK packets with the block number that we received. First ACK will contain block number 0 since only the request packet

has so far been received. The server keeps sending ACK packets for each data package received.

Important to know: All packets are 516 bytes. The first 2 bytes are the opcode, the next 2 is the block number and the rest is the data. If a packet less than 516 is received then you know it's the last one.

5. Some tricky parts when we handle this problem:

1. At first we use the absolute path like "/home/noah/read/" to locate our READDIR (reader) and WRITEDIR (writer), which means others should manually change the code and input the path address when they should verify the correctness of our server in their computers. But now we implemented a relative two-hierarchy folder with the main folder called “public” near our server Java file, which has two folders inside one called “read” and another one called “write” where files are saved inside. The path in the code is also converted to be relative.

2. Since on Linus tftp client is pre-installed but on Windows tftp client should be downloaded, we use different ways to test the code on different systems. And in our server hardcode is used, so it involves the difference in recognition of “/” and “\\” in Linux and Windows. Now we apply specific codes to handle this issue to make it fluently implemented on both two systems.