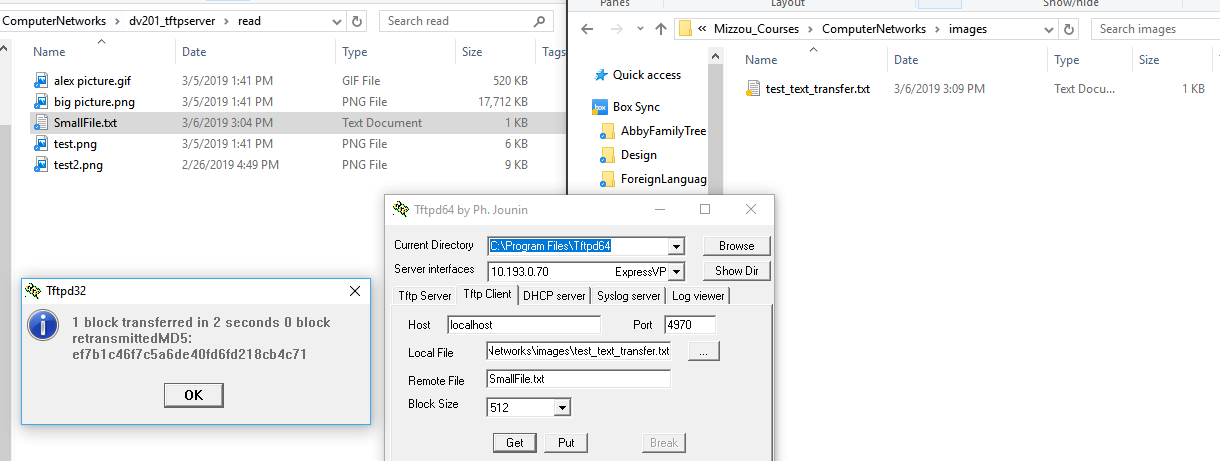
DV201 (Software Engineering) Assignment 3 (TFTP Server)

By: Alex and Fabian

# Problem 1 (TFTP Server):

**Read Request from Client with file less than 512 Bytes (Screenshot):**

In the following test the windows explorer window shows the contents of the “read” directory for the TFTP server. This directory contains all the available files that the client user can get. The explorer window on the right shows the local directory for the client user where the resultant image was stored. The TFTP client at the bottom shows that we will connect to localhost on port 4970 and in this case get the file SmallFile.txt. Since the file is less that 512 bytes it only requires 1 block to transfer. When the “Get” button is pressed the transfer begins and the popup message indicates that the 1 block has been successfully transferred and the file on the local filesystem is shown on the left (test\_text\_transfer.txt)



**Socket and sendSocket:**

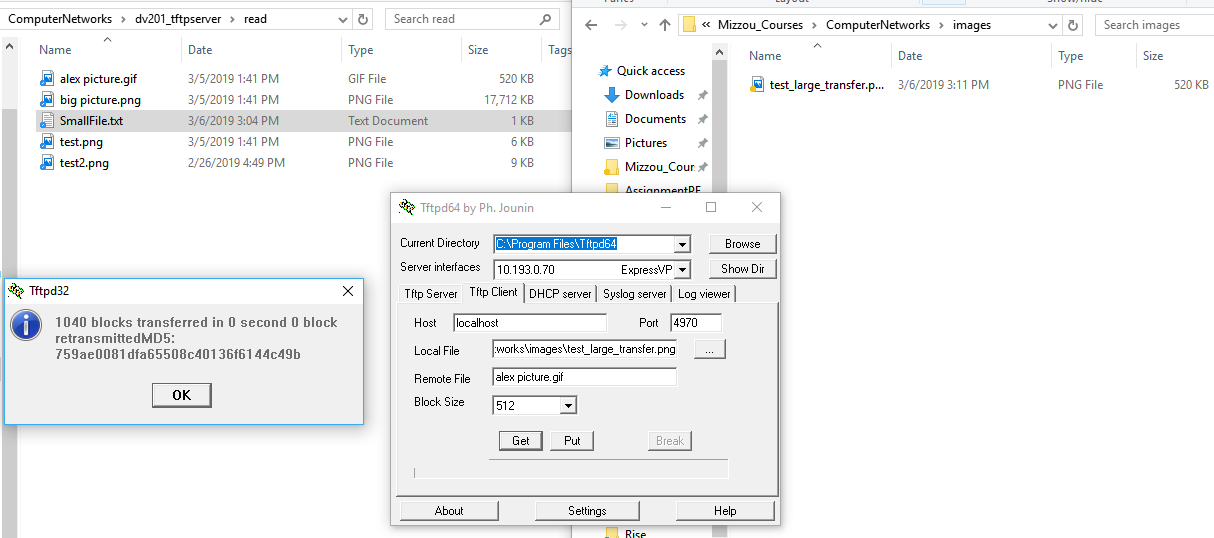
The reason that the server uses socket and sendSocket is that socket is used to start the server on a known port so incoming connections can be accepted. sendSocket is then used when a connection is accepted with socket to open a new random port so that the file/data can be transferred from the server to the client. This is done so that the known port the server runs on can be open and only used to accept new incoming requests while the server is handling/transmitting files to clients on other ports. sendSocket picks any available port while socket uses a well-known port so that users know which port to access the server on. In conclusion, sendSocket is used for the connection with a client and socket just for receiving the first read or write request.

# Problem 2:

**Multiple Read Requests from Client with files greater than 512 Bytes (Screenshot):**

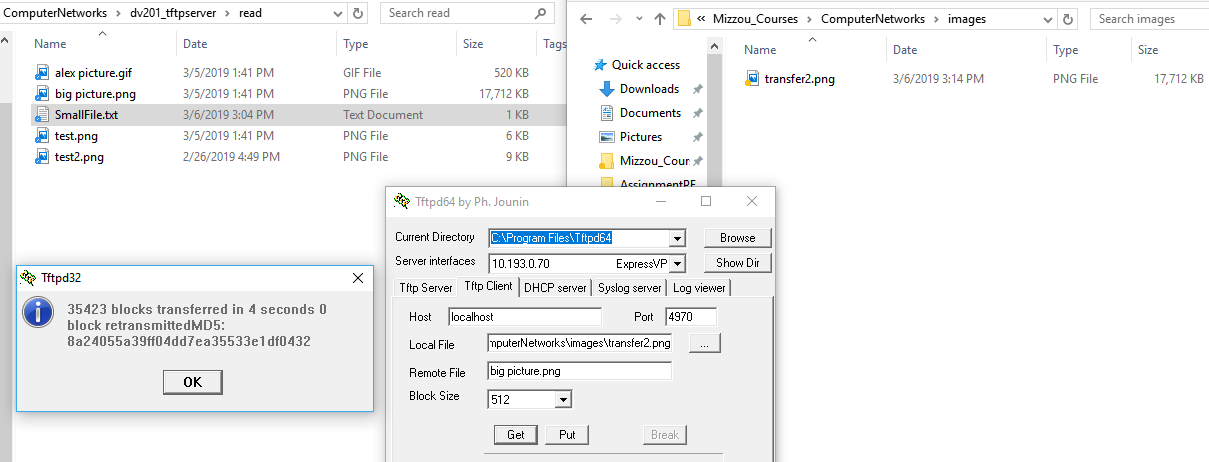
**Large Transfer 1:**

This test was performed in the same way as the test for problem 1 but this time with larger files. As you can see in the confirmation screenshot, for the file “alex picture.gif”, which is 520 KB in size, there were 1040 blocks transferred. The completed transfer to the local system is shown in the right explorer window. You can see that the size of the file on the server and on the local filesystem is the same.



**Large Transfer 2:**

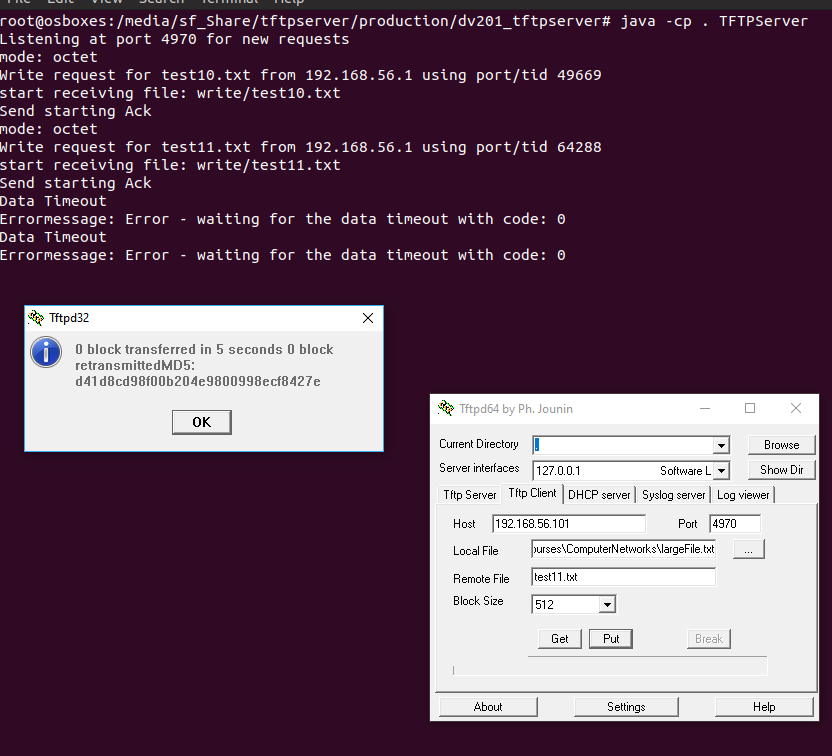
As you can see in the confirmation screenshot, for the file “big picture.png”, which is 17,712 KB in size, there were 35423 blocks transferred. This transfer took 4 seconds because of how large the image was but the server was still capable of handling the transfer. The completed transfer to the local system is shown in the right explorer window. You can see that the size of the file on the server and on the local filesystem is the same.



Timeouts and Retransmissions:

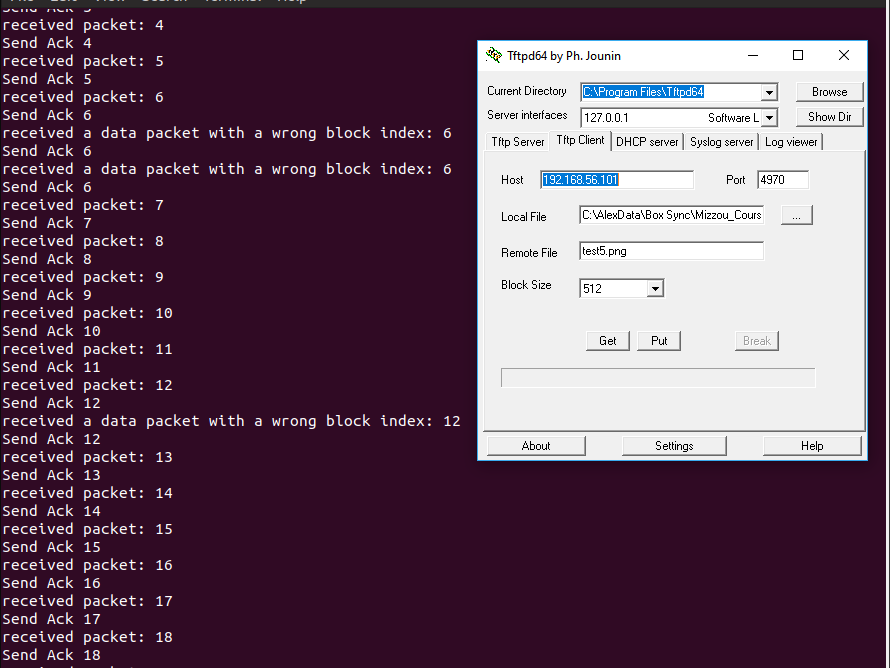
Our server uses two different timeouts, one for the receiving and one for the sending of files. In case we send a file to the client we use a smaller timeout and try to resend the data 5 times (resend in case of timeout and of if an ack with a wrong data number arrives). After 5 successive times we send an error message and abort the file transfer.  
When we receive a file from the client, our server uses a long timeout after sending an ack. After that timeout we send an error message and abort the file transfer. We do not resend the ack because the client should resend his data packet by itself, when it checks that it did not get an ack for a packet. In order to do this our timeout is so long for that case.

### Receiving Data Timeout:



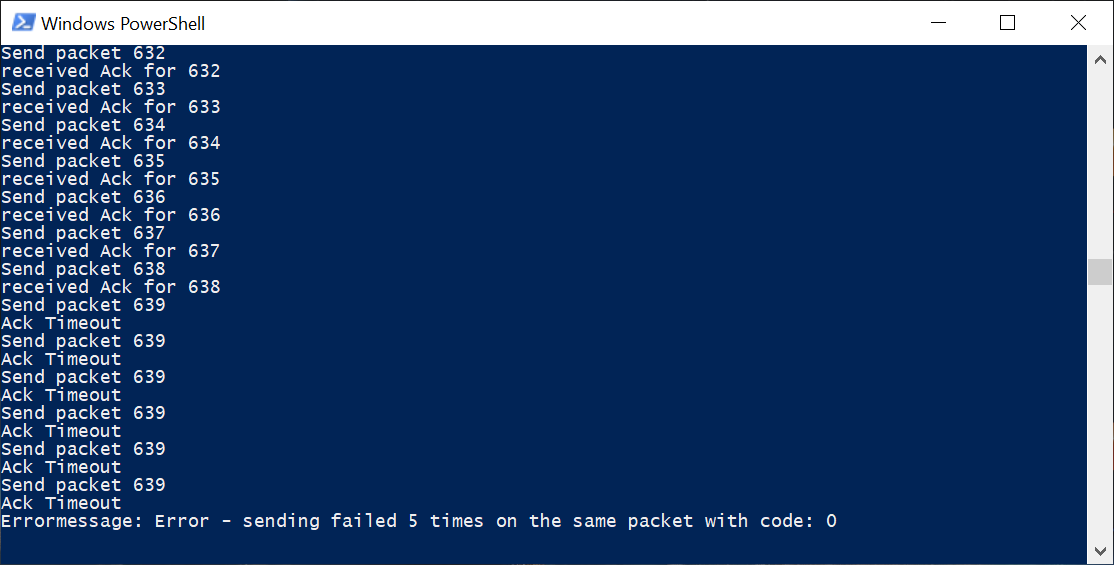
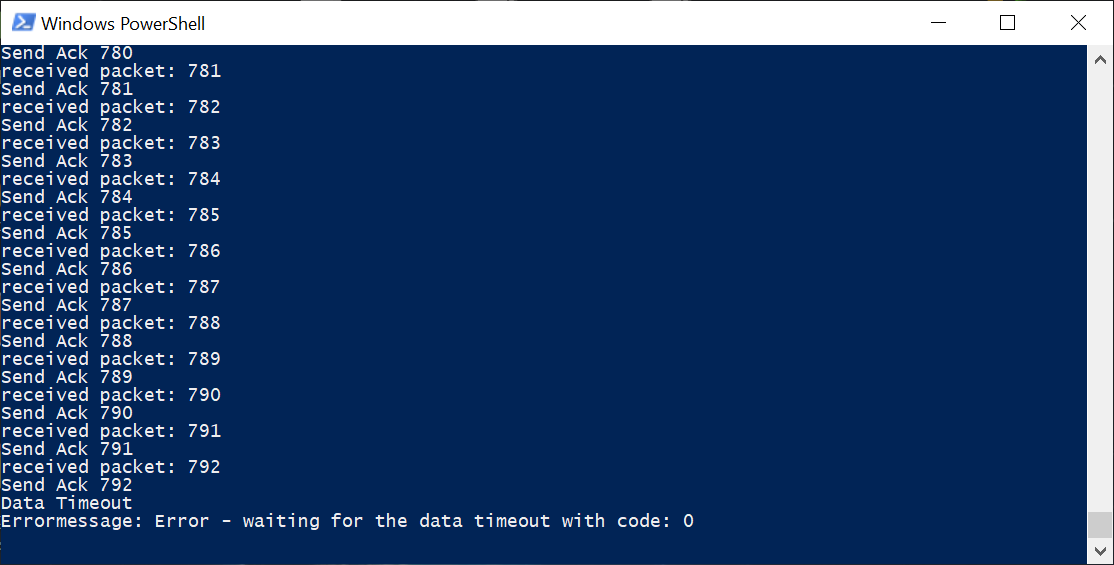
### Sporadic Packet Loss – timeout and retransmission:

To test the timeouts and retransmissions we added sporadic packet loss to incoming requests into the server using the following UNIX command ( tc qdisc add dev enp0s8 root netem loss 10%). This command drops 10% of packets on the selected network adapter. This means that somewhere in the transfer packets will be lost randomly. The screenshot below shows the terminal of the VM where the server is running. For example, look at block 12. The client is waiting for an ACK for block 12 from the server but the ACK block is lost because of the packet loss. The client will resend the data message and then the server shows “received a data packet with a wrong block index” because the server already received that data, the client just did not get the ack for it. After this resending our server sends an ack for block 12 again which arrives at the client, so the transfer continues normally. This shows our server is able to handle sporadic packet loss.



### No endless re-transmissions

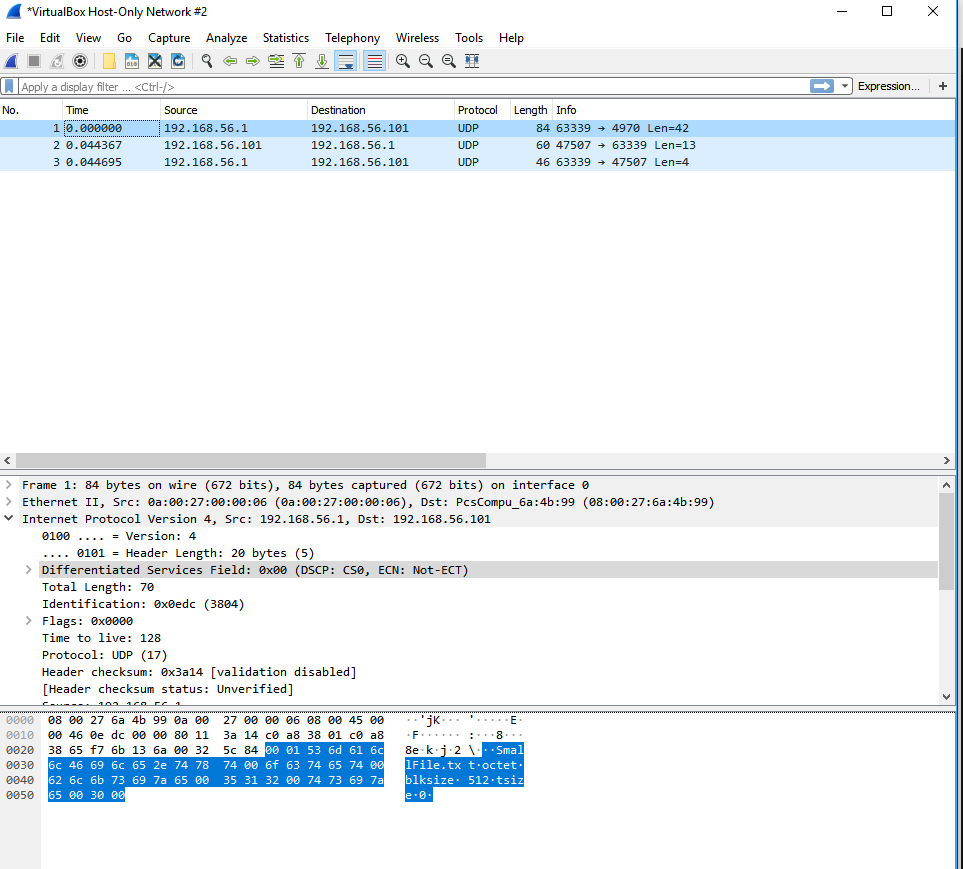
We should make sure that our sever does not end up in a endless retransmission. To test this we started a transmission and then just closed the client program. Once for sending, once for receiving a file.

Sending:Receiving:

## VG-Task 1:

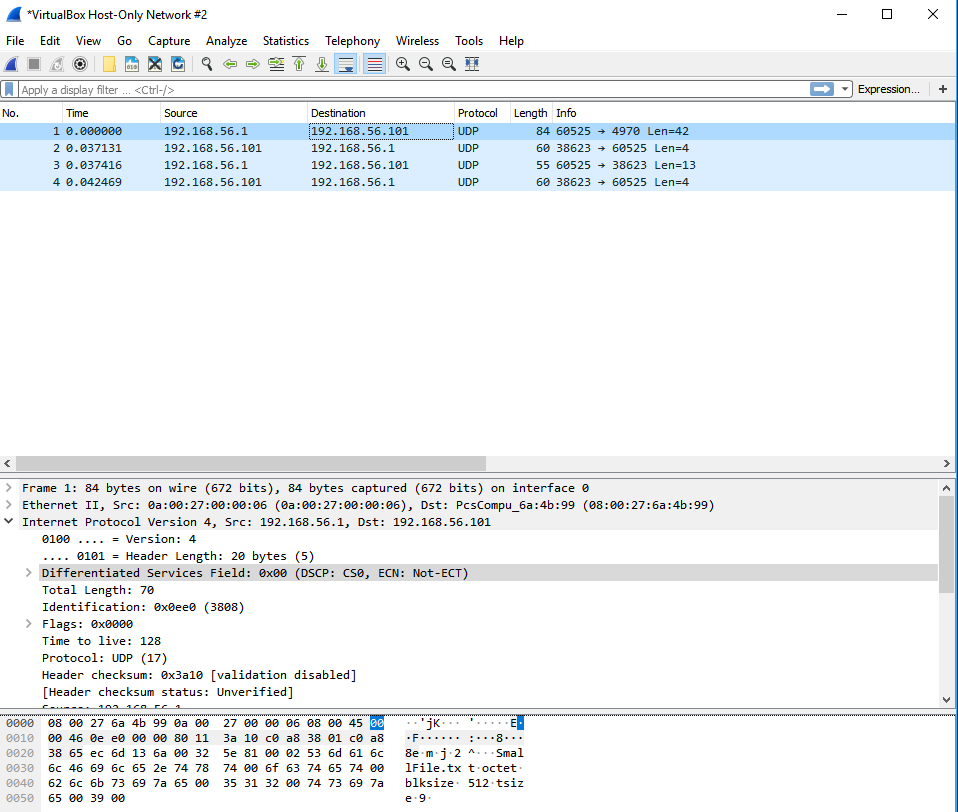
### Read Request:

For this test a read request was preformed to the server running on a virtual machine with the IP address 192.168.56.101. The client (my computer) is issuing a read request to the server in line 1 of the Wireshark screenshot. This request includes the file name, a 0-Byte, and stream mode (octet in this case). After this we can see additional information that is not specified by the RFC. This is using the UDP protocol so there is no handshake like TCP. Shown with the red circle below, the read request uses OP Code 1 to tell the TFTP server that the request is a read request. Then in line 2 the server sends the binary contents of the file to the client using a different sending port. This port will now be used for the following connection. The message starts with two bytes for the opcode showing 3, followed be two bytes showing the block number that is 1, because this is the first data packet for this file. After that you can find the binary data that is that. It is binary because of the mode “octet”. The size of the packet shows that this packet is the last one of the file transferred. In line 3 the client sends an acknowledgment (ACK) message (OP Code 4) that it has received the data successfully. The ack message contains the opcode for acknowledgement 4 and 2 blocks with the block number that is acknowledged, so 1.



### Write Request:

For this test a write request was performed to the server running on a virtual machine with the IP address 192.168.56.101. The client (my computer) is issuing a write request to the server in line 1 of the Wireshark screenshot. This request includes the file name followed by an 0-byte, and stream mode (octet in this case), followed by a 0-byte, both in netascii and block size (not part of RFC). This is using the UDP protocol so there is no handshake like TCP. Shown with the red circle below, the write request uses OP Code 2 to tell the TFTP server that the request is a write request. In line 2, the server sends an Acknowledgment (ACK) message to the client that it has received and accept the request. It is built the same way I described above but the block number is 0 because it is the starting ack for a write request. The server uses another port than 4970 to send this ack. This is the port that is now used for that connection. Then in line 3, the client transmits the binary data of the file to the server. The message is built the same way I described above. The size of this data packet indicates that this is the last data packet. In line 4, the server sends an ACK message to the client that it has received and written the data to the specified file. The ack contains the ack opcode and the block number 1 as usual.



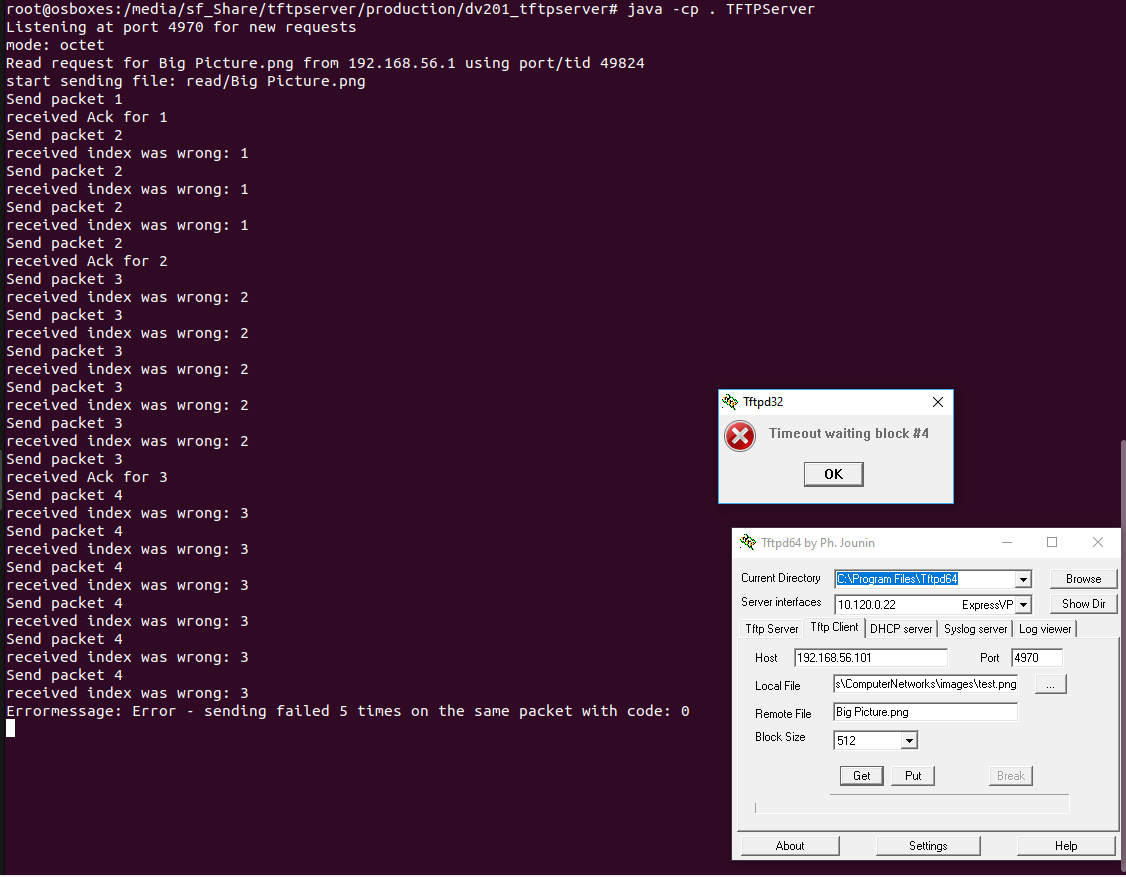
# Problem 3 (Error Handling):

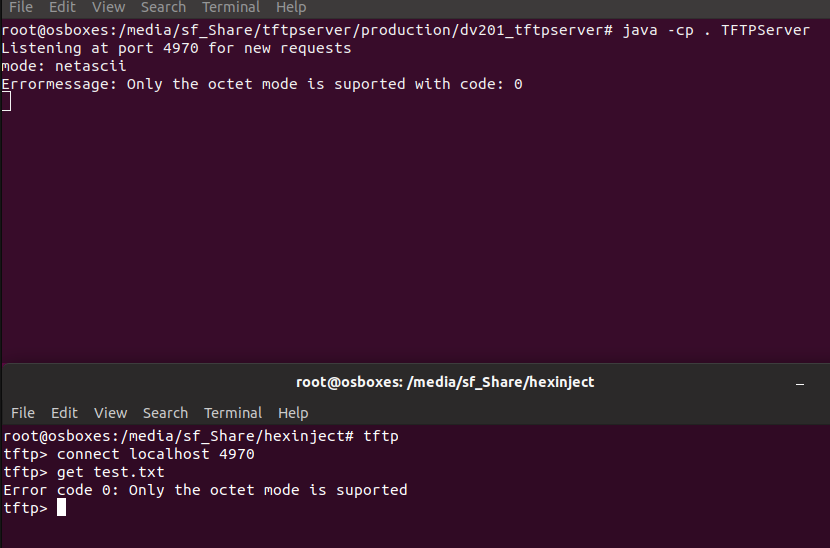
In the following screenshots the terminal window is the server running on a virtual machine

## Code 0:

According to the RFC 1350 spec the Code 0 Error code indicates an undefined error and you should check the error message if there is any. Our server can send a Code 0 for a few conditions. In the first screenshot below the server attempted to retransmit the same packet 5 times without receiving an acknowledgement and after not receiving an ACK message the 5th time it sent the Code 0 error message and aborts the connection. In the 2nd screenshot the client program (bottom terminal) tries to get a file using a mode other than the octet mode and the server (top terminal) returns a code 0 error. Our program will also send a code 0 on the following conditions, among other things:

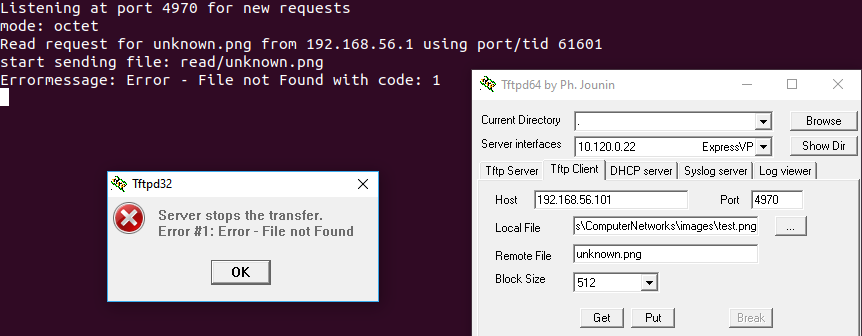
* Sending or Receiving a file larger than 32 MB
* Unknown Errors/Failures
* Timeout waiting Errors
* Errors receiving incoming data





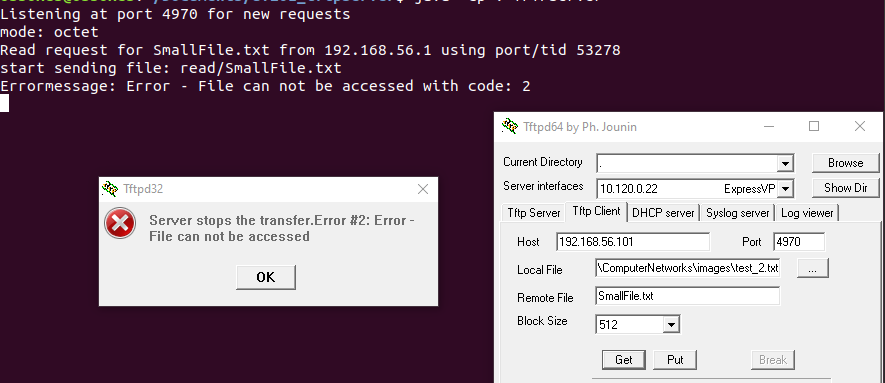
## Code 1:

According to the RFC 1350 spec the Code 1 Error code indicates that the File is not Found. Our server will send this error code if the client user requests a file that does not exist on the server. For this test the client user requested “unknown.png”. The server could not find the file and send the Code 1: File not found error to the client.



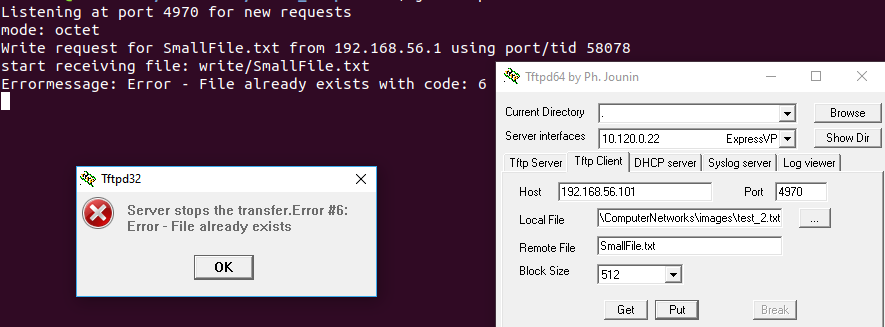
## Code 2:

According to the RFC 1350 spec the Code 2 Error code indicates that there is an access violation. For this test I changed the permissions of the file “SmallFile.txt” using “chmod a-r” which makes it so that no one user can read the file. This means when the server attempts to access the file it will cause an access violation and send a Code 2 error as shown by the error message window.



## Code 6:

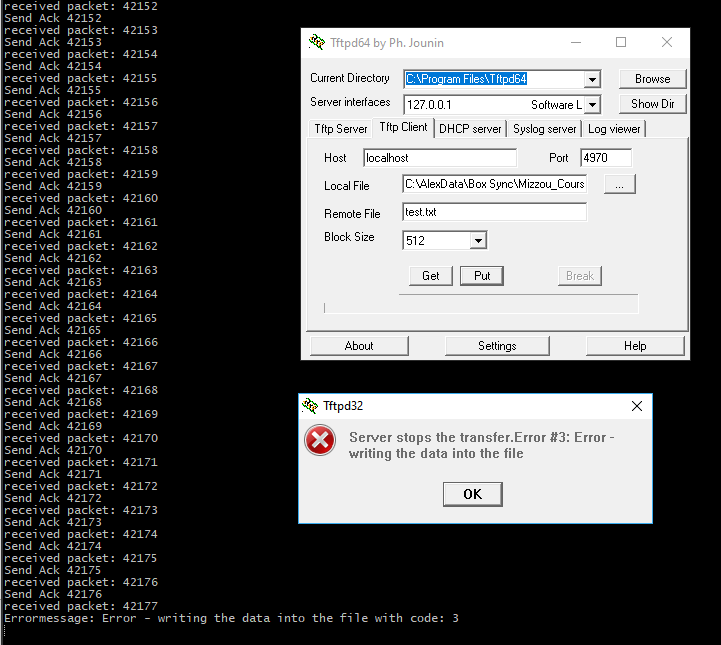
According to the RFC 1350 spec the Code 6 Error code indicates that the File Already Exists. Our server will send this error code if the file that the client user is trying to “Put” on the server already exists. In this case the file “SmallFile.txt” already exists on the server so the server will not allow it to be overwritten and the server sends a code 6 error message.



## VG-task 2:

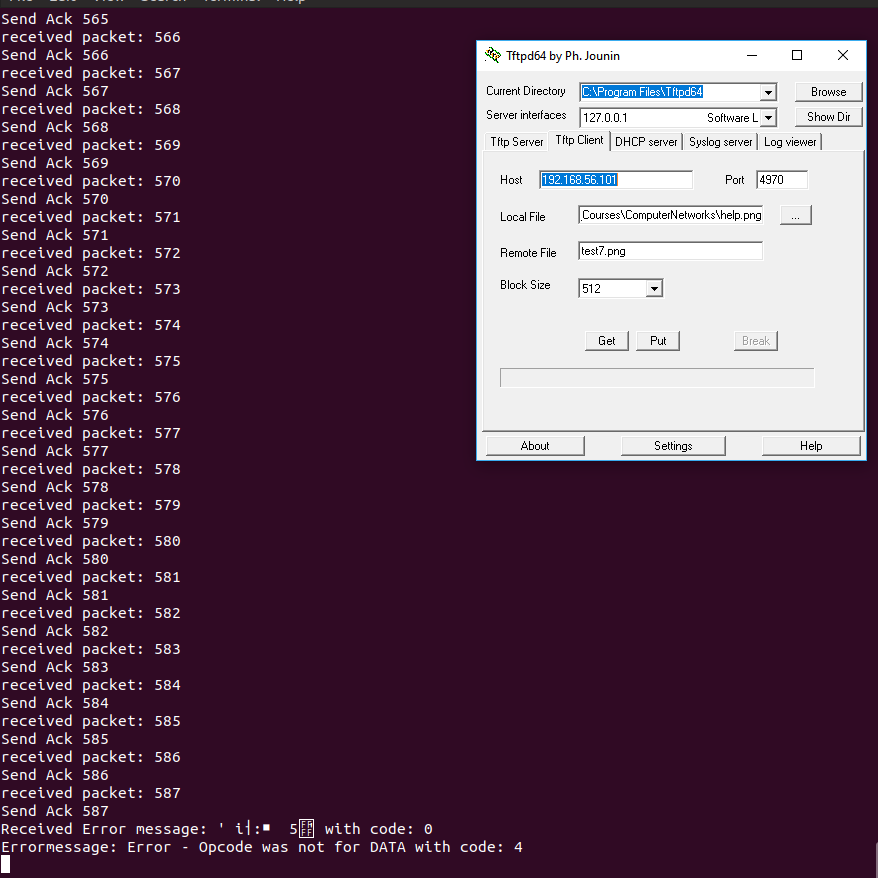
### Code 3:

According to the RFC 1350 spec the Code 3 Error code indicates that the disk is full or the allocation is exceeded. Our server will send this code if there is a problem writing the file to the disk. This could be another problem than disk full, but we are allowed to implement it like this. This was tested by generating a dummy file that nearly filled a USB drive. Then the transfer was attempted, and the server returned the code 3 error when there was no more space left on the disk.



### Code 4:

According to the RFC 1350 spec the Code 4 Error code indicates an illegal TFTP operation. Our server will return this error code if the client uses an invalid OP code (not read or write) to start a transmission. It will also return this error code if it receives a packet that should be a specific Op-Code and it is something different. For example, every error message from the client or if the server waits for data and gets for example a acknowledgement.



### Code 5:

According to the RFC 1350 spec the Code 5 Error code indicates an unknown transfer id. After a connection is established with a Read or Write Request, the further connection is handled about a random port as described above. This server port is than fixed to a specific client port (according to the RFC, it is only fixed to a port number and not to the combination port and IP of the client). If a UDP message arrives to this server port that is not from that specific client port (called TID in TFTP), the server does not handle this message and sends the error code 5 to the wrong client.