Mehdi Hmidi Mh223vk

Problem 1

TFTP server functions according to RFC1350. It listens to client requests on given port 4970. Provided code inserts the corresponding directory to either read or write.

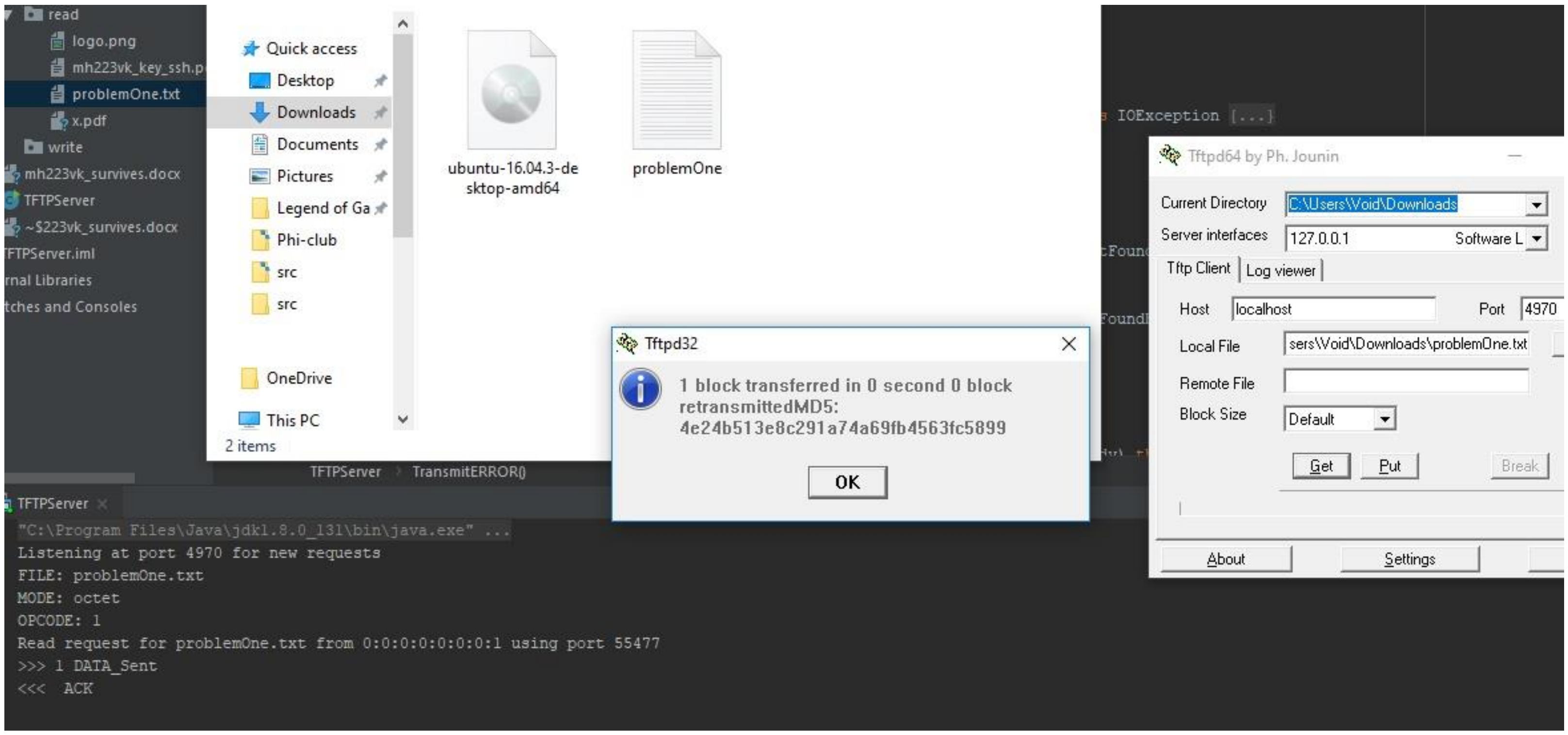
I take the request then parse it with ParseRQ required method.

I display the opcode 00 01 for “read” request and 00 02 for “write” request.

I detect errors in mode (only octet allowed) and the requested file name.

I start my TFTP client and specify settings including the directory I wish to store files into.

I click “GET” and the transfer is successful. Files bigger that MTU = 512 are handled in Problem 2



Tftpd32

1 block transferred in 0 second 0 block retransmittedMD5: 4e24b513e8c291a74a69fb4563fc5899

OK

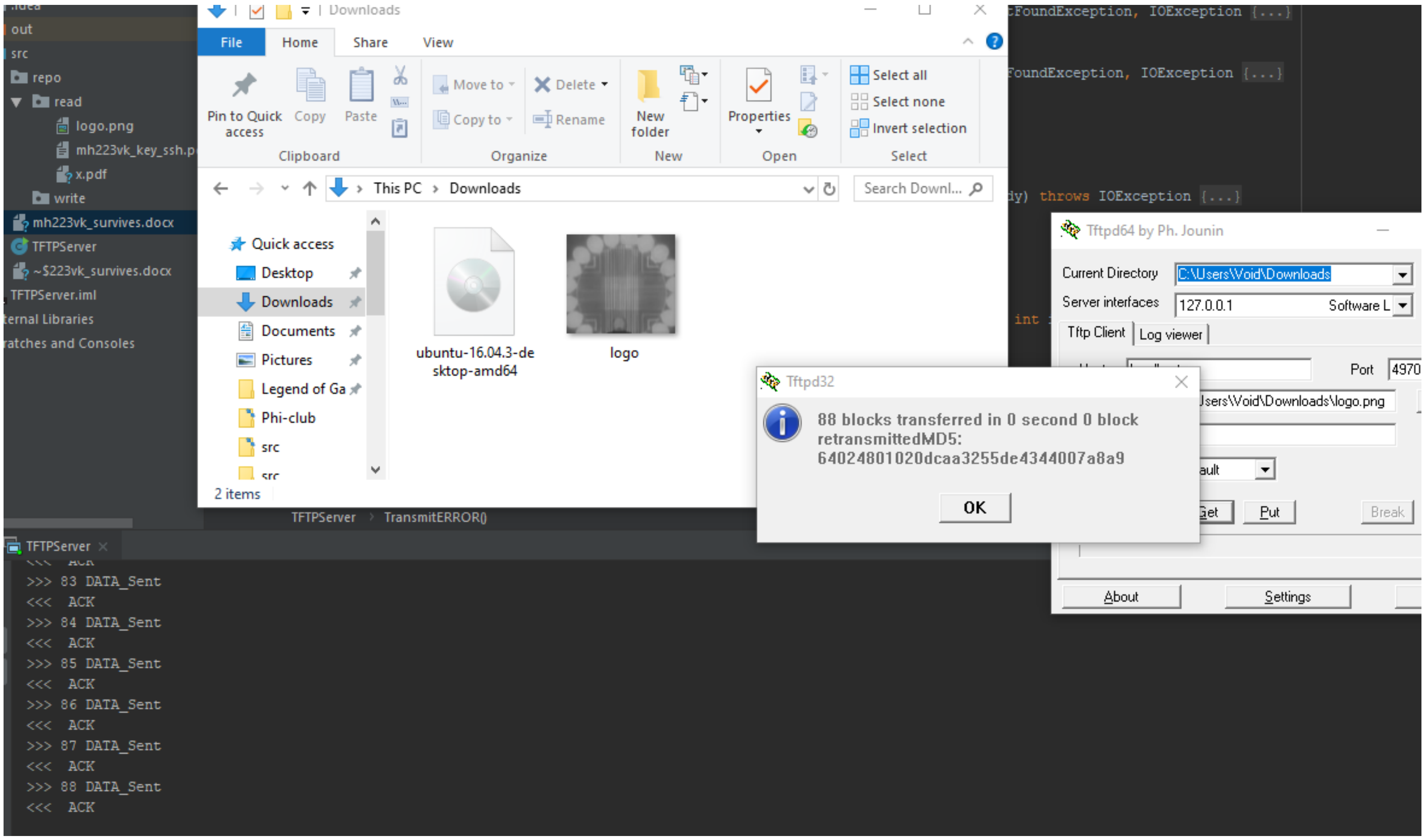
Current Directory: C:\Users\Void\Downloads
Server interfaces: 127.0.0.1 Software L
Host: localhost Port: 4970
Local File: sers\Void\Downloads\problemOne.txt
Remote File:
Block Size: Default

```
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
Listening at port 4970 for new requests
FILE: problemOne.txt
MODE: octet
OPCODE: 1
Read request for problemOne.txt from 0:0:0:0:0:0:0:1 using port 55477
>>> 1 DATA_Sent
<<<  ACK
```

Send socket looks for free ports after receiving a request, informs client about port for the connection, by including it in the first data packet or the ACK packet. Socket and send socket help handle multiple requests from multiple clients simultaneously.

Problem 2

Add a functionality that makes it possible to handle files larger than 512 bytes.


```
88 blocks transferred in 0 second 0 block
retransmittedMD5:
64024801020dcaa3255de4344007a8a9
```

```
>>> 83 DATA_Sent
<<<  ACK
>>> 84 DATA_Sent
<<<  ACK
>>> 85 DATA_Sent
<<<  ACK
>>> 86 DATA_Sent
<<<  ACK
>>> 87 DATA_Sent
<<<  ACK
>>> 88 DATA_Sent
<<<  ACK
```

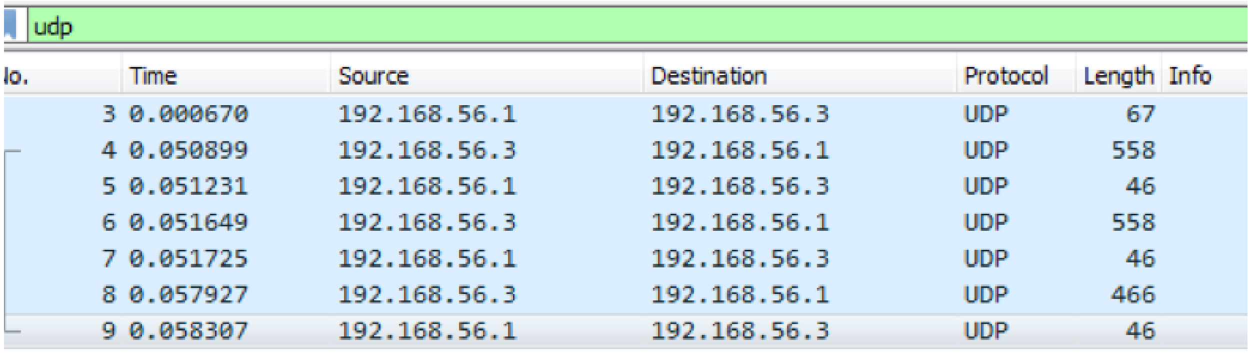
Server divides the file size by MTU = 512 then sends the first block of bytes with ***send-DATA-receive-ACK method***, and waits for the ACK from previous block of data. In essence the Server iterates until last block of data is acknowledged.

I added a timeout in case there is a problem during the transfer of data packets. Only 5 re-transmissions maximum are allowed.

For the Write request, ***HandleRQ*** starts by sending an ACK packet to initiate transaction with the client that in turn starts sending the file. In turn, ***receive-DATA-send-ACK*** checks if opcode is 00 03 for DATA packets and checks if the block number is correct. ACK is in turn sent back to the client. When the data packet received is smaller than MTU = 512 bytes, it is the last packet and no more data is expected, since it is assumed we send the maximum allowed unit in every block. If ACK is not received after sending a data packet, it will be re-transmitted again a maximum of 5 times then Server sends an error.

VG task:

I didn’t download a localhost adapter for Wireshark to analyze my local interface. To get a report on packets flowing to and from the server, I used an already setup virtual machine running Ubuntu. On the virtual environment my Server was running and Wireshark captured the packets transiting through the network interface. Next, I display a get request servicing a client’s read request.



udp

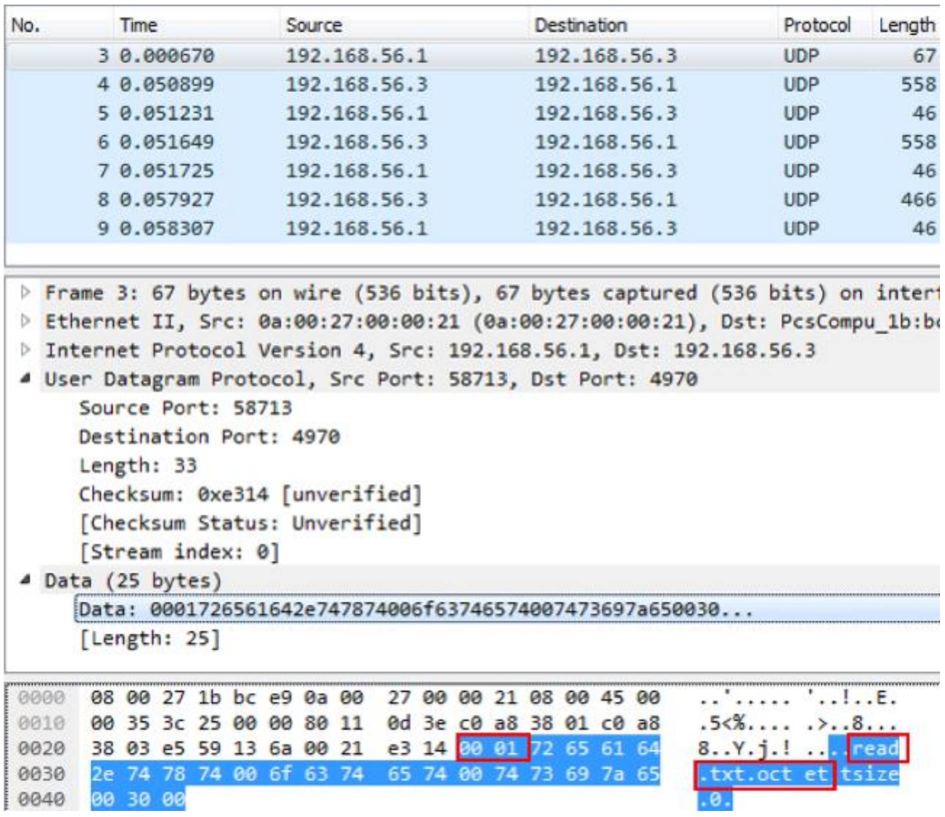
lo.	Time	Source	Destination	Protocol	Length	Info
3	0.000670	192.168.56.1	192.168.56.3	UDP	67	
4	0.050899	192.168.56.3	192.168.56.1	UDP	558	
5	0.051231	192.168.56.1	192.168.56.3	UDP	46	
6	0.051649	192.168.56.3	192.168.56.1	UDP	558	
7	0.051725	192.168.56.1	192.168.56.3	UDP	46	
8	0.057927	192.168.56.3	192.168.56.1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

Result on Wireshark

We see a difference in the length column to symbolize DATA packets and ACK packets being sent over the net interface.

Attempt 1:

First 2 bytes from the data represent the opcode indicating to the server that the request is a read. Here it is 00 01 which represent a read request .We also notice 45 bytes of headers from UDP and IPv4 headers are included



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

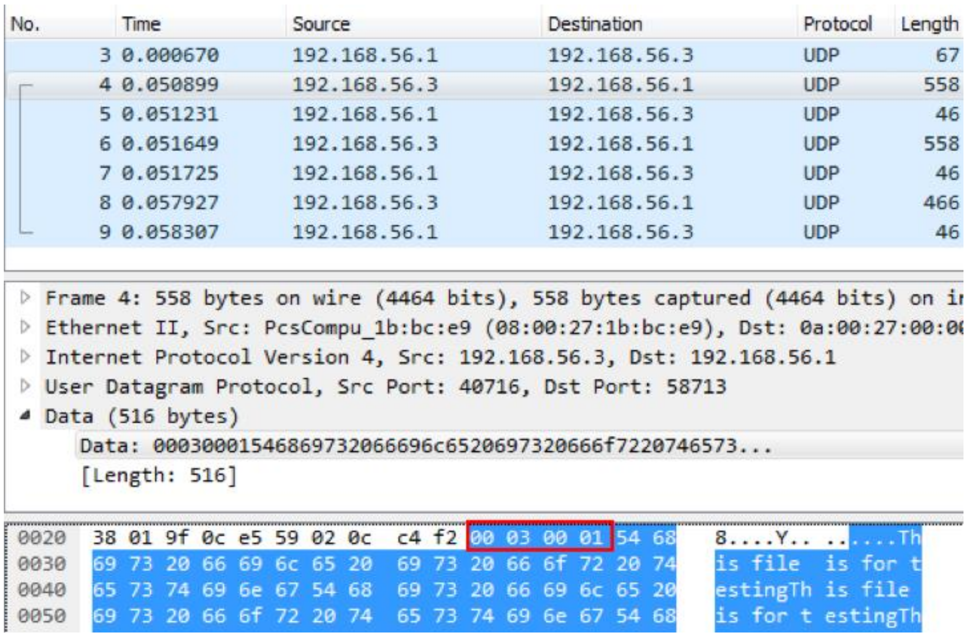
```
▷ Frame 3: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on inter
▷ Ethernet II, Src: 0a:00:27:00:00:21 (0a:00:27:00:00:21), Dst: PcsCompu_1b:b
▷ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.3
◢ User Datagram Protocol, Src Port: 58713, Dst Port: 4970
    Source Port: 58713
    Destination Port: 4970
    Length: 33
    Checksum: 0xe314 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
◢ Data (25 bytes)
    Data: 0001726561642e747874006f637465740074736 97a650030...
    [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.
```

Clearly, sent opcode is followed by the name of the requested file and the mode of transfer as it specified in the RFC for TFTP transfers.

Second in nature is the server's response to the client. Data block sent holds 516 of bytes since 4 bytes are taken by the opcode (00 03, DATA packet) and the block number of the data.

Here the block number is 00 01. In continuation, the server holds and waits for the client's ACK for block 1.

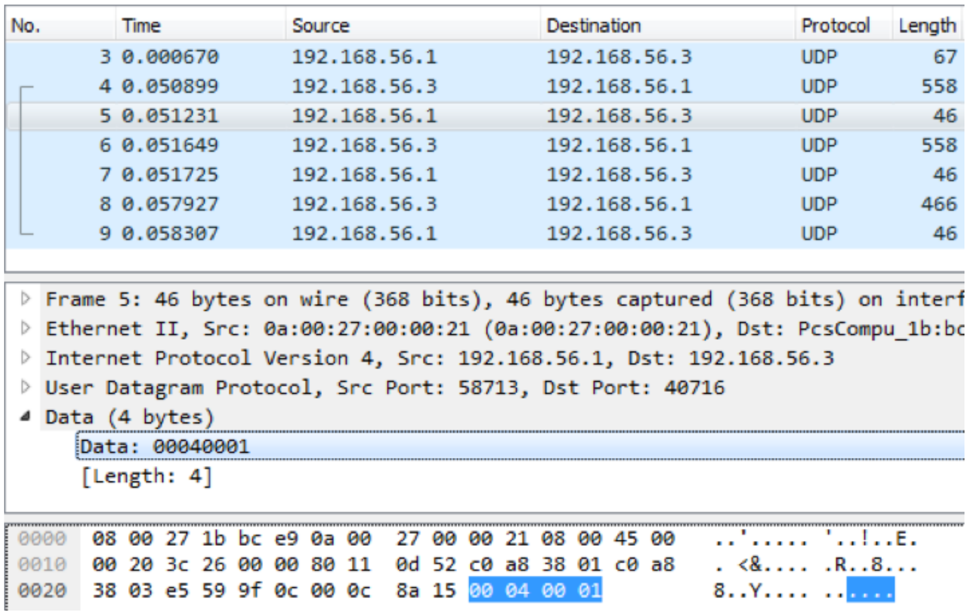


```
No.    Time        Source          Destination     Protocol  Length
    3 0.000670    192.168.56.1    192.168.56.3    UDP         67
    4 0.050899    192.168.56.3    192.168.56.1    UDP        558
    5 0.051231    192.168.56.1    192.168.56.3    UDP         46
    6 0.051649    192.168.56.3    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
```

Once the client successfully receives data, it responds with opcode 00 04, for an ACK, that acknowledges the 00 01 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 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.... ......
```

Analysis line by line

- Read request from client to TFTP Server (Example size 26 bytes)
- NBNS protocol which translates the file names to IP addresses.
- LLMNR protocol, allows hosts on the same network to perform name resolution.
- First data packet is sent from the server to client. The UDP packet length is 558, the data packet is 516. It´s actual size is 512 but we need 2 bytes for the opcode and 2 more for the block number.
- LLMNR protocol.
- ACK packet from client to server, for the first data packet.
- Data packet of requested file.
- ACK packet for last data packet.
- The client receive this packet and deduce it’s the last one since its size is smaller than 512 bytes
- Client sends a last ACK with an opcode 00 04 and the block nr to inform the server that the data has been received.

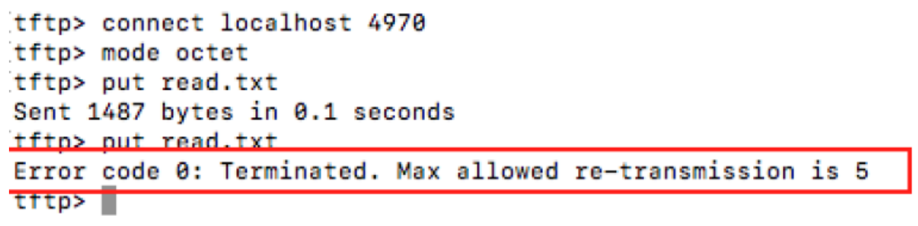
What is the difference between a read and a write request?

Read request:
- Op code is 00 01.
- Only when the packet is received it would send a new packet.
- The block number will be increased by 1, if the last sent packet was 512 bytes.
- When initial packet is sent => server waits for an ACK from the client.
- After the package is accepted the server will create a data packet of the requested file and send it to client
- Keyword to make this request is GET.

Write Request: *(sometimes you have to stop the server to open files since they are in runtime)*
- Op code is 2
- When “PUT” packet is received the server sends ACK packet to the client with same block number as the received data packet. If the last packet that is received has size of 512 bytes, the process would be repeated.
- Once the server has sent a first packet it would wait for a DATA packet from client.
- After accepting it the server would send an ACK packet with the same block Nr° to client.
- Keyword for this request is PUT.

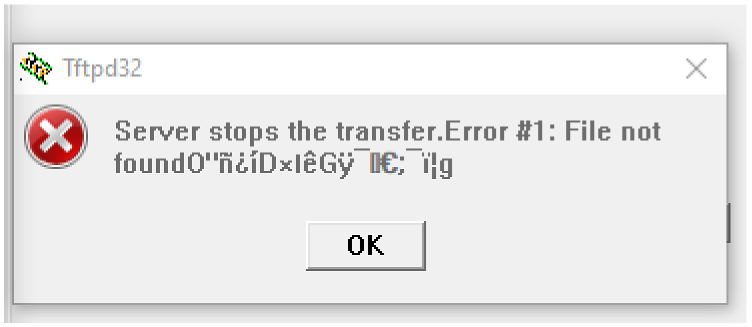
Problem 3:

ERROR 0: server throws this error when **File could not be read**, when the mode is not **OCTET** or the **retransmission happen more than five times**.

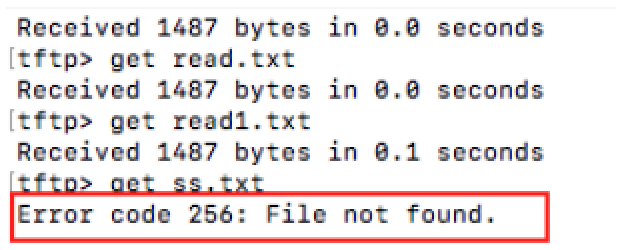


```
tftp> connect localhost 4970
tftp> mode octet
tftp> put read.txt
Sent 1487 bytes in 0.1 seconds
tftp> put read.txt
Error code 0: Terminated. Max allowed re-transmission is 5
tftp>
```

ERROR 1: File not found



```
Tftpd32
Server stops the transfer.Error #1: File not foundO"ñ¿íD×lêGÿ¯ÏЀ;¯ï¦g
OK
```



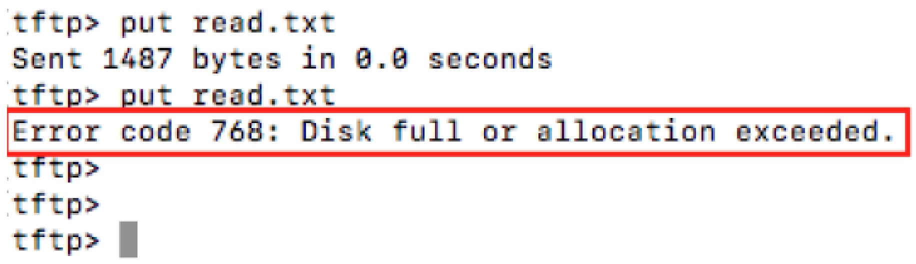
```
Received 1487 bytes in 0.0 seconds
tftp> get read.txt
Received 1487 bytes in 0.0 seconds
tftp> get read1.txt
Received 1487 bytes in 0.1 seconds
tftp> get ss.txt
Error code 256: File not found.
```

ERROR 2: Access violation

In the code I generate this error whenever an IOException happens. I couldn't find a way to quickly grab a screenshot and to create a situation where this error is created but I followed assignment objectives.

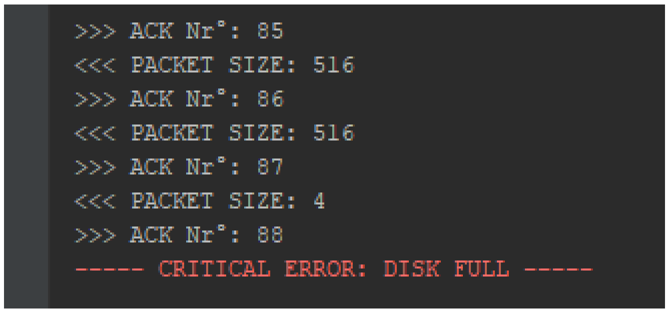
ERROR 3: Disk full

I set the write directory to 10MB and tried to put files that eventually exceeded the allocation of server storage.
The method responsible is the **databaseTier()**.



```
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>
```

On the server's side the same thing is sent

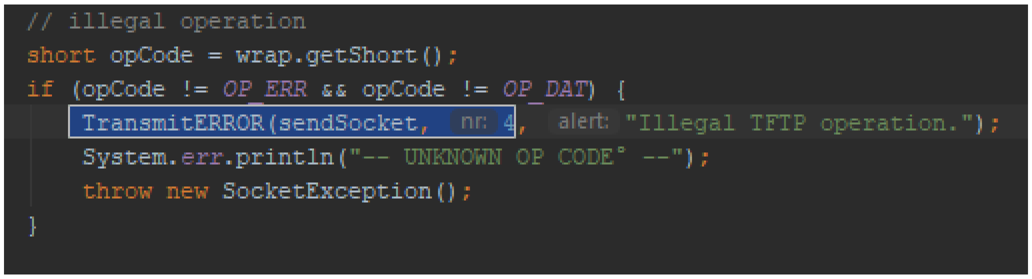


```
>>> ACK Nr°: 85
<<< PACKET SIZE: 516
>>> ACK Nr°: 86
<<< PACKET SIZE: 516
>>> ACK Nr°: 87
<<< PACKET SIZE: 4
>>> ACK Nr°: 88
----- CRITICAL ERROR: DISK FULL -----
```

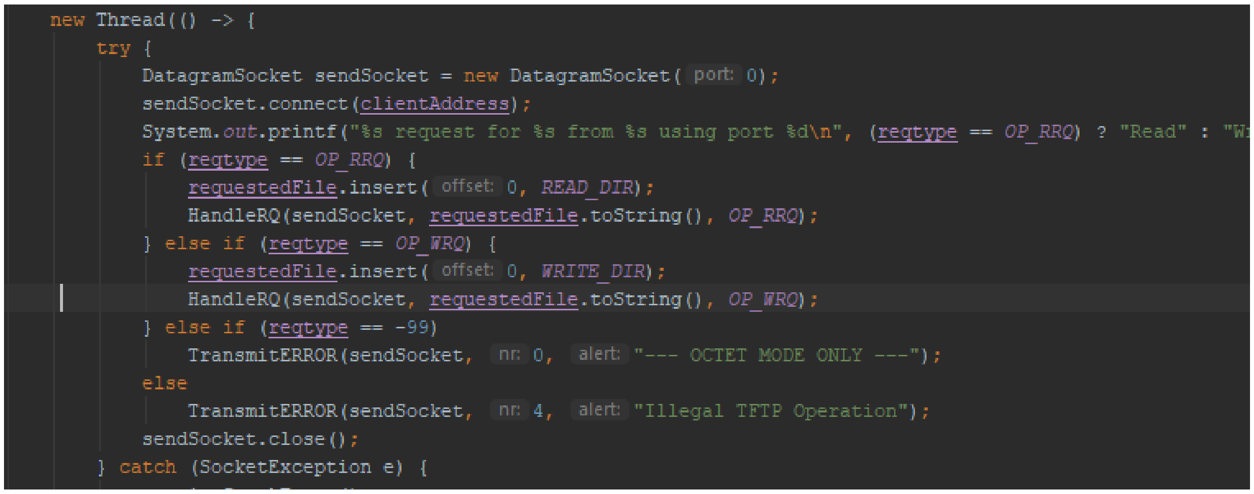
ERROR 4: ILLEGAL TFTP OPEREATION

Server should throw this error whenever:

- Opcode is not recognized.
- Supposed to get an ACK but instead gets a different kind of packet.



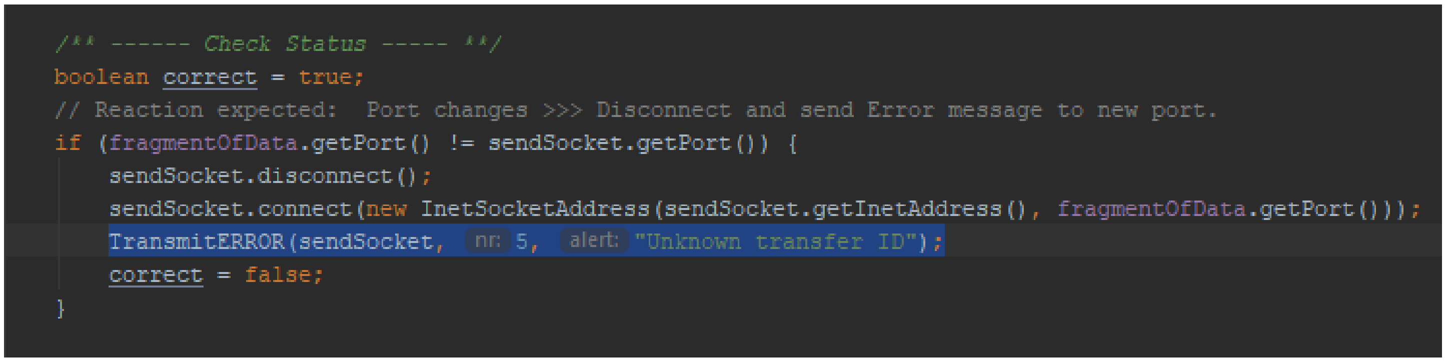
```
// illegal operation
short opCode = wrap.getShort();
if (opCode != OP_ERR && opCode != OP_DAT) {
    TransmitERROR(sendSocket, nr: 4, alert: "Illegal TFTP operation.");
    System.err.println("-- UNKNOWN OP CODE° --");
    throw new SocketException();
}
```



```
new Thread(() -> {
    try {
        DatagramSocket sendSocket = new DatagramSocket( port: 0);
        sendSocket.connect(clientAddress);
        System.out.printf("%s request for %s from %s using port %d\n", (reqtype == OP_RRQ) ? "Read" : "W
        if (reqtype == OP_RRQ) {
            requestedFile.insert( offset: 0, READ_DIR);
            HandleRQ(sendSocket, requestedFile.toString(), OP_RRQ);
        } else if (reqtype == OP_WRQ) {
            requestedFile.insert( offset: 0, WRITE_DIR);
            HandleRQ(sendSocket, requestedFile.toString(), OP_WRQ);
        } else if (reqtype == -99)
            TransmitERROR(sendSocket, nr: 0, alert: "--- OCTET MODE ONLY ---");
        else
            TransmitERROR(sendSocket, nr: 4, alert: "Illegal TFTP Operation");
        sendSocket.close();
    } catch (SocketException e) {
```

ERROR 5: Unknown Transfer ID

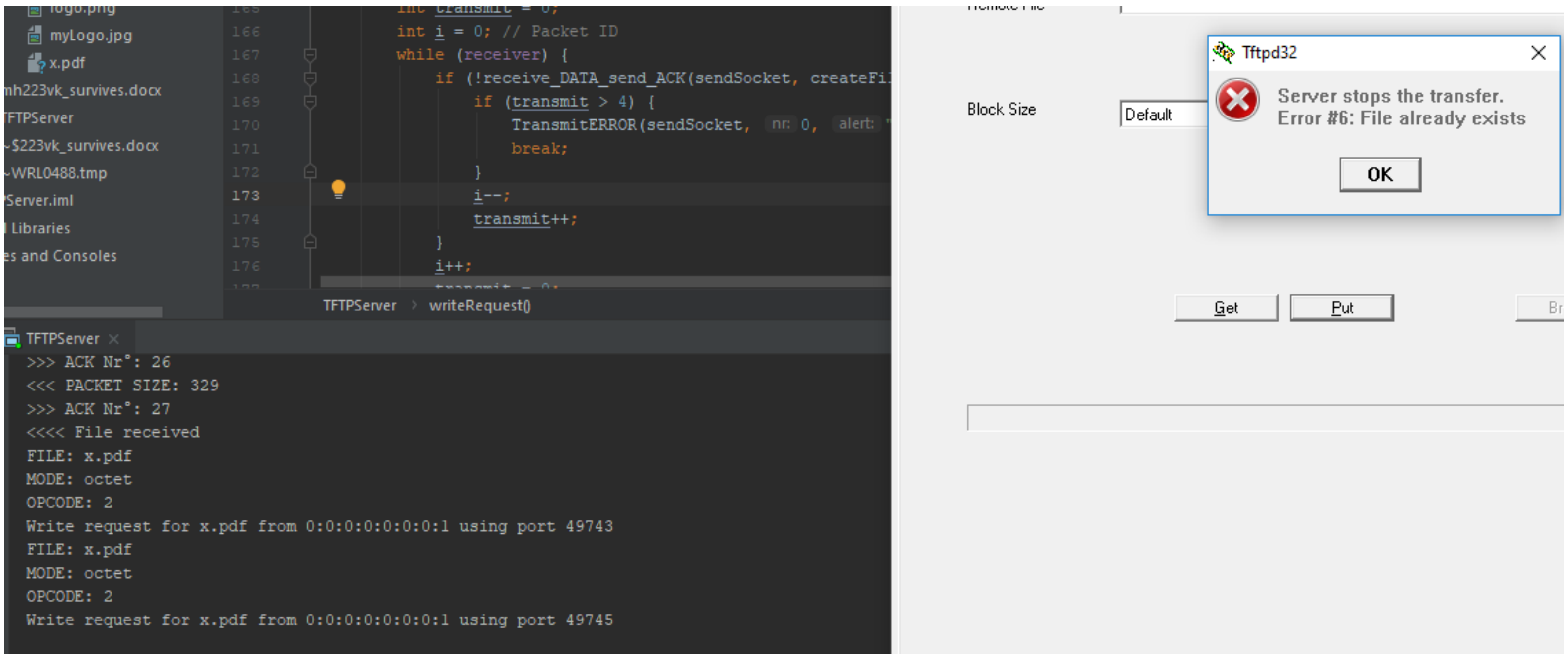
It is difficult to capture some of these errors so I change the code and see how it behaves. Keep this into consideration. You can check the error handling in the code.



```
/** ------ Check Status ----- **/
boolean correct = true;
// Reaction expected:  Port changes >>> Disconnect and send Error message to new port.
if (fragmentOfData.getPort() != sendSocket.getPort()) {
    sendSocket.disconnect();
    sendSocket.connect(new InetSocketAddress(sendSocket.getInetAddress(), fragmentOfData.getPort()));
    TransmitERROR(sendSocket,  nr: 5,  alert: "Unknown transfer ID");
    correct = false;
}
```

ERROR 6: File Already Exists

Client requests to write an already existing file to the server. System flags the attempt and sends an error. This is easy to capture.



```
logo.png
myLogo.jpg
x.pdf
mh223vk_survives.docx
TFTPServer
~$223vk_survives.docx
~WRL0488.tmp
Server.iml
Libraries
es and Consoles
```

```
        int transmit = 0;
        int i = 0; // Packet ID
        while (receiver) {
            if (!receive_DATA_send_ACK(sendSocket, createFi
                if (transmit > 4) {
                    TransmitERROR(sendSocket,  nr: 0,  alert:
                    break;
                }
                i--;
                transmit++;
            }
            i++;
            transmit = 0;
```

TFTPServer › writeRequest()

```
>>> ACK Nr°: 26
<<< PACKET SIZE: 329
>>> ACK Nr°: 27
<<<< File received
FILE: x.pdf
MODE: octet
OPCODE: 2
Write request for x.pdf from 0:0:0:0:0:0:0:1 using port 49743
FILE: x.pdf
MODE: octet
OPCODE: 2
Write request for x.pdf from 0:0:0:0:0:0:0:1 using port 49745
```

Tftpd32

Server stops the transfer.
Error #6: File already exists