ASSIGNMENT 3

Computer Network : Introdcution

Zacky Kharboutli
Zk222ay

Charles Mairey
Cm222pd

Contents

1 Introduction

1.1 Problem overview

This is the report of the third assignment of the course Computer Networks. The goal of this assignment is to create a TFTP server with java which can handle TFTP request coming from an application like TFTPD on windows or from the terminal.

- The first problem is to get the provided code working to implement the read requests by filling the given methods.
- The second problem is to allow the transfer of file bigger than 512 bytes, to add a timeout with retransmissions functionality and to implement the write requests.
- The last problem is dedicated to implementing the different error codes of the TFTP.

The report will address the way the problems were solved and the some pictures were add to show the results.

1.2 Responsibilities

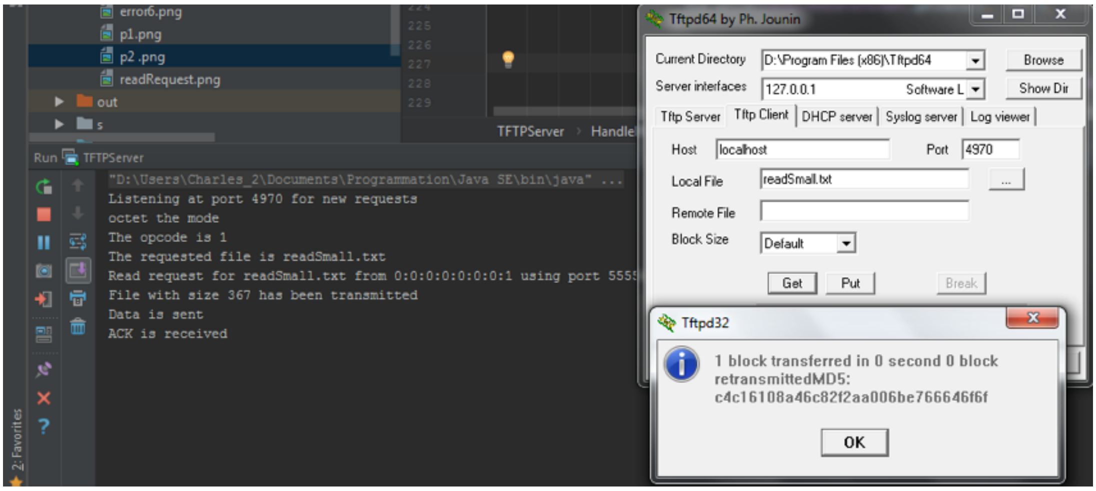
Charles Mairey: Implemented the write request and VG-Task 1 with Wireshark traffic analysis of read and write requests and wrote the report. (50%)

Zacky Kharboutli: Implemented the first problem(read request) finished what was needed for read request in the second problem and implemented the error responses.(50%)

2 Problem One :

We started by implementing the Read request by filling the give method in the provided code.
When the program is launched, it starts listening on port 4970 for any given request. When an incoming request is caught, its address is taken with the receiveFrom method. We then parse the request in the parseRQ method. This method get the mode of transfer requested (octet is the only mode implemented), the type of request thanks to the opcode which is the first 2 bytes of the request packet (00 01 means "read", 00 02 means "write"), and the name of the file requested.
Then if the opcode is 00 01, the file name is put after the name of the directory in which the requested file is supposed to be located. The handleRQ method is called, which, after checking it exists, will prepare the file for transmission, checking its size is smaller than 512 bytes. (Files bigger than 512 bytes will be handle in problem 2. The method send-DATA-receive-ACK will then send the file data within a packet containing also the opcode 00 03 meaning Data packet and the block number.

Why socket and send socket?
Send socket is used in the thread that created for the client connection in the original socket on port 4970. Send socket looks for free port after receiving a read/write request and inform the client about this port as it is going to be used in the connection. In other words, the server tells the client which port by including it in the first data packet or the ACK packet (according to the TFTP file). Using socket and send socket helps us to handle multiple request from multiple clients at once as the socket is always free once the request is handled in a separate thread.



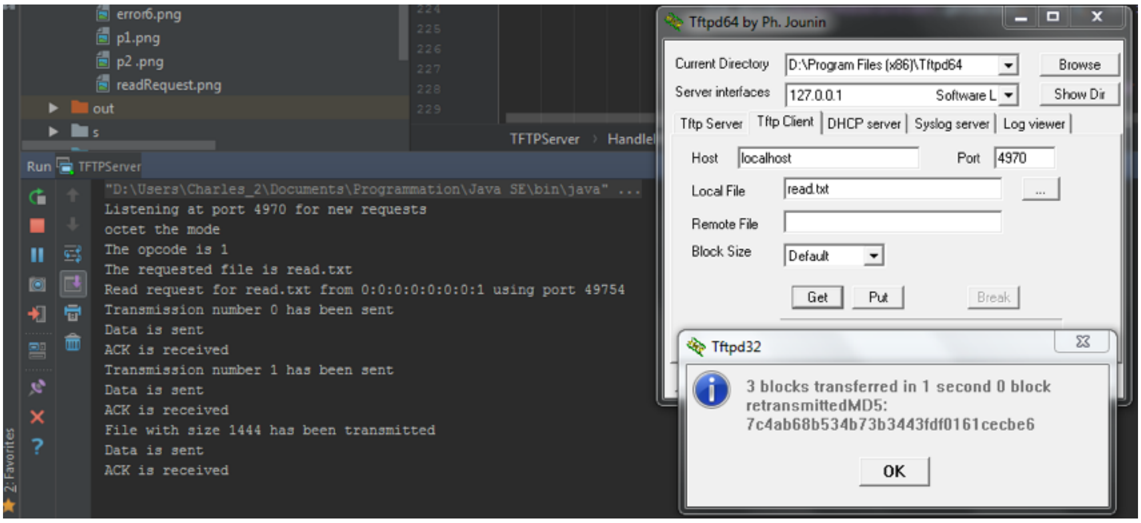
```
"D:\Users\Charles_2\Documents\Programmation\Java SE\bin\java" ...
Listening at port 4970 for new requests
octet the mode
The opcode is 1
The requested file is readSmall.txt
Read request for readSmall.txt from 0:0:0:0:0:0:0:1 using port 5555
File with size 367 has been transmitted
Data is sent
ACK is received
```

Current Directory: D:\Program Files (x86)\Tftpd64
Server interfaces: 127.0.0.1 Software L
Host: localhost
Port: 4970
Local File: readSmall.txt
Remote File:
Block Size: Default

1 block transferred in 0 second 0 block retransmittedMD5: c4c16108a46c82f2aa006be766646f6f

3 Problem Two

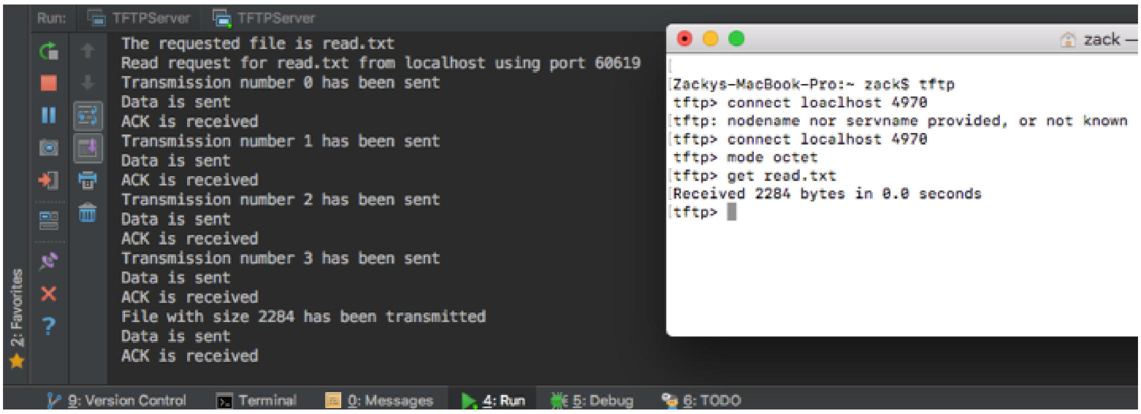
We need to allow the transfer of files bigger than 512 bytes. In the method HandleRQ, we divided the file size by 512 to know how many packets have to be sent. We then take the first 512 bytes of the file and use it as parameter for the send-DATA-receive-ACK method, then the next 512 bytes after receiving the ACK for the previous block of data.
We do this until what is left to be sent is smaller than 512 bytes. The data packet received by the Client will be smaller than 512 and it will deduce that it's the last data packet of the file. It will send an ACK packet for this block and the transfer is finished.
A timeout has been implemented in case there is a problem during the transfer of one data packet or one ACK (an ACK packet never arrives to destination for example), the data packet will be retransmitted. Same if the ACK is received but acknowledge the wrong block of data. 5 re-transmissions maximum are allowed before termination of the transfer.



```
"D:\Users\Charles_2\Documents\Programmation\Java SE\bin\java" ...
Listening at port 4970 for new requests
octet the mode
The opcode is 1
The requested file is read.txt
Read request for read.txt from 0:0:0:0:0:0:0:1 using port 49754
Transmission number 0 has been sent
Data is sent
ACK is received
Transmission number 1 has been sent
Data is sent
ACK is received
File with size 1444 has been transmitted
Data is sent
ACK is received
```

Current Directory: D:\Program Files (x86)\Tftpd64
Server interfaces: 127.0.0.1 Software L
Host: localhost
Port: 4970
Local File: read.txt
Remote File:
Block Size: Default

3 blocks transferred in 1 second 0 block retransmittedMD5: 7c4ab68b534b73b3443fdf0161cecbe6

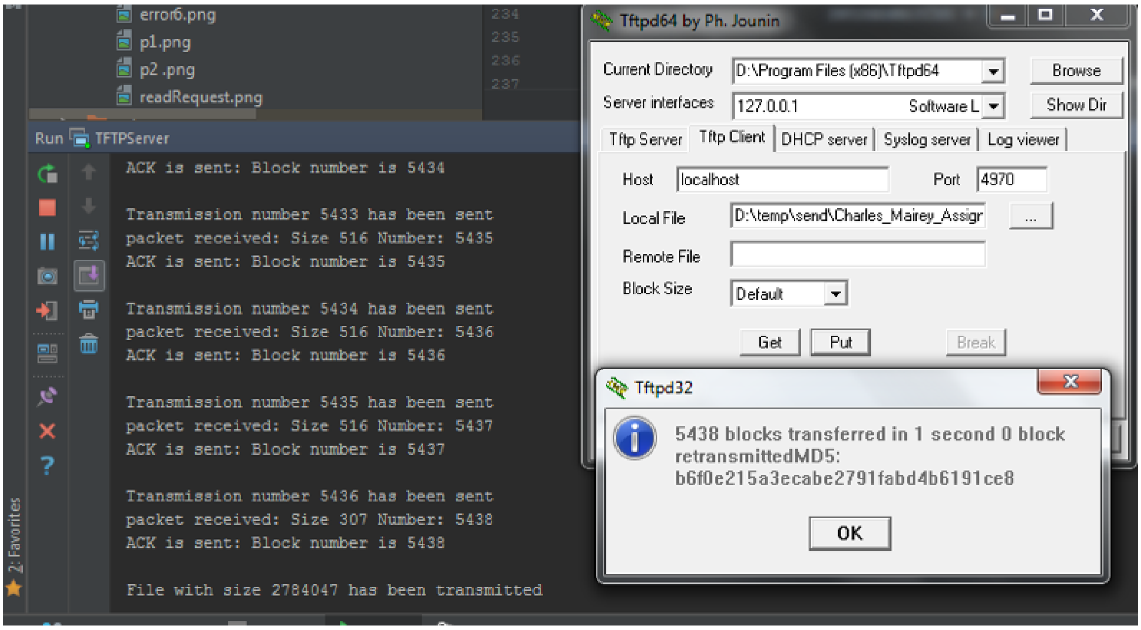


```
The requested file is read.txt
Read request for read.txt from localhost using port 60619
Transmission number 0 has been sent
Data is sent
ACK is received
Transmission number 1 has been sent
Data is sent
ACK is received
Transmission number 2 has been sent
Data is sent
ACK is received
Transmission number 3 has been sent
Data is sent
ACK is received
File with size 2284 has been transmitted
Data is sent
ACK is received
```

```
Zackys-MacBook-Pro:~ zack$ tftp
tftp> connect loaclhost 4970
tftp: nodename nor servname provided, or not known
tftp> connect localhost 4970
tftp> mode octet
tftp> get read.txt
Received 2284 bytes in 0.0 seconds
tftp>
```

We then implemented the Write request functionnality. The request sent by the client is parsed in ParseRQ and if the opcode read is 00 02, it is a write request. The HandleRQ then start by sending the ACK packet to acknowledge the reception of the request, just then the client will start sending the file. The method receive-DATA-send-ACK then receive the next packet. The opcode is checked to be equal to 00 03 meaning a Data packet and we also check that the block number is in the continuation of the last packet

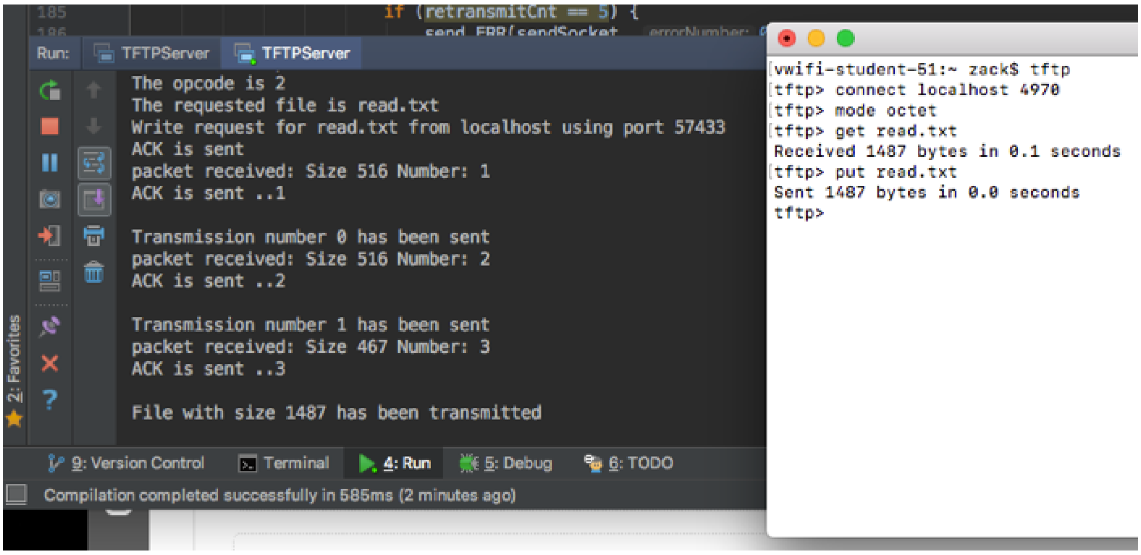
received. Only then, the ACK packet containing this block number will be sent to the client.
HandleRQ then create the new file according to the file name obtained in the request packet. The data received is written at the end of the file. When the data packet received is smaller than 512 bytes, it means it is the last packet and that no more data is expected. This data is writtent to the end of the file and the FileOutputStream used to write to the file is closed. The last ACK of the last block is sent to the client and the transfer is finished.
If no ACK is received after sending a data packet, it will be re-transmitted again a maximum of 5 times.



```
ACK is sent: Block number is 5434

Transmission number 5433 has been sent
packet received: Size 516 Number: 5435
ACK is sent: Block number is 5435

Transmission number 5434 has been sent
packet received: Size 516 Number: 5436
ACK is sent: Block number is 5436

Transmission number 5435 has been sent
packet received: Size 516 Number: 5437
ACK is sent: Block number is 5437

Transmission number 5436 has been sent
packet received: Size 307 Number: 5438
ACK is sent: Block number is 5438

File with size 2784047 has been transmitted
```

Current Directory: D:\Program Files (x86)\Tftpd64
Server interfaces: 127.0.0.1 Software L
Host: localhost
Port: 4970
Local File: D:\temp\send\Charles_Mairey_Assigr
Remote File:
Block Size: Default

5438 blocks transferred in 1 second 0 block retransmittedMD5: b6f0e215a3ecabe2791fabd4b6191ce8



```
if (retransmitCnt == 5) {
    send_ERR(sendSocket, errorNumber:
```

```
The opcode is 2
The requested file is read.txt
Write request for read.txt from localhost using port 57433
ACK is sent
packet received: Size 516 Number: 1
ACK is sent ..1

Transmission number 0 has been sent
packet received: Size 516 Number: 2
ACK is sent ..2

Transmission number 1 has been sent
packet received: Size 467 Number: 3
ACK is sent ..3

File with size 1487 has been transmitted
```

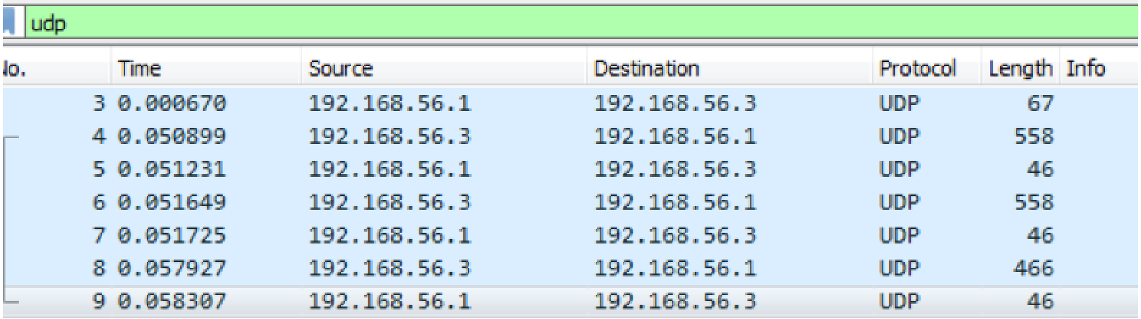
Compilation completed successfully in 585ms (2 minutes ago)

```
vwifi-student-51:~ zack$ tftp
tftp> connect localhost 4970
tftp> mode octet
tftp> get read.txt
Received 1487 bytes in 0.1 seconds
tftp> put read.txt
Sent 1487 bytes in 0.0 seconds
tftp>
```

3.1 VG Task 1:

Wireshark can't analyse the local interface to see the packets flowing to and from the server. So we used a virtual machine running Ubuntu 15.04 on Virtual Box. The TFTP

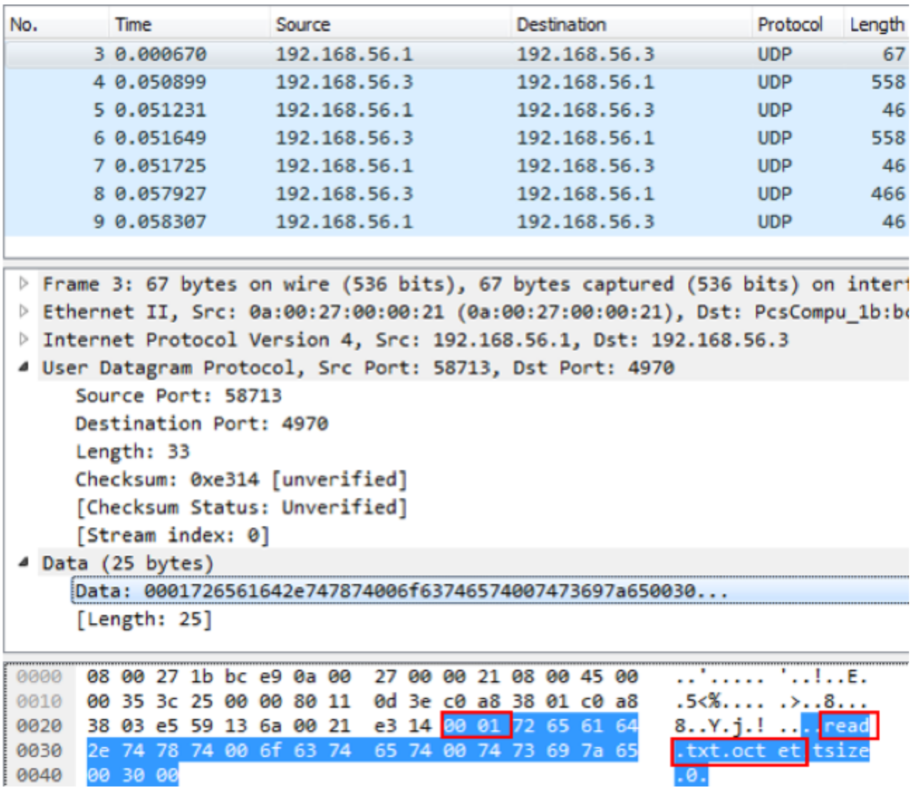
server was running on that virtual environment. On the host (windows 7) we were sending read and put request with the tftpd software. Wireshark was then capturing the packets transiting through the Virtual Box network interface.
We created a .txt file of size of size 1444 octet. One TFTP data packet's size being up to 512, a total of 3 data packets should logically be sent by the server to the client. The file is put on the server directories and we do a GET request from tftpd on windows, this the result on Wireshark.



udp

lo.	Time	Source	Destination	Protocol	Length	Info
3	0.000670	192.168.56.1	192.168.56.3	UDP	67	
4	0.050899	192.168.56.3	192.168.56.1	UDP	558	
5	0.051231	192.168.56.1	192.168.56.3	UDP	46	
6	0.051649	192.168.56.3	192.168.56.1	UDP	558	
7	0.051725	192.168.56.1	192.168.56.3	UDP	46	
8	0.057927	192.168.56.3	192.168.56.1	UDP	466	
9	0.058307	192.168.56.1	192.168.56.3	UDP	46	

In the Length column, we clearly see the alternation between the data packets and the ACK packets due to their size being quite different. We also see on the source and destination columns that it a form of discussion between both entities, one sending a packet, then the other sending one too.

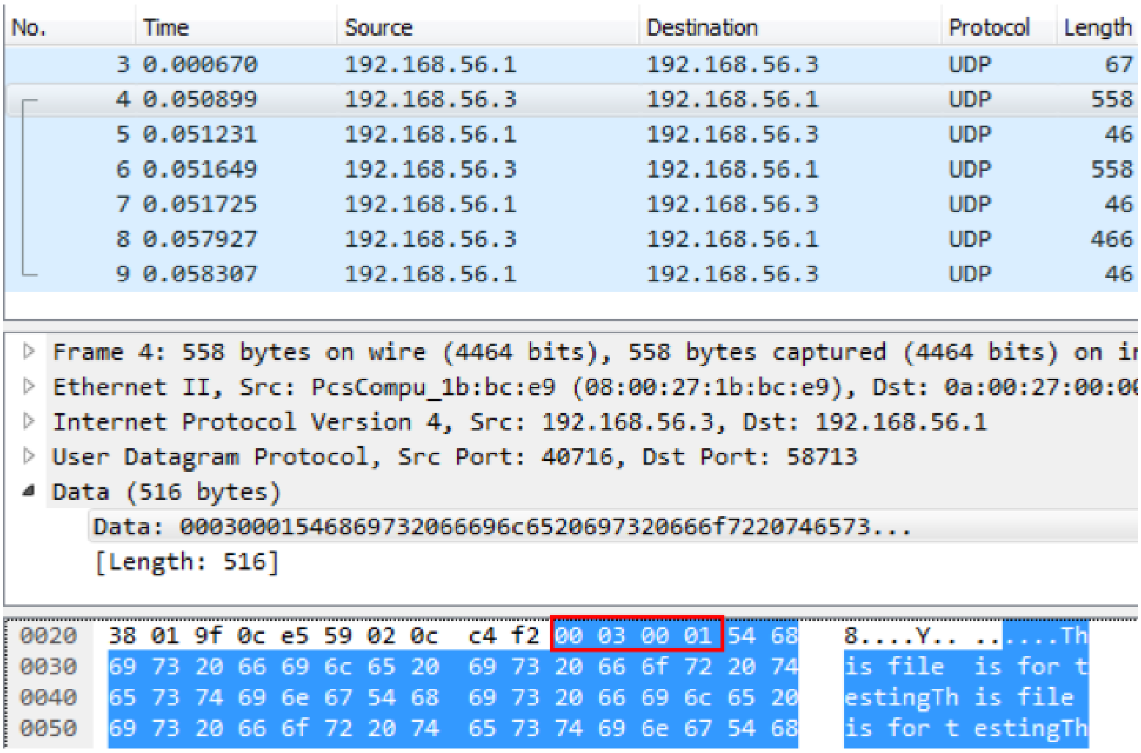


No.	Time	Source	Destination	Protocol	Length
3	0.000670	192.168.56.1	192.168.56.3	UDP	67
4	0.050899	192.168.56.3	192.168.56.1	UDP	558
5	0.051231	192.168.56.1	192.168.56.3	UDP	46
6	0.051649	192.168.56.3	192.168.56.1	UDP	558
7	0.051725	192.168.56.1	192.168.56.3	UDP	46
8	0.057927	192.168.56.3	192.168.56.1	UDP	466
9	0.058307	192.168.56.1	192.168.56.3	UDP	46

```
▷ Frame 3: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on inter
▷ Ethernet II, Src: 0a:00:27:00:00:21 (0a:00:27:00:00:21), Dst: PcsCompu_1b:b
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.3
◢ User Datagram Protocol, Src Port: 58713, Dst Port: 4970
    Source Port: 58713
    Destination Port: 4970
    Length: 33
    Checksum: 0xe314 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
◢ Data (25 bytes)
    Data: 0001726561642e747874006f63746574007473697a650030...
    [Length: 25]
```

```
0000  08 00 27 1b bc e9 0a 00  27 00 00 21 08 00 45 00   ..'..... '..!..E.
0010  00 35 3c 25 00 00 80 11  0d 3e c0 a8 38 01 c0 a8   .5<%.... .>..8...
0020  38 03 e5 59 13 6a 00 21  e3 14 00 01 72 65 61 64   8..Y.j.! ....read
0030  2e 74 78 74 00 6f 63 74  65 74 00 74 73 69 7a 65   .txt.oct et.tsize
0040  00 30 00                                           .0.
```

The first packet is sent by the client to the server and is 67 bytes large, from which 25 are data. Which means 45 bytes of headers from UDP and IPv4 headers. The 2 first bytes from the data are as expected the opcode indicating to the server if the request is read or write. Here the opcode is 00 01 which represent a read request. A write request would have an opcode of 00 02.
Furthermore we can see that the request text is available in clear in the packet. Thus we can see that the opcode is followed by the name of the requested file (read.txt) and the mode of transfer (octet) as it should be in TFTP transfers for the first packet.

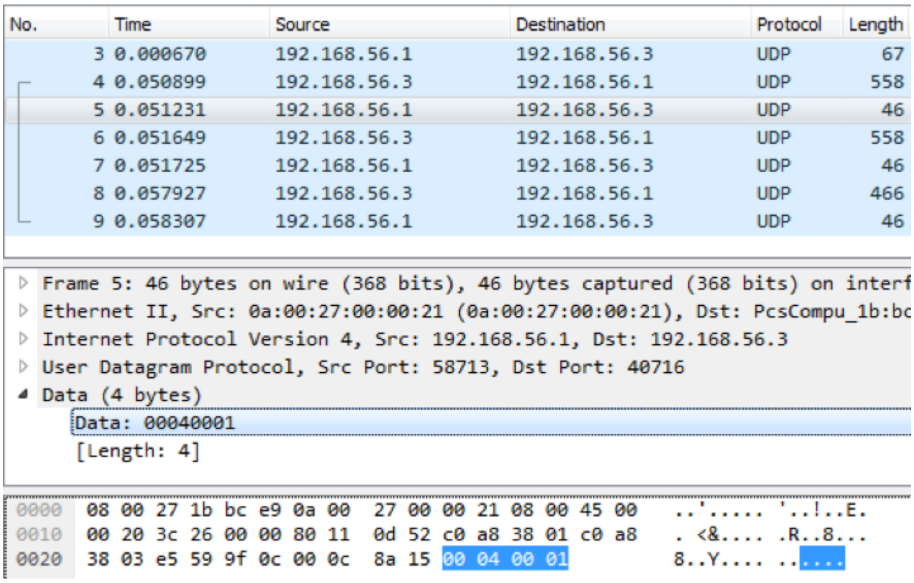


No.	Time	Source	Destination	Protocol	Length
3	0.000670	192.168.56.1	192.168.56.3	UDP	67
4	0.050899	192.168.56.3	192.168.56.1	UDP	558
5	0.051231	192.168.56.1	192.168.56.3	UDP	46
6	0.051649	192.168.56.3	192.168.56.1	UDP	558
7	0.051725	192.168.56.1	192.168.56.3	UDP	46
8	0.057927	192.168.56.3	192.168.56.1	UDP	466
9	0.058307	192.168.56.1	192.168.56.3	UDP	46

```
▷ Frame 4: 558 bytes on wire (4464 bits), 558 bytes captured (4464 bits) on i
▷ Ethernet II, Src: PcsCompu_1b:bc:e9 (08:00:27:1b:bc:e9), Dst: 0a:00:27:00:0
▷ Internet Protocol Version 4, Src: 192.168.56.3, Dst: 192.168.56.1
▷ User Datagram Protocol, Src Port: 40716, Dst Port: 58713
◢ Data (516 bytes)
      Data: 000300015468697320666696c6520697320666f7220746573...
      [Length: 516]
```

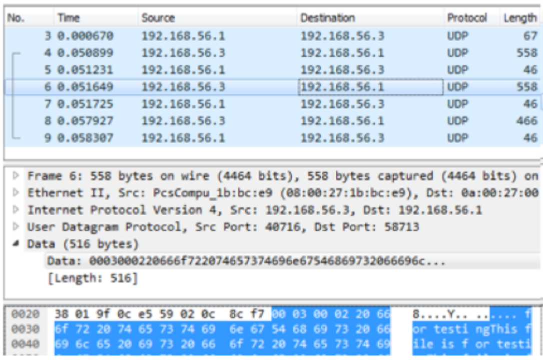
```
0020  38 01 9f 0c e5 59 02 0c  c4 f2 00 03 00 01 54 68   8....Y.. ......Th
0030  69 73 20 66 69 6c 65 20  69 73 20 66 6f 72 20 74   is file  is for t
0040  65 73 74 69 6e 67 54 68  69 73 20 66 69 6c 65 20   estingTh is file 
0050  69 73 20 66 6f 72 20 74  65 73 74 69 6e 67 54 68   is for t estingTh
```

The second packet is the server responding to the client in function of the request. Here it was a read request, so the packet following the request includes the first 512 bytes of the file (or less if file smaller than 512 bytes). However the data takes 516 bytes because of the 4 bytes taken by the opcode (00 03, indicating that it's a data packet) and the block number of the data. Here the block number is 00 01 meaning it's the first block. The server now awaits the ACK for block 1. We notice again the data readable directly from the packet which shows the content of the text file.

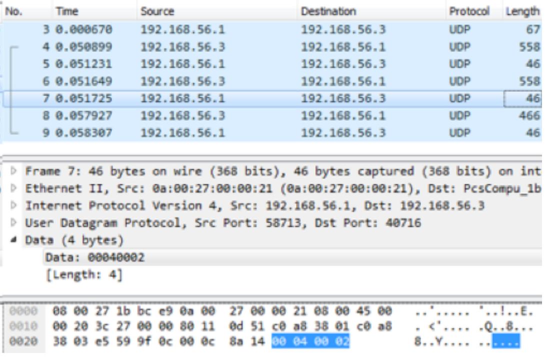


```
No.   Time        Source          Destination     Protocol  Length
  3 0.000670    192.168.56.1    192.168.56.3    UDP         67
  4 0.050899    192.168.56.3    192.168.56.1    UDP        558
  5 0.051231    192.168.56.1    192.168.56.3    UDP         46
  6 0.051649    192.168.56.3    192.168.56.1    UDP        558
  7 0.051725    192.168.56.1    192.168.56.3    UDP         46
  8 0.057927    192.168.56.3    192.168.56.1    UDP        466
  9 0.058307    192.168.56.1    192.168.56.3    UDP         46

▷ Frame 5: 46 bytes on wire (368 bits), 46 bytes captured (368 bits) on interf
▷ Ethernet II, Src: 0a:00:27:00:00:21 (0a:00:27:00:00:21), Dst: PcsCompu_1b:bc
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.3
▷ User Datagram Protocol, Src Port: 58713, Dst Port: 40716
◢ Data (4 bytes)
    Data: 00040001
    [Length: 4]

0000  08 00 27 1b bc e9 0a 00  27 00 00 21 08 00 45 00   ..'..... '..!..E.
0010  00 20 3c 26 00 00 80 11  0d 52 c0 a8 38 01 c0 a8   . <&.... .R..8...
0020  38 03 e5 59 9f 0c 00 0c  8a 15 00 04 00 01         8..Y.... ......
```

Client has successfully received data from the server's first packet. So it responds with packet with 00 04 as opcode, representing an ACK packet, and 00 01 as the block number.

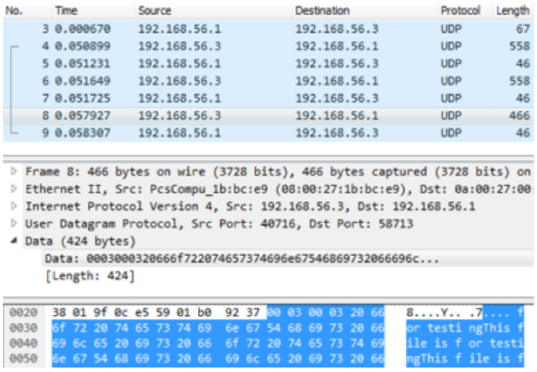


```
No.   Time        Source          Destination     Protocol  Length
  3 0.000670    192.168.56.1    192.168.56.3    UDP         67
  4 0.050899    192.168.56.3    192.168.56.1    UDP        558
  5 0.051231    192.168.56.1    192.168.56.3    UDP         46
  6 0.051649    192.168.56.3    192.168.56.1    UDP        558
  7 0.051725    192.168.56.1    192.168.56.3    UDP         46
  8 0.057927    192.168.56.3    192.168.56.1    UDP        466
  9 0.058307    192.168.56.1    192.168.56.3    UDP         46

▷ Frame 6: 558 bytes on wire (4464 bits), 558 bytes captured (4464 bits) on
▷ Ethernet II, Src: PcsCompu_1b:bc:e9 (08:00:27:1b:bc:e9), Dst: 0a:00:27:00
▷ Internet Protocol Version 4, Src: 192.168.56.3, Dst: 192.168.56.1
▷ User Datagram Protocol, Src Port: 40716, Dst Port: 58713
◢ Data (516 bytes)
    Data: 0003000220666f72207465737469 6e6754686973206669 6c...
    [Length: 516]

0020  38 01 9f 0c e5 59 02 0c  8c f7 00 03 00 02 20 66   8....Y.. ...... f
0030  6f 72 20 74 65 73 74 69  6e 67 54 68 69 73 20 66   or testi ngThis f
0040  69 6c 65 20 69 73 20 66  6f 72 20 74 65 73 74 69   ile is f or testi
```

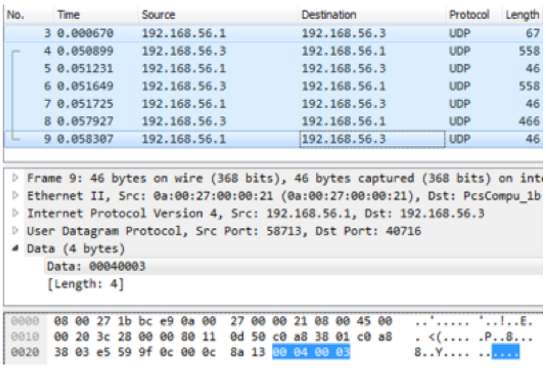


```
No.   Time        Source          Destination     Protocol  Length
  3 0.000670    192.168.56.1    192.168.56.3    UDP         67
  4 0.050899    192.168.56.3    192.168.56.1    UDP        558
  5 0.051231    192.168.56.1    192.168.56.3    UDP         46
  6 0.051649    192.168.56.3    192.168.56.1    UDP        558
  7 0.051725    192.168.56.1    192.168.56.3    UDP         46
  8 0.057927    192.168.56.3    192.168.56.1    UDP        466
  9 0.058307    192.168.56.1    192.168.56.3    UDP         46

▷ Frame 7: 46 bytes on wire (368 bits), 46 bytes captured (368 bits) on int
▷ Ethernet II, Src: 0a:00:27:00:00:21 (0a:00:27:00:00:21), Dst: PcsCompu_1b
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.3
▷ User Datagram Protocol, Src Port: 58713, Dst Port: 40716
◢ Data (4 bytes)
    Data: 00040002
    [Length: 4]

0000  08 00 27 1b bc e9 0a 00  27 00 00 21 08 00 45 00   ..'..... '..!..E.
0010  00 20 3c 27 00 00 80 11  0d 51 c0 a8 38 01 c0 a8   . <'.... .Q..8...
0020  38 03 e5 59 9f 0c 00 0c  8a 14 00 04 00 02         8..Y.... ......
```

The following packets are similar to the previous ones. When the previous data packet has been acknowledge and send a new data packet (opcode 00 03) with the previous block number +1 (00 02). Data size still 512 bytes as it is not last packet. The client receives the data and acknowledges it with another packet (opcode 00 04 and block 00 02).



```
No.   Time        Source          Destination     Protocol  Length
  3 0.000670    192.168.56.1    192.168.56.3    UDP         67
  4 0.050899    192.168.56.3    192.168.56.1    UDP        558
  5 0.051231    192.168.56.1    192.168.56.3    UDP         46
  6 0.051649    192.168.56.3    192.168.56.1    UDP        558
  7 0.051725    192.168.56.1    192.168.56.3    UDP         46
  8 0.057927    192.168.56.3    192.168.56.1    UDP        466
  9 0.058307    192.168.56.1    192.168.56.3    UDP         46

▷ Frame 8: 466 bytes on wire (3728 bits), 466 bytes captured (3728 bits) on
▷ Ethernet II, Src: PcsCompu_1b:bc:e9 (08:00:27:1b:bc:e9), Dst: 0a:00:27:00
▷ Internet Protocol Version 4, Src: 192.168.56.3, Dst: 192.168.56.1
▷ User Datagram Protocol, Src Port: 40716, Dst Port: 58713
◢ Data (424 bytes)
    Data: 0003000320666f72207465737469 6e6754686973206669 6c...
    [Length: 424]

0020  38 01 9f 0c e5 59 01 b0  92 37 00 03 00 03 20 66   8....Y.. .7.... f
0030  6f 72 20 74 65 73 74 69  6e 67 54 68 69 73 20 66   or testi ngThis f
0040  69 6c 65 20 69 73 20 66  6f 72 20 74 65 73 74 69   ile is f or testi
0050  6e 67 54 68 69 73 20 66  69 6c 65 20 69 73 20 66   ngThis f ile is f
```



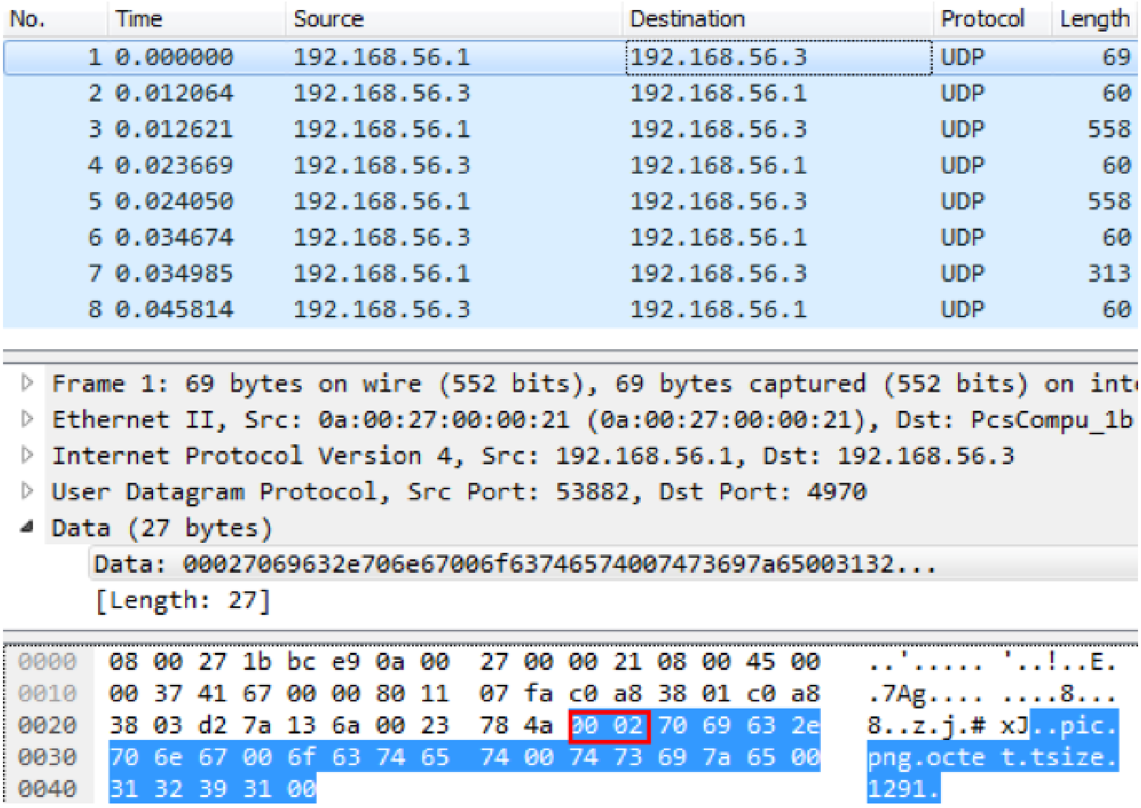
```
No.   Time        Source          Destination     Protocol  Length
  3 0.000670    192.168.56.1    192.168.56.3    UDP         67
  4 0.050899    192.168.56.3    192.168.56.1    UDP        558
  5 0.051231    192.168.56.1    192.168.56.3    UDP         46
  6 0.051649    192.168.56.3    192.168.56.1    UDP        558
  7 0.051725    192.168.56.1    192.168.56.3    UDP         46
  8 0.057927    192.168.56.3    192.168.56.1    UDP        466
  9 0.058307    192.168.56.1    192.168.56.3    UDP         46

▷ Frame 9: 46 bytes on wire (368 bits), 46 bytes captured (368 bits) on int
▷ Ethernet II, Src: 0a:00:27:00:00:21 (0a:00:27:00:00:21), Dst: PcsCompu_1b
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.3
▷ User Datagram Protocol, Src Port: 58713, Dst Port: 40716
◢ Data (4 bytes)
    Data: 00040003
    [Length: 4]

0000  08 00 27 1b bc e9 0a 00  27 00 00 21 08 00 45 00   ..'..... '..!..E.
0010  00 20 3c 28 00 00 80 11  0d 50 c0 a8 38 01 c0 a8   . <(.... .P..8...
0020  38 03 e5 59 9f 0c 00 0c  8a 13 00 04 00 03         8..Y.... ......
```

The server have receive the ACK. The file has only 420 more bytes to be sent since it was 1444 bytes and 512 bytes has already been sent two times. Thus the packet send by the server take again the opcode 00 03 (data packet) and the next block number being 00 03 to reach a 424 bytes data size.

The client receive this packet and deduce it's the last one since its size is smaller than 512 bytes (516 if opcode and block number are counted). It sends a last ACK with an opcode 00 04 and the block nr 00 03 to inform the server that the data has been received.

What is the difference between a read and a write request?

I sent an image of size 1291 bytes through write



No.	Time	Source	Destination	Protocol	Length
1	0.000000	192.168.56.1	192.168.56.3	UDP	69
2	0.012064	192.168.56.3	192.168.56.1	UDP	60
3	0.012621	192.168.56.1	192.168.56.3	UDP	558
4	0.023669	192.168.56.3	192.168.56.1	UDP	60
5	0.024050	192.168.56.1	192.168.56.3	UDP	558
6	0.034674	192.168.56.3	192.168.56.1	UDP	60
7	0.034985	192.168.56.1	192.168.56.3	UDP	313
8	0.045814	192.168.56.3	192.168.56.1	UDP	60

```
▷ Frame 1: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on int
▷ Ethernet II, Src: 0a:00:27:00:00:21 (0a:00:27:00:00:21), Dst: PcsCompu_1b
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.3
▷ User Datagram Protocol, Src Port: 53882, Dst Port: 4970
◢ Data (27 bytes)
     Data: 00027069632e706e67006f63746574007473697a65003132...
     [Length: 27]
```

```
0000  08 00 27 1b bc e9 0a 00  27 00 00 21 08 00 45 00   ..'..... '..!..E.
0010  00 37 41 67 00 00 80 11  07 fa c0 a8 38 01 c0 a8   .7Ag.... ....8...
0020  38 03 d2 7a 13 6a 00 23  78 4a 00 02 70 69 63 2e   8..z.j.# xJ..pic.
0030  70 6e 67 00 6f 63 74 65  74 00 74 73 69 7a 65 00   png.octe t.tsize.
0040  31 32 39 31 00                                     1291.
```

As for the Read request, the client start by a request packet, specifying the type of request, the name of the file and the mode of transfer. Here the opcode for the request is 00 02 representing "write" instead of 00 01 for "read".

After that the client will not start immediately sending data but the server will first acknowledge the reception of the write request with a packet with an 00 04 opcode (ACK) and a 00 00 block number.

When the client receive this ACK, it starts sending 512 bytes (516 with opcode and block number) block of data and wait for the ACK from the server after the it's sent. This part of the process is very similar to the read request, the client and the server are just switch.

When the client send the last data with a packet smaller than 512 bytes, the server send a last acknowledgement packet with the right block number and the transmission is finished.

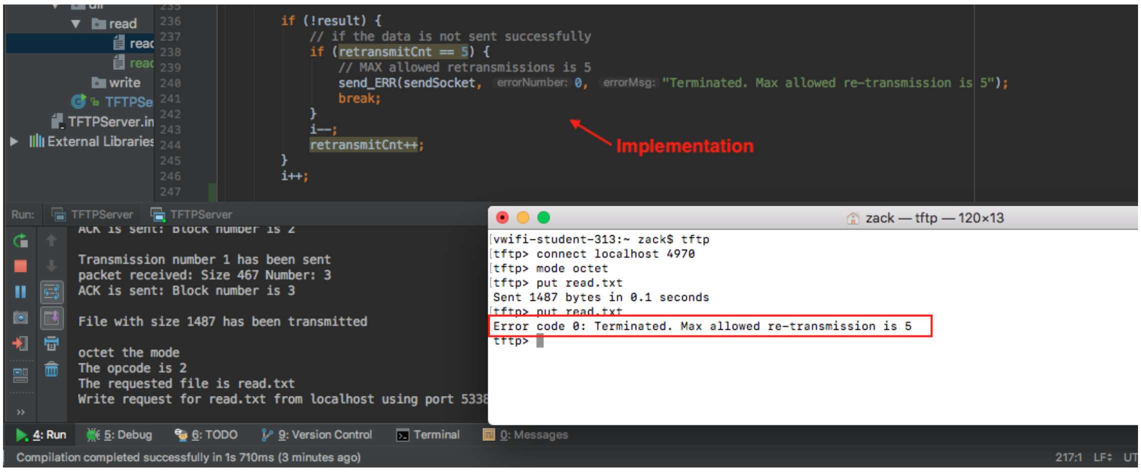
The difference with read request is that the opcode in the request packet differs and after that, the roles from the 2 entities are switched. The client wait the ACK and send the data. The server waits for the data and send ACK.

4 Problem Three:

This problem was about implementing a different error response that are provided in the file Rcf1350. What is asked is to implement errors number 0,1,2,6 and then as a VG task implement the rest 3,4,5.

4.1 Error 0:

This error is not specified or defined and in our implementation the server throws this error when the retransmission happen more than five times as we set the max allowed retransmission times is 5. Whenever the packet is lost or corrupted the server will retransmit it and to prevent the server from getting stuck in this loop, it was set to be only 5 time.



```
if (!result) {
    // if the data is not sent successfully
    if (retransmitCnt == 5) {
        // MAX allowed retransmissions is 5
        send_ERR(sendSocket, errorNumber: 0, errorMsg: "Terminated. Max allowed re-transmission is 5");
        break;
    }
    i--;
    retransmitCnt++;
}
i++;
```

Implementation

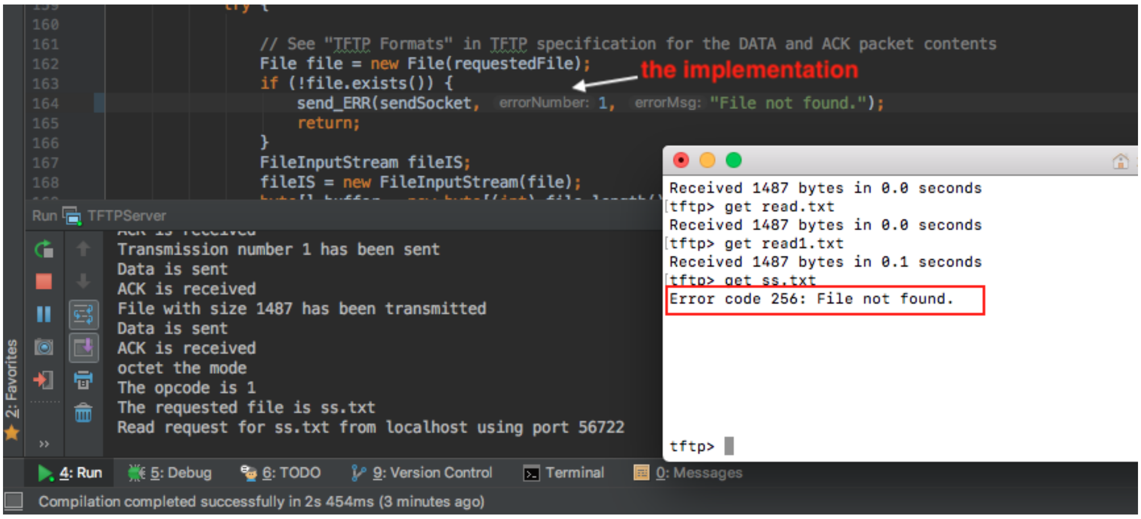
```
ACK is sent: Block number is 2

Transmission number 1 has been sent
packet received: Size 467 Number: 3
ACK is sent: Block number is 3

File with size 1487 has been transmitted

octet the mode
The opcode is 2
The requested file is read.txt
Write request for read.txt from localhost using port 5338
```

```
vwifi-student-313:~ zack$ tftp
tftp> connect localhost 4970
tftp> mode octet
tftp> put read.txt
Sent 1487 bytes in 0.1 seconds
tftp> put read.txt
Error code 0: Terminated. Max allowed re-transmission is 5
tftp>
```

read
write
TFTPServer.in
External Libraries
Run: TFTPServer TFTPServer
zack — tftp — 120×13
4: Run 5: Debug 6: TODO 9: Version Control Terminal 0: Messages
Compilation completed successfully in 1s 710ms (3 minutes ago)
217:1 LF÷

4.2 Error 1:

This error message according to the TFTP file specification should be thrown when the requested file is not found.



```
try {
    // See "TFTP Formats" in TFTP specification for the DATA and ACK packet contents
    File file = new File(requestedFile);
    if (!file.exists()) {
        send_ERR(sendSocket, errorNumber: 1, errorMsg: "File not found.");
        return;
    }
    FileInputStream fileIS;
    fileIS = new FileInputStream(file);
```

the implementation

```
ACK is received
Transmission number 1 has been sent
Data is sent
ACK is received
File with size 1487 has been transmitted
Data is sent
ACK is received
octet the mode
The opcode is 1
The requested file is ss.txt
Read request for ss.txt from localhost using port 56722
```

```
Received 1487 bytes in 0.0 seconds
tftp> get read.txt
Received 1487 bytes in 0.0 seconds
tftp> get read1.txt
Received 1487 bytes in 0.1 seconds
tftp> get ss.txt
Error code 256: File not found.

tftp>
```

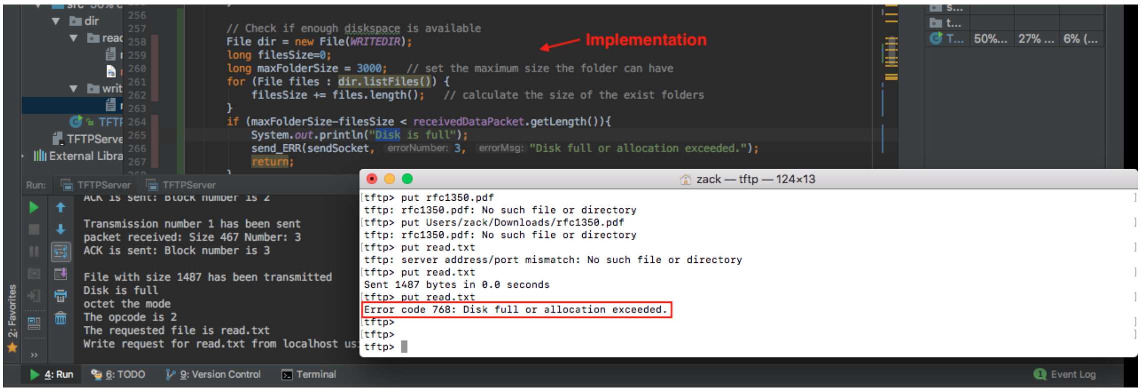
Run TFTPServer
2: Favorites
4: Run 5: Debug 6: TODO 9: Version Control Terminal 0: Messages
Compilation completed successfully in 2s 454ms (3 minutes ago)

4.3 Error 2:

This error occurs whenever there is an access violation on the file we need to perform an operation on it. As it was written in the assignment we generate this error when an IOException happens. I did not manage to reproduce it so I couldn't take a screenshot.

4.4 Error 3:

This error has the message or the meaning "Disk full or allocation exceeded" which is supposed to happen every time the client try to put a file on a folder that does not have enough space to have the folder. To reproduce it I set the write directory to 3MB and tried to put a file that has a size bigger than the enough space.



```java
// Check if enough diskspace is available
File dir = new File(WRITEDIR);
long filesSize=0;
long maxFolderSize = 3000;   // set the maximum size the folder can have
for (File files : dir.listFiles()) {
    filesSize += files.length();   // calculate the size of the exist folders
}
if (maxFolderSize-filesSize < receivedDataPacket.getLength()){
    System.out.println("Disk is full");
    send_ERR(sendSocket, errorNumber: 3, errorMsg: "Disk full or allocation exceeded.");
    return;
```

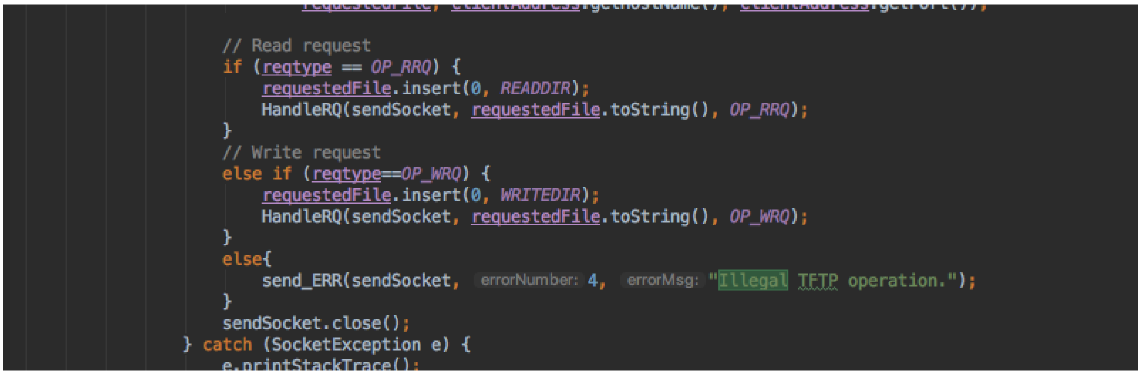
```
ACK is sent: Block number is 2

Transmission number 1 has been sent
packet received: Size 467 Number: 3
ACK is sent: Block number is 3

File with size 1487 has been transmitted
Disk is full
octet the mode
The opcode is 2
The requested file is read.txt
Write request for read.txt from localhost us
```

zack — tftp — 124×13

```
tftp> put rfc1350.pdf
tftp: rfc1350.pdf: No such file or directory
tftp> put Users/zack/Downloads/rfc1350.pdf
tftp: rfc1350.pdf: No such file or directory
tftp> put read.txt
tftp: server address/port mismatch: No such file or directory
tftp> put read.txt
Sent 1487 bytes in 0.0 seconds
tftp> put read.txt
Error code 768: Disk full or allocation exceeded.
tftp>
tftp>
tftp>
```

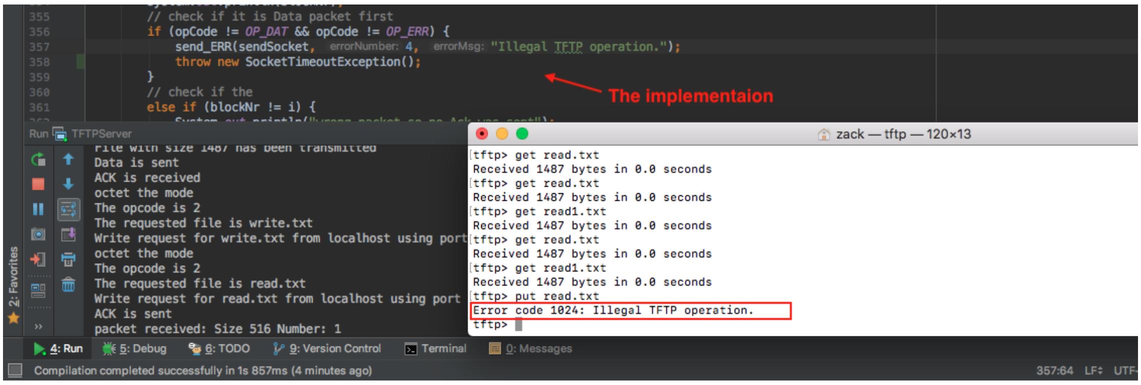
4.5 Error 4:

ILLEGAL TFTP OPEREATION. As we only have write and read or put/get legal operations. The server should throw this error when any other operation is performed and in our implementation we implemented this error when a the request type is not read or write as it is shown in figure 4.6 And whenever we the server is supposed to get an ACK but instead getting another packet which is not also an error packet or waiting for data packet and receive another kind of packet which is not error packet as well like in Figure 4.7



```java
                    requestedFile, clientAddress.getHostName(), clientAddress.getPort());

                // Read request
                if (reqtype == OP_RRQ) {
                    requestedFile.insert(0, READDIR);
                    HandleRQ(sendSocket, requestedFile.toString(), OP_RRQ);
                }
                // Write request
                else if (reqtype==OP_WRQ) {
                    requestedFile.insert(0, WRITEDIR);
                    HandleRQ(sendSocket, requestedFile.toString(), OP_WRQ);
                }
                else{
                    send_ERR(sendSocket, errorNumber: 4, errorMsg: "Illegal TFTP operation.");
                }
                sendSocket.close();
            } catch (SocketException e) {
                e.printStackTrace();
```

Figure 4.1:



```
// check if it is Data packet first
if (opCode != OP_DAT && opCode != OP_ERR) {
    send_ERR(sendSocket, errorNumber: 4, errorMsg: "Illegal TFTP operation.");
    throw new SocketTimeoutException();
}
// check if the
else if (blockNr != i) {
```

The implementaion

```
File with size 1487 has been transmitted
Data is sent
ACK is received
octet the mode
The opcode is 2
The requested file is write.txt
Write request for write.txt from localhost using port
octet the mode
The opcode is 2
The requested file is read.txt
Write request for read.txt from localhost using port
ACK is sent
packet received: Size 516 Number: 1
```

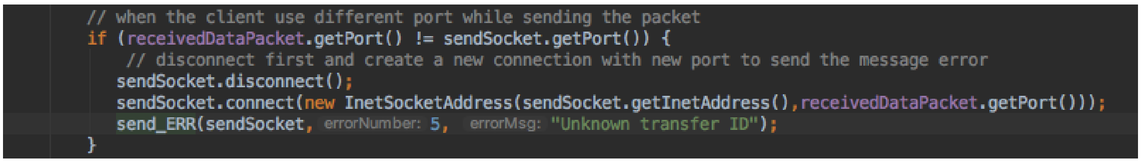
```
tftp> get read.txt
Received 1487 bytes in 0.0 seconds
tftp> get read.txt
Received 1487 bytes in 0.0 seconds
tftp> get read1.txt
Received 1487 bytes in 0.0 seconds
tftp> get read.txt
Received 1487 bytes in 0.0 seconds
tftp> get read1.txt
Received 1487 bytes in 0.0 seconds
tftp> put read.txt
Error code 1024: Illegal TFTP operation.
tftp>
```

Compilation completed successfully in 1s 857ms (4 minutes ago)

Figure 4.2:

4.6 Error 5:

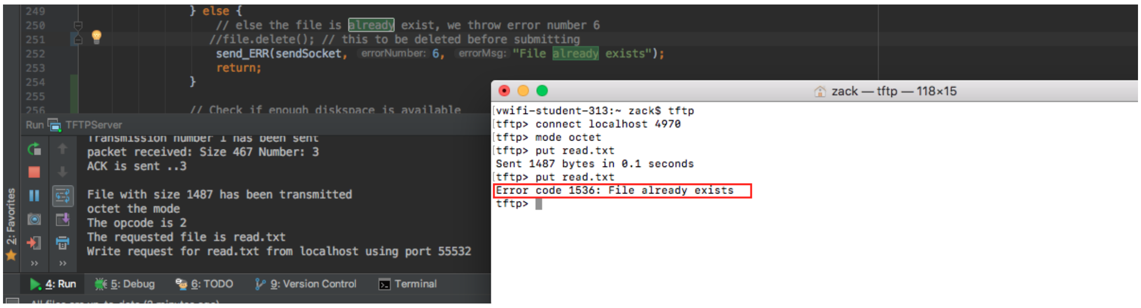
Unknown Transfer ID. When the client suddenly change the port, it was using to send the data or to communicate with the server. However, I couldn't capture this error but in figure you can see the implementation.



```
// when the client use different port while sending the packet
if (receivedDataPacket.getPort() != sendSocket.getPort()) {
    // disconnect first and create a new connection with new port to send the message error
   sendSocket.disconnect();
   sendSocket.connect(new InetSocketAddress(sendSocket.getInetAddress(),receivedDataPacket.getPort()));
   send_ERR(sendSocket, errorNumber: 5, errorMsg: "Unknown transfer ID");
}
```

4.7 Error 6:

File Already exist. When the client requests the operation write on an already existing file in the write directory of our server. Figure 4.8 shows the implementation and the error response.



```
} else {
    // else the file is already exist, we throw error number 6
   //file.delete(); // this to be deleted before submitting
    send_ERR(sendSocket, errorNumber: 6, errorMsg: "File already exists");
    return;
}

// Check if enough diskspace is available
```

```
Transmission number 1 has been sent
packet received: Size 467 Number: 3
ACK is sent ..3

File with size 1487 has been transmitted
octet the mode
The opcode is 2
The requested file is read.txt
Write request for read.txt from localhost using port 55532
```

```
vwifi-student-313:~ zack$ tftp
tftp> connect localhost 4970
tftp> mode octet
tftp> put read.txt
Sent 1487 bytes in 0.1 seconds
tftp> put read.txt
Error code 1536: File already exists
tftp>
```

Figure 4.3: