Linnéuniversitetet
Kalmar Växjö

Report

1DV701 - Assignment 3

TFTP Server

Author: Eric Enoksson
Tutor: Morgan Ericsson
Semester: Spring 19
Course: Computer Networks
Course code: 1DV701

Table of contents

Introduction **3**

Problem 1 **3**

Problem 2 **4**

RRQ 4

WRQ 5

VG-task 1 7

Differences between read and write requests in tftp 8

Problem 3 **8**

Error code 0: Not defined, see error message (if any) 8

Error code 1: File not found 9

Error code 2: Access violation 9

Error code 6: File already exists 9

VG-task 2 9

Error code 3: Disk full or allocation exceeded 10

Error code 4: Illegal TFTP operation 10

Error code 5: Unknown Transfer ID 10

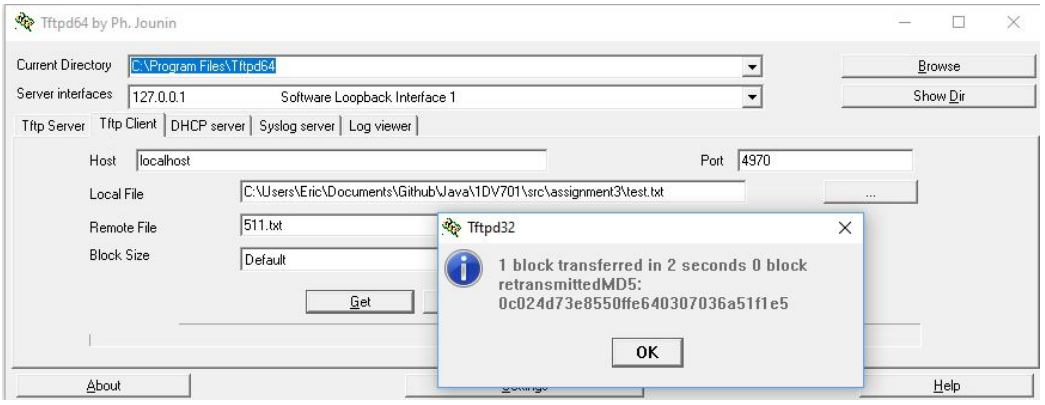
Introduction

The server is started by running the main method in the TFTPServer class.

A short description of the classes, exceptions excluded:
TFTPServer - listens on specified port for incoming requests. Creates a ServerThread object when a request is received.
ServerThread - calls appropriate methods and deals with errors.
ErrorHandler - contains a single method that sends errors to a host.
RequestHandler - handles request parsing, and read/write requests.
TransferHandler - sends and receives data and acknowledgements.

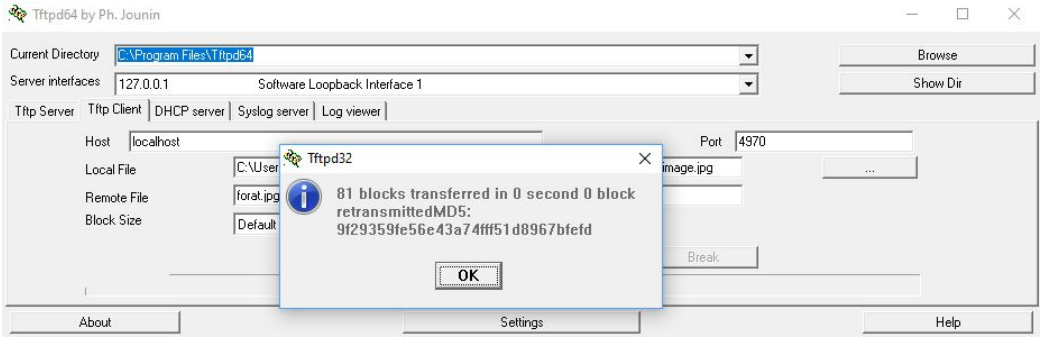
Class hierarchy overview
TFTPServer
 |- uses ServerThread
  |- uses ErrorHandler
  |- uses RequestHandler
   |- uses TransferHandler

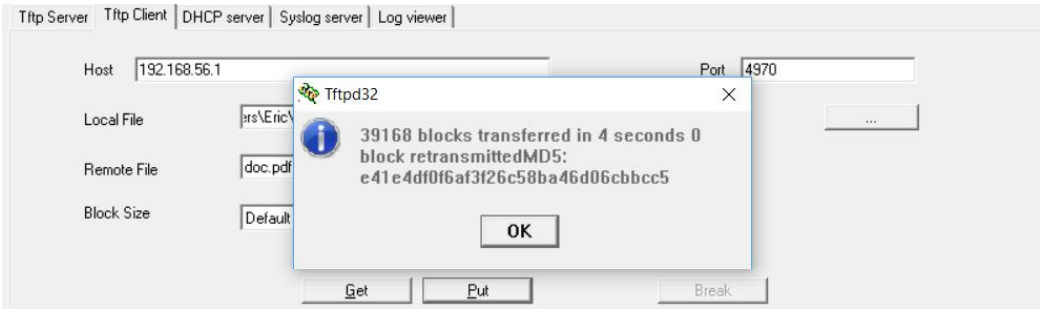
Problem 1

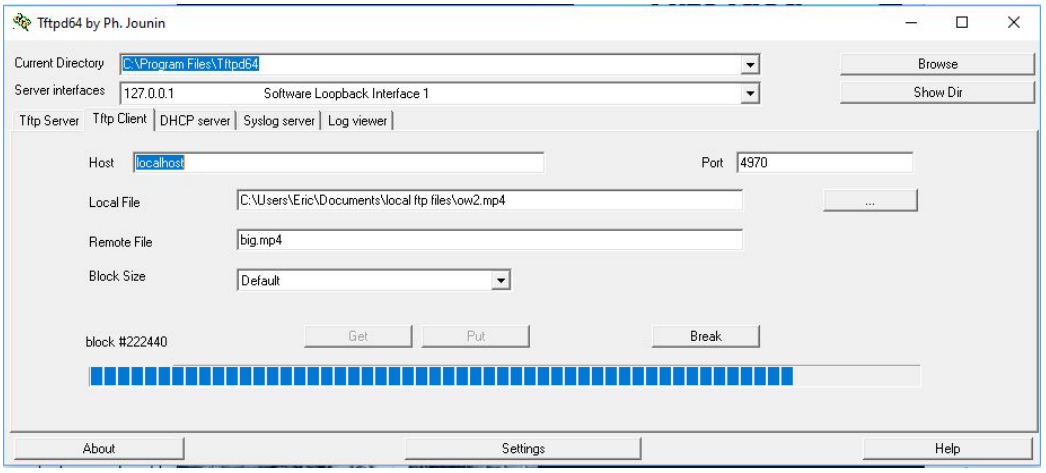


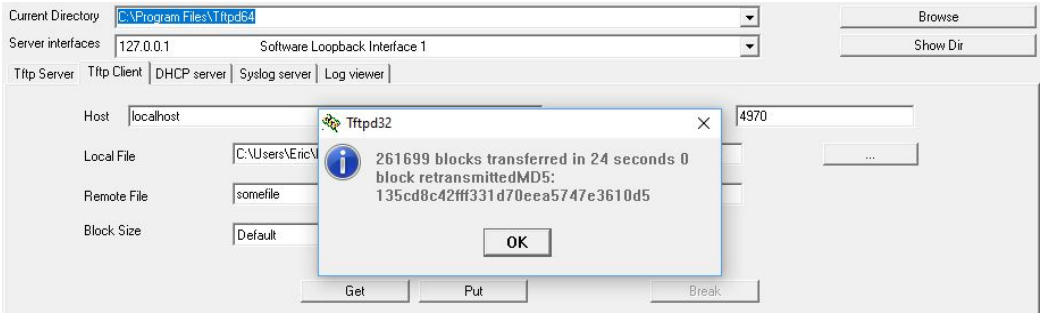
The variable “socket” is used to receive requests from clients. A client can only contact the server on a known open port (the listening port). In the request packet, the source transfer identifier (TID) refers to a free port on the client side and destination TID refers to the listening port on the server, in this case 4970. The server creates a new socket (sendSocket), using the source TID from the request as its destination TID, but instead of using the listening port as its source port, the server assigns a free port dedicated to that specific connection, in order to keep the listening port open for other requests. The sendSocket is the one used for the file transfer.

Problem 2









RRQ

Timeouts were tested by simply disabling the line of code sending the data packet from the server. To make the functionality transparent to myself, I added console prints (though it says it's sending packets, it's not in this case since sending is disabled).

```
Listening at port 4970 for new requests
host: 0:0:0:0:0:0:0:1
Read request for 0:0:0:0:0:0:0:1 using port 56757
prepared to send block 1: 512 B
sending packet...
packet sent. waiting on ack. expecting block 1
Attempt 1: Ack receive timed out. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 1
Attempt 2: Ack receive timed out. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 1
Attempt 3: Ack receive timed out. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 1
Attempt 4: Ack receive timed out. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 1
Attempt 5: Ack receive timed out. Maximum re-transmit attempts reached. Aborting file transfer
--TRANSFER FAILURE
```

To test the acknowledgements, I added a random int 0-1 to the block of the received ack packet (maybe it would've made more sense to subtract it, but I don't think it matters). This to cause the check to fail about half of the time.

```
prepared to send block 11: 512 B
sending packet...
packet sent. waiting on ack. expecting block 11
Attempt 1: Received block is not equal to expected block. Received 12, expected 11. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 11
Attempt 2: Received block is not equal to expected block. Received 12, expected 11. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 11
Attempt 3: Received block is not equal to expected block. Received 12, expected 11. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 11
ACK RECEIVED. BLOCK=11

prepared to send block 12: 512 B
sending packet...
packet sent. waiting on ack. expecting block 12
ACK RECEIVED. BLOCK=12

prepared to send block 13: 512 B
sending packet...
packet sent. waiting on ack. expecting block 13
Attempt 1: Received block is not equal to expected block. Received 14, expected 13. Re-sending packet
sending packet...
packet sent. waiting on ack. expecting block 13
ACK RECEIVED. BLOCK=13
```

WRQ

I tested the timeout functionality for write requests by making it send only if a randomized integer became 0.

```
if(new Random().nextInt(3) == 0) {
```

The way I tested RRQ timeouts would have worked as well, but this made the functionality clearer.

I added some console prints this time as well to be able to follow it easily.

```
waiting on data
RECEIVED 516 B. BLOCK=43
sending ack. block 43
----
waiting on data
RECEIVED 516 B. BLOCK=43
sending ack. block 43
ACK actually sent
----
waiting on data
RECEIVED 516 B. BLOCK=44
sending ack. block 44
ACK actually sent
----
```

"ACK actually sent" is printed when the if statement was true.

Even though re-sending during a WRQ is mostly a client side thing, I tested what happens if you send an acknowledgment of the previous packet (i.e. the last packet sent from the client was lost or the previous server acknowledgement was lost). I did so by randomly removing 0-1 from the received block and sent an acknowledgement based on that.

This could only really be a problem if the server did not use the block from the received packet to base its acknowledgement on. However, we can see it's working well.

```
receivedBlock = (short) (receivedBlock - new Random().nextInt(2));
```

```
----
waiting on data
RECEIVED 516 B. BLOCK=330
sending ack. block 330
----
waiting on data
RECEIVED 516 B. BLOCK=331
sending ack. block 330
----
waiting on data
RECEIVED 516 B. BLOCK=331
sending ack. block 331
----
waiting on data
RECEIVED 516 B. BLOCK=332
sending ack. block 331
```

I did have some problems with timeouts. Since if the client does not receive an ACK packet, the client will wait for a certain amount of time, decided by the client itself. At

the same time, the server will after sending the ACK packet, start listening again immediately, assuming the packet received wasn't the last one (< 516 B). If you then have a lower timeout value (or roughly the same) than on the client, the server receive() can timeout before the old packet is re-transmitted from the client.

VG-task 1

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.56.3	192.168.56.1	UDP	60	51745 → 4970 Len=16
2	0.003590	192.168.56.1	192.168.56.3	NBNS	92	Name query NBSTAT *<00><00><00><00><00><00><00><00><00><00><00><00><00><00><00><00>
3	0.003945	192.168.56.3	192.168.56.1	ICMP	120	Destination unreachable (Port unreachable)
4	0.004271	fe80::6da9:3da0:e2f…	ff02::1:3	LLMNR	105	Standard query 0x62e3 PTR 3.56.168.192.in-addr.arpa
5	0.004528	192.168.56.1	224.0.0.252	LLMNR	85	Standard query 0x62e3 PTR 3.56.168.192.in-addr.arpa
6	0.415442	fe80::6da9:3da0:e2f…	ff02::1:3	LLMNR	105	Standard query 0x62e3 PTR 3.56.168.192.in-addr.arpa
7	0.415608	192.168.56.1	224.0.0.252	LLMNR	85	Standard query 0x62e3 PTR 3.56.168.192.in-addr.arpa
8	1.503596	192.168.56.1	192.168.56.3	NBNS	92	Name query NBSTAT *<00><00><00><00><00><00><00><00><00><00><00><00><00><00><00><00>
9	1.503912	192.168.56.3	192.168.56.1	ICMP	120	Destination unreachable (Port unreachable)
10	3.003830	192.168.56.1	192.168.56.3	NBNS	92	Name query NBSTAT *<00><00><00><00><00><00><00><00><00><00><00><00><00><00><00><00>
11	3.004482	192.168.56.3	192.168.56.1	ICMP	120	Destination unreachable (Port unreachable)
12	4.546607	192.168.56.1	192.168.56.3	UDP	558	57547 → 51745 Len=516
13	4.547135	192.168.56.3	192.168.56.1	UDP	60	51745 → 57547 Len=4
14	4.547357	192.168.56.1	192.168.56.3	UDP	47	57547 → 51745 Len=5
15	4.547748	192.168.56.3	192.168.56.1	UDP	60	51745 → 57547 Len=4

1: 192.168.56.3 (the client) sends a read request to 192.168.56.1 (the server) from port 51745 to known server port 4970. The UDP packet length is 16. 2 bytes for opcode, 2 bytes for the two 0-byte separators, and in this case 12 for filename and mode.

```
Data (16 bytes)
    Data: 00013531332e747874006f6374657400
    [Length: 16]

0000  0a 00 27 00 00 0e 08 00  27 4c f9 33 08 00 45 00   ··'····· 'L·3··E·
0010  00 2c 0a 20 40 00 40 11  3f 4c c0 a8 38 03 c0 a8   ·,· @·@· ?L··8···
0020  38 01 ca 21 13 6a 00 18  88 3a 00 01 35 31 33 2e   8··!·j·· ·:··513.
0030  74 78 74 00 6f 63 74 65  74 00 00 00              txt·octe t···
```

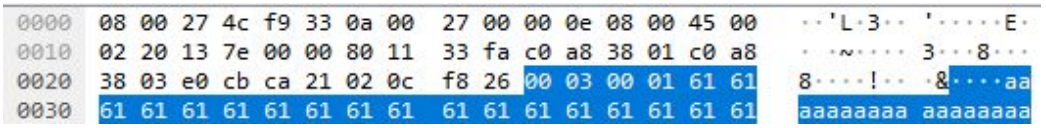
Looking at it closely, the bit marked in blue is the UDP packet (the rest is overhead from underlying protocols making the total length of the sent packet 60). Here we can see the first two bytes (op code) are 00 01 which indicates a read request. After that we have 7 bytes for the name of the requested file: *35 31 33 2e 74 78 74* which is easier to read in char format: *513.txt*. It's then separated by a 0-byte (00) and after that follows 5 bytes for mode which is *6f 63 74 65 74* or in char string: *octet*. The UDP data ends with another 0-byte.

2, 8, 10: Protocol: NetBios Name Service. The server is trying resolve the hostname of the client through NBNS - a protocol supporting legacy software.

3, 9, 11: Protocol: Internet Control Message Protocol is a message protocol that in this case is used to send an error message saying the destination port is unreachable (I don't know why it was unreachable).

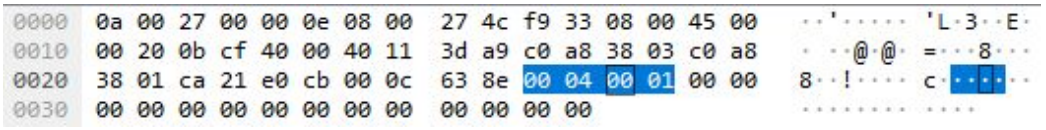
4-7: Protocol: Link Local Multicast Name Resolution. The server is sending queries for reverse dns lookups to a network multicast address, using both IPv4 and IPv6. It's mainly used by windows to resolve local network hostnames. I believe this is caused by the use of getHostName() from the InetSocketAddress class.

12: The server sends the first data packet from port 57547 to client port 51745. It is a full sized (516 B) packet indicating it's not the final packet. Since the barebone UDP protocol doesn't include any op code or block number headers, wireshark will not show it, unlike for TCP. One have to look manually in the packet to see that.



When we do that we see that op code is 3 (*00 03*) and block is 1 (*00 01*). After that follows the payload (image is cut).

13: The client sends a 4-byte UDP packet.



It tells us the packet has op code 4, indicating this is an acknowledgement packet, and that the block number is 1. The client has acknowledged the first packet!

14: The server sends a packet with length 5, indicating this is the last packet it wants to send.

15: The client acknowledges the packet and the transfer is complete.

Differences between read and write requests in tftp

An RRQ has op code 1, a WRQ has op code 2. Read requests are used for client download (GET), and write requests are used for client upload (PUT). The main difference lies in which host sends data and which send acks.

RRQ: After the request, the server immediately sends a data packet to the client with block=1. When received, the client acknowledges the data packet by sending an ack packet with the same block number. Server increments the block number for each acknowledged data packet. Rinse and repeat until it's all sent or an error occurs.

WRQ: Following the request, the server sends an ack packet with block=0, telling the client it's ok to start transmitting. After that, the scenario is the same as in a read request, but with the roles reversed.

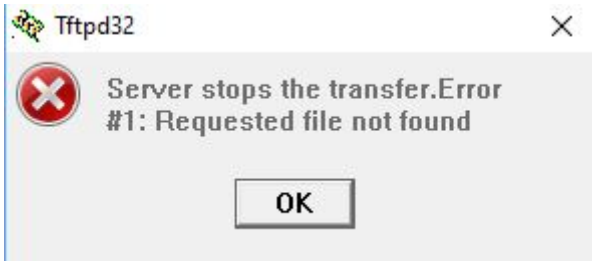
Problem 3

One thing I noticed: there seem to be a small bug in the suggested windows client (tftpd64). If the client tries to put a 0-byte file, e.g. a text file with nothing in it, the client doesn't seem to send anything more than the write request for the file, while any server would expect a data packet, even if empty, to indicate a transfer termination. Therefore the server will time out on receiving a packet. This does not happen with the ubuntu client.

Error code 0: Not defined, see error message (if any)

This error happens if the transfer times out on the server side (server failed to receive the correct data or ack packet a certain amount of times).

Error code 1: File not found
Requested file was not found on the server.



Note: In the case of a read access error, a FileNotFoundException can still be thrown on windows (by FileInputStream). This because windows seem to have a problem with File.canRead() (at least on some versions).
https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6203387&fbclid=IwAR3ap5O0D7PcDvjMoy_94Dm2nQ63_DrHCJMpz-tq0j0mPnBBkk9dvKjG1Rw.

Error code 2: Access violation
This is sent on a generic IOException as suggested in the assignment. It should also be sent on read access error, on non windows machines at least. I triggered it by closing a file input stream. In doing so I discovered another small bug in the client: if the error code is 2, the message is cut off. This should say "Access violation".

Error code 6: File already exists
This error is sent when client tries to PUT a file when a file of the same name already exists on the server (to avoid overwriting the old file).

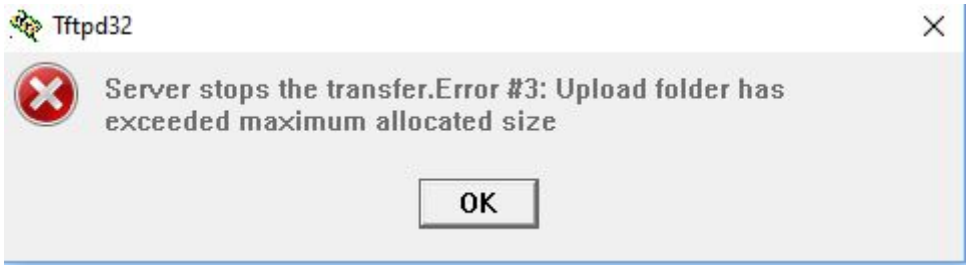
VG-task 2

The amount of info about TFTP error codes on the internet are almost non-existential, and sometimes the titles doesn't say it all. I interpreted the use of them the best I could.

Error code 3: Disk full or allocation exceeded

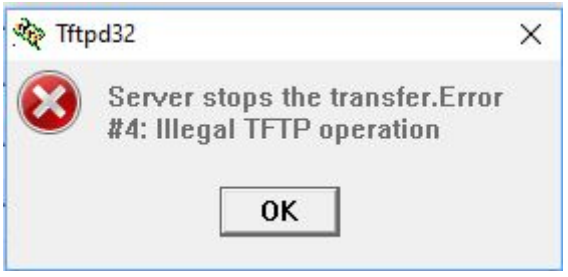
In the case of error 3, it would of course happen when the server disk is full, but since I couldn't find an easy way to test that, I instead decided to allocate a maximum size for the upload directory (specified in server settings).

I realized that re-calculating the size of the directory content for every packet would take up a lot of CPU time, so I instead used an interval setting (also specified in server settings) that makes sure the control runs every (by default) 1 uploaded megabyte. This means the actual directory allocation may exceed the maximum setting by the interval - 1 byte. However, I felt this method was more secure than e.g. calculating the directory content size at the start of the upload, add to a size variable and then check if the variable exceeded the maximum allocated space. If several users would start uploading at the same time, the actual space could potentially exceed the allocated space way more than what it can now. Another way to test it could have been to allow a maximum size for each user (keep a server log of how much data each user has uploaded), but the only real way to identify the user would be via IP, so this could easily be exploited by the user by changing IP.



Error code 4: Illegal TFTP operation

Since I can't specify the request myself with the client I'm using, to reproduce error 4: Illegal TFTP operation, I temporarily counted an RRQ as op code 0 on the server. That causes the server to send this error.



Error code 5: Unknown Transfer ID

I implemented this check as a part of the validatePacket method in TransferHandler. It simply checks that the destination port in the sent packet is equal to the source port in the received packet. I controlled the functionality of the implementation by manually modifying one of the ports in the method.


Tftpd32
Server stops the transfer.
Error #5: Unknown transfer ID
OK