

# Computer networks - 1DV701

## Assignment 3

*Author:* Michael Johansson & Jakob  
Heyder  
*Semester:* VT 2017

## **Contents**

<b>1</b>	<b>Assignment summary</b>	<b>1</b>
<b>2</b>	<b>Problem one</b>	<b>2</b>
<b>3</b>	<b>Problem two</b>	<b>3</b>
3.1	VG.1 wireshark analyze . . . . .	4
<b>4</b>	<b>Problem three</b>	<b>7</b>
4.1	VG.2 Remaining error codes . . . . .	9
	<b>References</b>	<b>11</b>

## **1 Assignment summary**

- **Michael** Work percentage 50%
- **Jakob** Work percentage 50%

## 2 Problem one

In figure 2.1 we show a successful RRQ (Read-Request) to the implemented tftp server retrieving a small test-text file.

We use the original socket as a kind of server socket accepting the requests on port 4970 (Usually 69 in TFTP) and then after receiving a request, either write or read, we open a new thread for the specific client connection. In this thread we use the sending socket which acquires a free port (zero port indicates to get a random free port e.g. 5800) and then send the new specified port to the client in the ACK (WRQ) or DATA (RRQ) packet. This is defined in the TFTP-Specification that the initialization will be on a predefined port and then the server communicates the used port to the client. This way we can also handle multiple client requests since the original socket is freed after handling the client in a separate thread. Also the send socket connects to the remote port and address of the client given from the request.

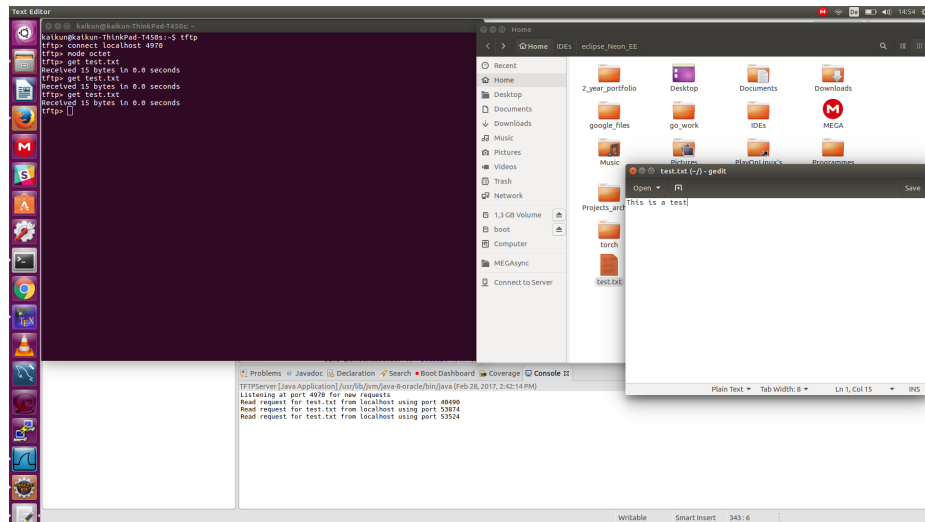


Figure 2.1: Successful READ-Request. We can see the tftp-client in the terminal window, eclipse tftp server and the home directory where the retrieved file is saved to.

### 3 Problem two

In figure 3.1 we send and receive multiple large files. The task says sending but it is not clear if it means sends from client or server. However both is on the screenshot with files larger then 512 bytes. The case of files which are so large that they exceed the package number of a short is thought of but not mentioned in the specification. Therefore its left out and assumed out of scope in this assignment.

We implemented a timeout mechanism that the sending socket times out if it does not receive the next DATA/ACK packet in 150ms. If it times out it will try to retransmit or simply tries to receive again. We have a re-transmission counter which will be increased and stops the execution after 5 retransmissions/waiting-times so that no infinite loop will occur.

The execution of a write-request can be seen also on figure 3.1.

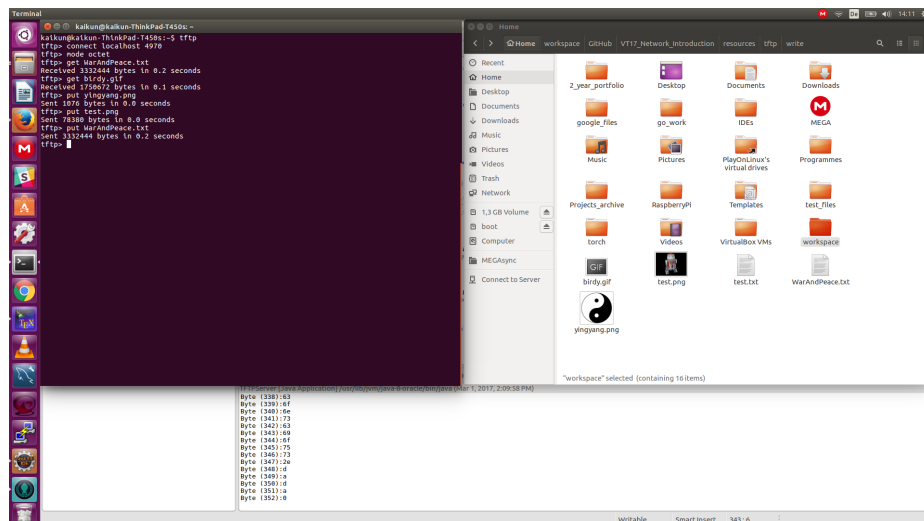


Figure 3.1: Read and write multiple large files. On the left hand side we see the iftp client transferring files from the home directory or reading files from the server. We can see the bytes transferred.

### 3.1 VG.1 wireshark analyze

In figure 3.2 we can see the line marked where the client, in this case with the localhost - 127.0.0.1 - src-address from the randomly choosen free port 53874 is sending a UDP Packet to the server. The server address is also localhost - 127.0.0.1 - dest-address and the port is the chosen 4970, which is usually 69 for the TFTP protocols. In the bottom rows we can see the content of the package and beside all the headers of the lower layers the data of the UDP packet is marked. In the data we can see it start with the 2Bytes 00 01 which is defined in the TFTP-Specification as the operational code for a read request. The next bytes, in this case 8, until a zero byte are defined as the filename string. After the zero byte the mode will be defined by the following bytes closed with a zero byte again. In this case we can see that the OP-Code is RRQ, the filename of the requested file is test.txt and the mode is octet.

In figure 3.3 we can see the line marked where the server answers to the clients read request. We see that the destination and source address are again localhost, but would have switched. The switch we can identify on the ports, while the client port 53874 is now the receiving port the sending port from the server changed to 53964. This tells the client that this will be the port on the server side used for the rest of the file transmission. The initial port 4970 is only used to handle new requests. In the bottom rows we can see the data send back from the server. It starts with the 2Bytes 00 03 which is the OP-Code for DATA packages recording to the TFTP-Specification. The next two bytes 00 01 indicate the package number. In this case it is the first data package sent. It follows the actual file-data. In this case it is a text file and simply transfers the ASCII-Characters saved in it, "This is a test.".

In figure 3.4 we can see the line marked where the client sends an acknowledgement back to the server that it received the data package. We can see that the source and destination port flipped again and that the client now sends to the server port it received the data package from. The client sends only 4 Bytes data back. The first two indicating the OP-Code again in this case 00 04, meaning it is an acknowledgment packet and 2 more bytes 00 01 referencing the package number of the received packet which gets acknowledged. Due to the fact that the received data package was less than 516 bytes (512 inc. 4bytes header data) the client knows that the file is complete and with sending the ACK-Packet back the connection is terminated normally.



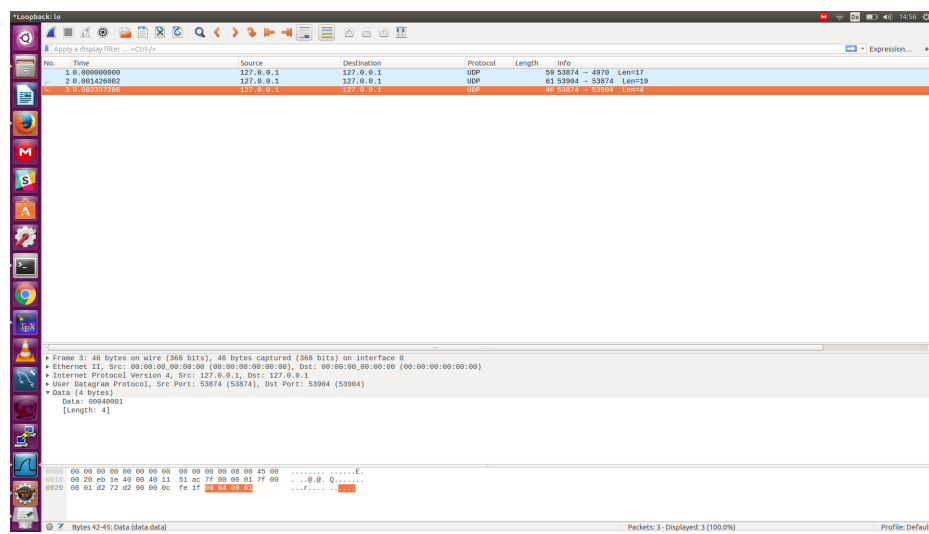


Figure 3.4: We see a screen of the acknowledgement of the TFTP-client to the received data file of the TFTP-Server. The data is again marked in the bottom rows.



## 4 Problem three

In the figure 4.1 we can see how our server sends an error packet for when the connection times out. The first two bytes is 05 to show that is an error packet. Then the next two bytes is the error code, for this packet the error code is 0. This code has the meaning "Not defined, see error message (if any)."[1]. After this error code there is a string with a message, this is then terminated with a zero byte at the end as figure 4.2 shows.

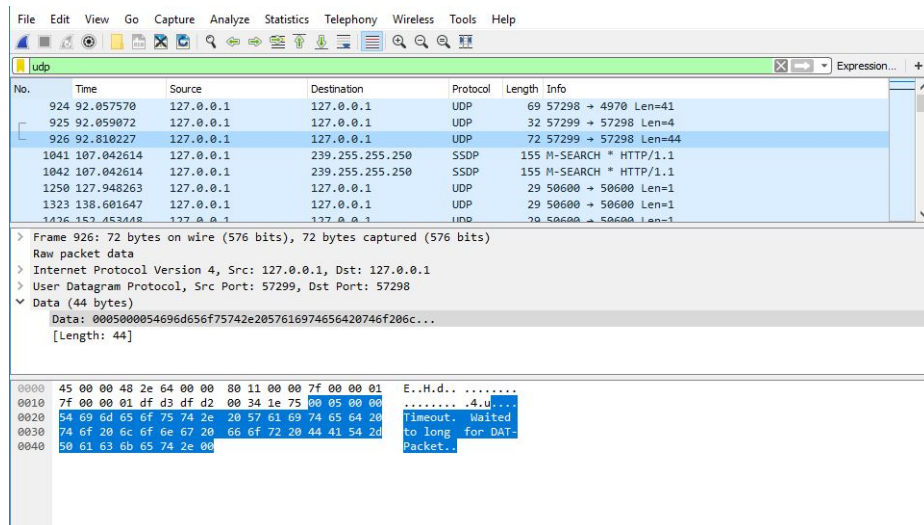


Figure 4.1

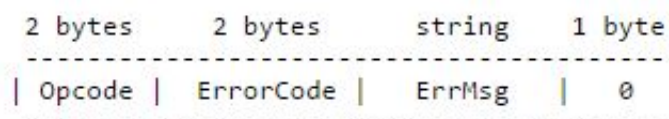


Figure 5-4: ERROR packet

Figure 4.2

As we see in figure 4.3 the error codes look quite the same, all that changes are the error code bytes(0-7) and what message we send with the packet. This Error number 1 is for when the server can't find the file the client asks for, An File not found error.

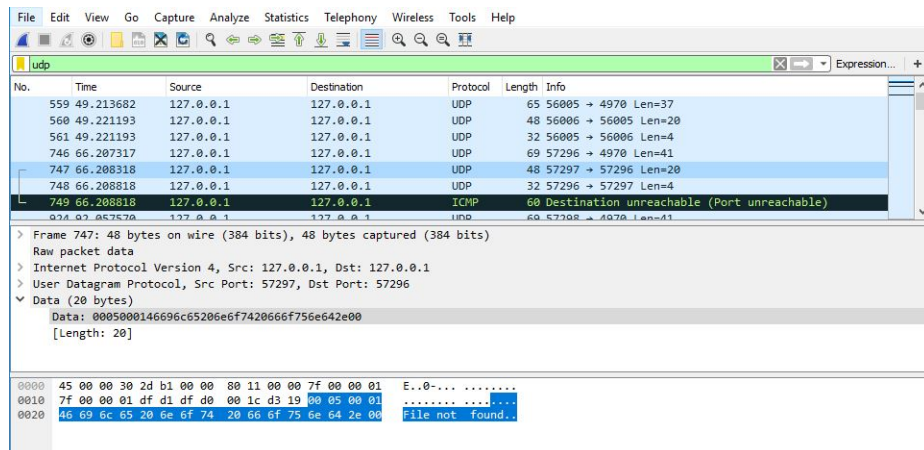


Figure 4.3

The error code number 2 in figure 4.4 is when there is some access violation on a file that the server tries to open/write/read to. I simulated this error on the write folder in our server, when i removed the permission to write to this folder the server could not write the file the client wanted to PUT. As we don't know how large the file will be we are saving the data into a buffer before we create a file, so this error gets sent only when we have read all the data and then tries to write this to a file.

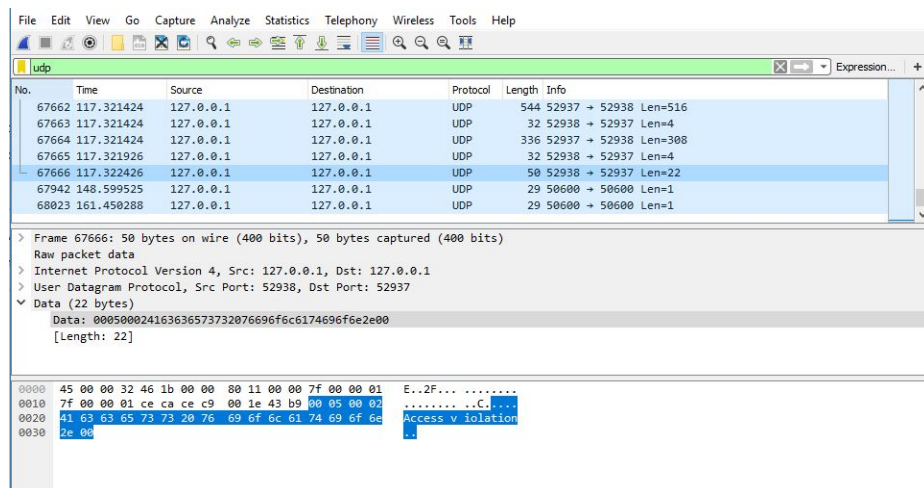


Figure 4.4

Figure 4.5 shows the error code 6 which is sent when a client tries to PUT a file that are already present in the Write directory of the server. An File already exist error packet.

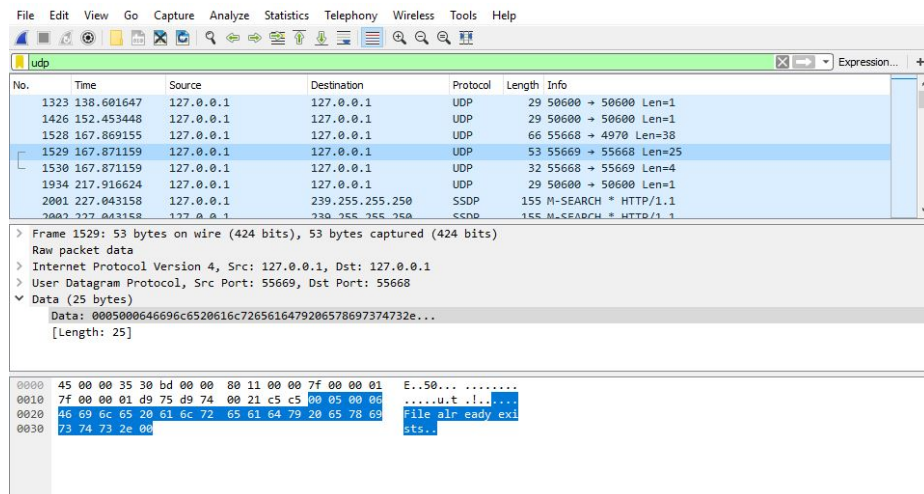


Figure 4.5

#### 4.1 VG.2 Remaining error codes

Figure 4.6 shows our error packet for if the disk or in our case folder is full. We simulated this by setting an max size for our folder of 10MB then we have a method that checks the free space each time we are going to PUT a file. If the folder would be more the 10MB after we added the file, instead of adding the file we send an error packet with code 3 "Disk full or allocation exceeded." [1].

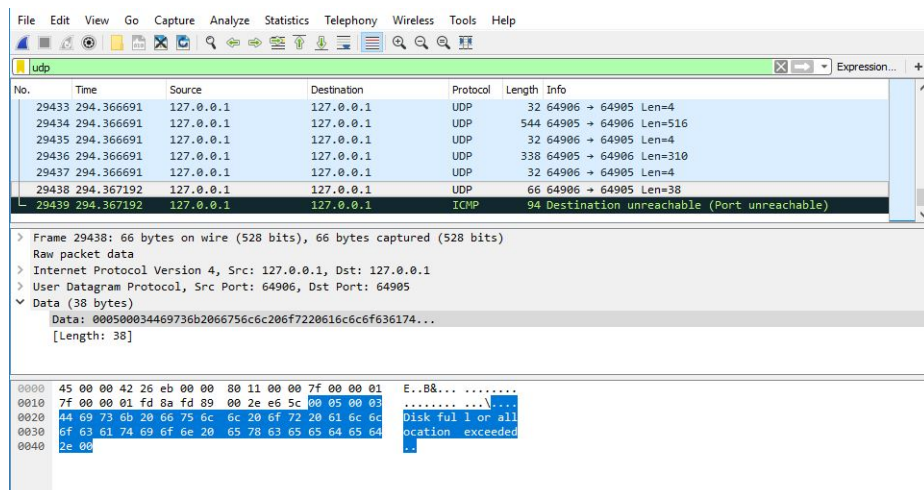
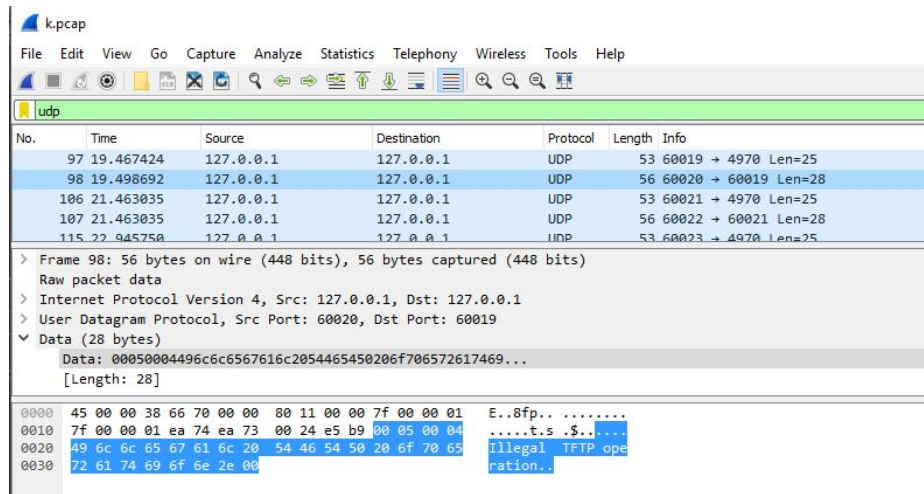


Figure 4.6

We see i figure 4.7 that we send an error packet with code 4 when the client

tries to do an illegal TFTP operation. The only operations that are legal to use are GET/PUT.



The image shows a Wireshark capture of a UDP packet. The packet list shows four packets from 127.0.0.1 to 127.0.0.1 on ports 60019, 60020, 60021, and 60022. Packet 98 is selected, showing details for Internet Protocol Version 4 and User Datagram Protocol. The raw packet data is displayed in hexadecimal and ASCII. The ASCII column shows the text "Illegal TFTP operation..".

No.	Time	Source	Destination	Protocol	Length	Info
97	19.467424	127.0.0.1	127.0.0.1	UDP	53	60019 → 4970 Len=25
98	19.498692	127.0.0.1	127.0.0.1	UDP	56	60020 → 60019 Len=28
106	21.463035	127.0.0.1	127.0.0.1	UDP	53	60021 → 4970 Len=25
107	21.463035	127.0.0.1	127.0.0.1	UDP	56	60022 → 60021 Len=28
115	22.945750	127.0.0.1	127.0.0.1	UDP	53	60023 → 4970 Len=25

Frame 98: 56 bytes on wire (448 bits), 56 bytes captured (448 bits)  
Raw packet data  
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1  
> User Datagram Protocol, Src Port: 60020, Dst Port: 60019  
Data (28 bytes)  
Data: 00050004496c6c6567616c2054465450206f706572617469...  
[Length: 28]

Offset	Hex	ASCII
0000	45 00 00 38 66 70 00 00 80 11 00 00 7f 00 00 01	E..8fp.. .....
0010	7f 00 00 01 ea 74 ea 73 00 24 e5 b9 00 05 00 04	.....t.s\$. ....
0020	49 6c 6c 65 67 61 6c 20 54 46 54 50 20 6f 70 65	Illegal TFTP operation..
0030	72 61 74 69 6f 6e 2e 00	

Figure 4.7

Error 5 can we see in our implementation, this is shown in figure 4.8. As this is quite hard to simulate we have no wireshark capture of it. But how it would work is that if the server gets a packet with a port number we didn't expect we send back an "Unknown transfer ID." [1] error packet. This is also the only error that should not terminate the connection. So when we get this we send back an error but we keep sending to the original port.

```
// ERROR: transferId changed => error message sent to other port but connection maintained
if (receivePacket.getPort() != sendSocket.getPort()){
    DatagramSocket invalidPort = new DatagramSocket( port: 0);
    invalidPort.connect(new InetSocketAddress(sendSocket.getInetAddress(), receivePacket.getPort()));
    send_ERR(invalidPort, errCode: 5, errMessage: "Unknown transfer ID.");
}
```

Figure 4.8

## References

- [1] “RFC 1350 - the tftp protocol,” <https://tools.ietf.org/html/rfc1350>, accessed: 02-03-2017.