



Linnéuniversitetet

Kalmar Växjö

Report

Assignment 3

IDV701



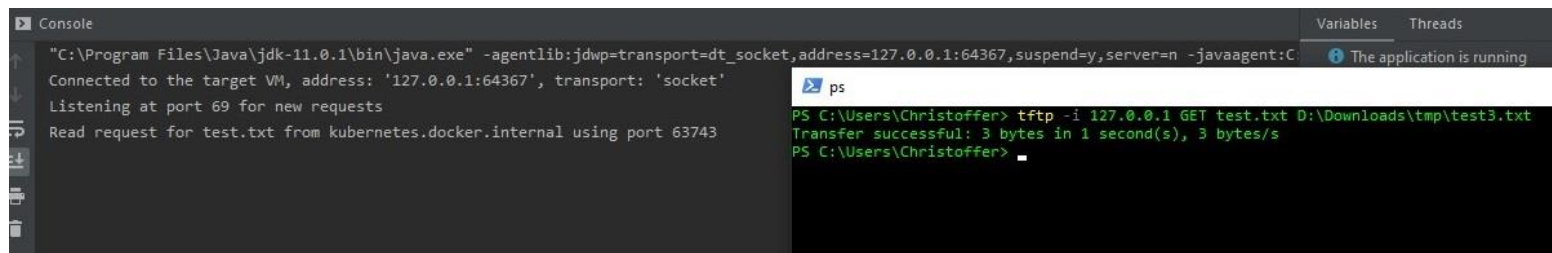
Author: Christoffer Lundström
Semester: Spring 2020
Email cl222ae@student.lnu.se

Contents

1 Problem 1	2
1.1 Discussion	2
2 Problem 2	3
2.1 Screenshots	3
2.2 Discussion	4
2.3 VG 1	5
2.3.1 Discussion	6
3 Problem 3	7
3.1 Screenshots	7
3.2 Discussion	9
3.3 VG 2	10
3.3.1 Discussion	11

1 Problem 1

4695	29.288033	127.0.0.1	127.0.0.1	TFTP	39 Data Packet, Block: 1 (last)
4696	29.288137	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 1



```
"C:\Program Files\Java\jdk-11.0.1\bin\java.exe" -agentlib:jdwp=transport=dt_socket,address=127.0.0.1:64367,suspend=y,server=n -javaagent:C:\Program Files\Java\jdk-11.0.1\lib\dt_socket.jar -jar C:\Users\Christoffer\Downloads\tftp.jar
Connected to the target VM, address: '127.0.0.1:64367', transport: 'socket'
Listening at port 69 for new requests
Read request for test.txt from kubernetes.docker.internal using port 63743

PS C:\Users\Christoffer> tftp -i 127.0.0.1 GET test.txt D:\Downloads\tmp\test3.txt
Transfer successful: 3 bytes in 1 second(s), 3 bytes/s
PS C:\Users\Christoffer>
```

1.1 Discussion

The first procedure of TFTP could be considered a handshake where the server receives a transfer request through port 69 which is standard procedure for the protocol.

From there on random ports in the standard (ephemeral) range are chosen for the actual data transfer. This is the reason why two sockets are used in the starter code; one for the initial request and one to initiate the data transfer.

2 Problem 2

2.1 Screenshots

Downloading 1794 bytes using GET.

```
PS C:\Users\Christoffer> tftp -i 127.0.0.1 PUT D:\Downloads\tmp\1700byte.txt
Transfer successful: 1794 bytes in 1 second(s), 1794 bytes/s
PS C:\Users\Christoffer> tftp -i 127.0.0.1 GET 1700byte.txt D:\Downloads\tmp\test22.txt
Transfer successful: 1794 bytes in 1 second(s), 1794 bytes/s
PS C:\Users\Christoffer> █
```

Uploading 1.9 GB Ubuntu image to server.

```
PS C:\Users\Christoffer> tftp -i 127.0.0.1 PUT D:\Hyper-V\ubuntu-18.04.2-desktop-amd64.iso
Transfer successful: 1996488704 bytes in 255 second(s), 7829367 bytes/s
PS C:\Users\Christoffer>
```

Downloading 1.9 GB Ubuntu image.

```
PS C:\Users\Christoffer> tftp -i 127.0.0.1 GET ubuntu-18.04.2-desktop-amd64.iso D:\Downloads\tmp\ubuntu.iso
Transfer successful: 1996488704 bytes in 264 second(s), 7562457 bytes/s
PS C:\Users\Christoffer>
```

Edge case:

A lot of blocks are sent. Last received is 0 bytes because $1,996,488,704 \text{ bytes} \bmod 512$ gives a remainder of 0 bytes (thus no end of transmission packet).

```
Receiving 512 bytes.
Sending ack for block 3899392
Receiving 0 bytes.
Sending ack for block 3899393
```

First ack is ignored and next request is delayed by 1000ms

```
Listening at port 69 for new requests
Read request for 1700byte.txt from kubernetes.docker.internal using port 58837
Sending 512 bytes. Block num 1
Awaiting ack for block 1
Request timeout..1
Sending 512 bytes. Block num 1
Awaiting ack for block 1
Ack received..
Sending 512 bytes. Block num 2
Awaiting ack for block 2
Ack received..
Sending 512 bytes. Block num 3
Awaiting ack for block 3
Ack received..
Sending 258 bytes. Block num 4
Awaiting ack for block 4
Request timeout..1
Sending 258 bytes. Block num 4
Awaiting ack for block 4
Ack received..
```

2.2 Discussion

In my examples I download and upload a textfile which is 1794 bytes long. This divides the file into 4 packets (512, 512, 512, 258). The last packet is less than 512 bytes and thus indicates end of transfer.

In the second example I upload and then download a huge file (1.99 GB) which is then divided into 3,899,393 packets.

Calculation

$1,996,488,704 \text{ bytes} / 512 = 3\,899\,392 \text{ packets} + 1 \text{ end of transmission.}$

A last packet is also sent because the recipient has not received an end of transmission packet because the prior packet matched the buffer size (512).

$1,996,488,704 \text{ bytes} \bmod 512 = 0$

Timeout and re-transmission tests

In my solution I tested retransmission by ignoring a specific block number and also setting a `Thread.sleep` after sending a block of data to make the transmission loop timeout.

The timeout is set to 1000 milliseconds so if the socket times out the function will return `Result.ERR` to the `await()` state handler which resends the packet and waits for another 1000ms before resending a maximum of three times. These values can of course be modified based on performance needs, but for my purposes I felt that three retries was sufficient.

2.3 VG 1

Uploading 1794 bytes over 4 blocks.

17	5.313151	127.0.0.1	127.0.0.1	TFTP	53 Read Request, File: 1700byte.txt, Transfer type: octet
154	5.326774	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 1
155	5.326866	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 1
294	5.343157	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 2
295	5.343219	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 2
296	5.343336	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 3
297	5.346601	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 3
298	5.346694	127.0.0.1	127.0.0.1	TFTP	294 Data Packet, Block: 4 (last)
299	5.346723	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 4

Image of last block in Ubuntu image download (1.9 GB).

7799...	261.421304	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 3899390
7799...	261.421330	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 3899390
7799...	261.421370	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 3899391
7799...	261.421395	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 3899391
7799...	261.421435	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 3899392
7799...	261.421460	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 3899392
7799...	261.421578	127.0.0.1	127.0.0.1	TFTP	36 Data Packet, Block: 3899393 (last)
7799...	261.421603	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 3899393

A typical data packet (opcode 3, block number, 0-512 byte data).

▼ Trivial File Transfer Protocol
Opcode: Data Packet (3)
[Source File: ubuntu-18.04.2-desktop-amd64.iso]
Block: 32768
[Full Block Number: 3899392]
▼ Data (512 bytes)
Data: 4546492050415254000001005c0000002e91706000000000...
> Text: EFI PART
[Length: 512]

Last packet contains no data in this case because data length mod 512 is zero.

▼ Trivial File Transfer Protocol
Opcode: Data Packet (3)
[Source File: ubuntu-18.04.2-desktop-amd64.iso]
Block: 32769
[Full Block Number: 3899393]

A typical ack contains 2 byte opcode and 2 byte block number

> [Timestamps]
▼ Trivial File Transfer Protocol
Opcode: Acknowledgement (4)
[Source File: ubuntu-18.04.2-desktop-amd64.iso]
Block: 32769
[Full Block Number: 3899393]

Write request for 1794 bytes.

7	2.583477	127.0.0.1	127.0.0.1	TFTP	53 Write Request, File: 1700byte.txt, Transfer type: octet
142	2.601214	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 0
165	2.702711	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 1
188	2.706535	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 1
189	2.706591	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 2
192	2.707378	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 2
193	2.707410	127.0.0.1	127.0.0.1	TFTP	548 Data Packet, Block: 3
194	2.707484	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 3
195	2.707511	127.0.0.1	127.0.0.1	TFTP	294 Data Packet, Block: 4 (last)
196	2.707607	127.0.0.1	127.0.0.1	TFTP	36 Acknowledgement, Block: 4

2.3.1 Discussion

Read and write requests are nearly identical actions. The only things that really differentiate the two is a difference in opcode and the direction of data. The UDP header that wraps the protocol will of course have src and destination fields as reflections of each other.

Read request

A read request is immediately followed up by a data packet of block number 1. After that an ack is sent to acknowledge that the packet was received.

Write request

The write request is different from the read request in that the first response to a write request is an Ack of block 0 which indicates to the client that the server is ready to receive data. After that the cycle of data packets and acks are identical.

Packets recorded

ACK: Contains a 2-byte opcode (4) and a 2-byte block number.

Data: Contains opcode 3, a block number and 0-512 bytes of data.

Block: 2 byte block number.

[Source file]: Metadata created by Wireshark, not sent with the actual packet.

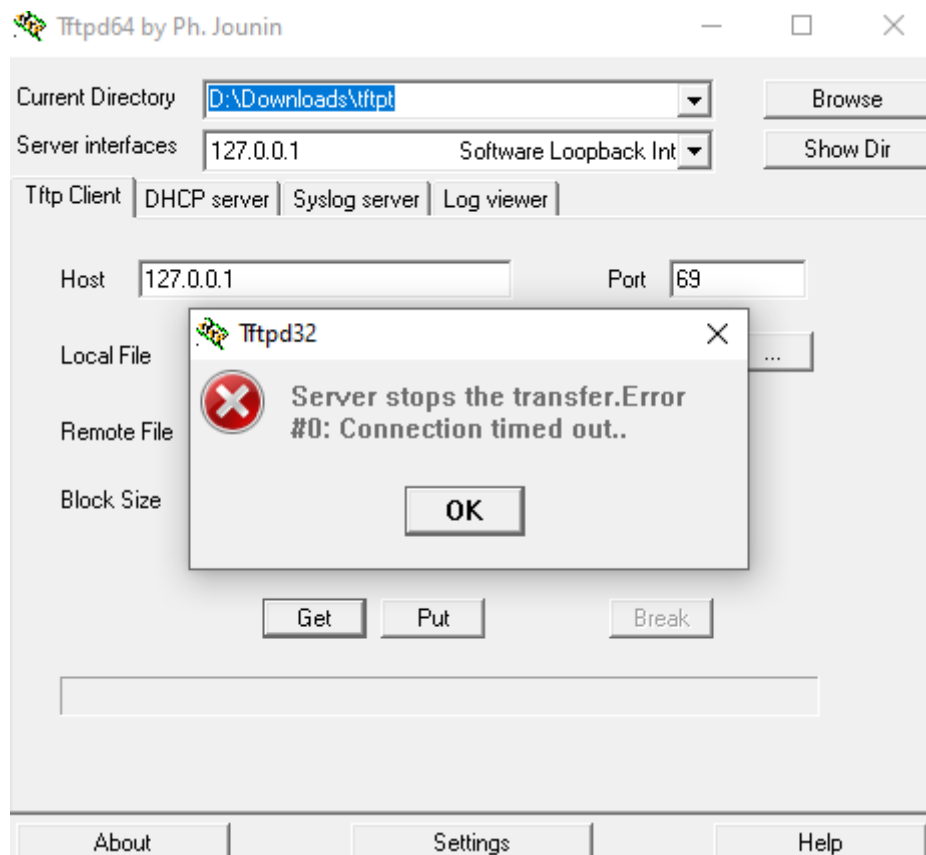
[Full Block number]: Metadata created by Wireshark, not sent with the actual packet, but can be used to identify blocks even if they have exceeded the maximum blocknumber (65535) and wrapped around starting from zero again as seen in the 1.9 GB download example.

3 Problem 3

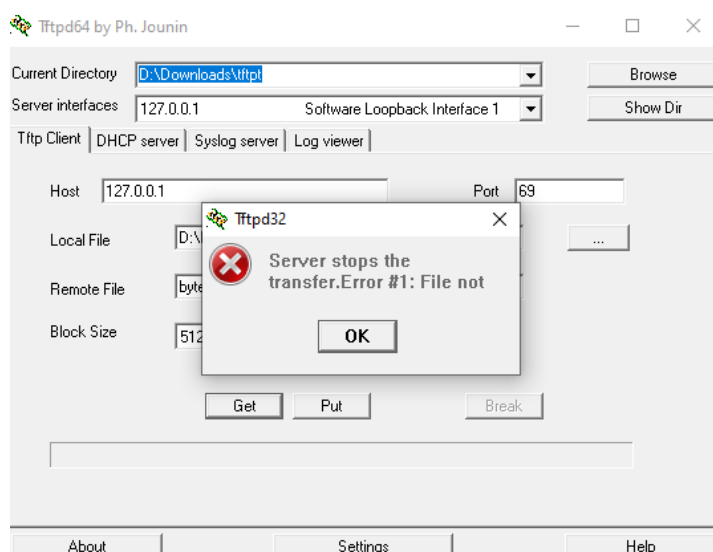
3.1 Screenshots

Error 0 – Connection timeout

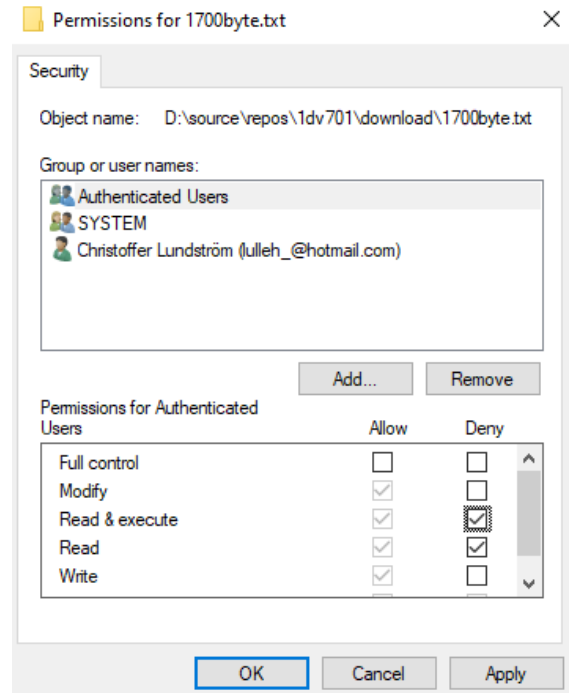
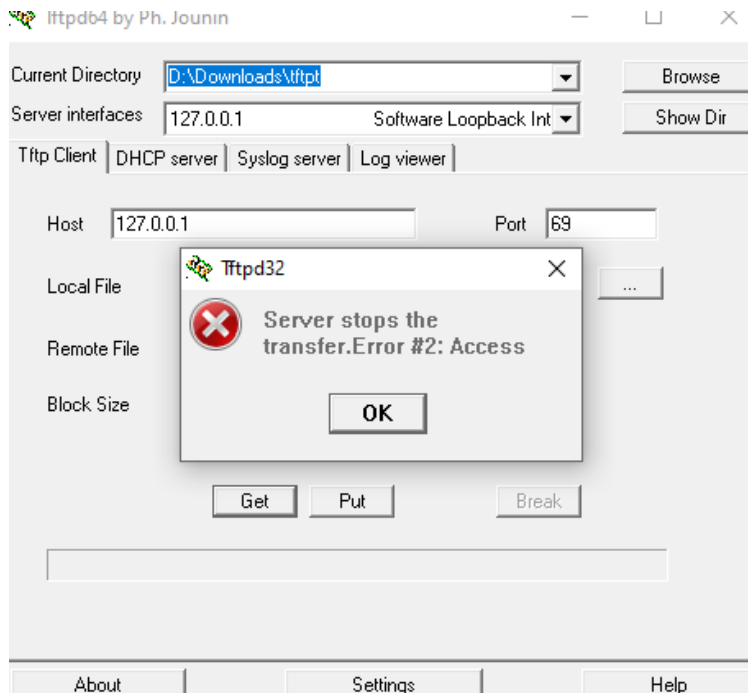
Is used as the general error 0. TimeoutException is for example thrown when a packet has been retransmitted three times with no response.



Error 1 File not found

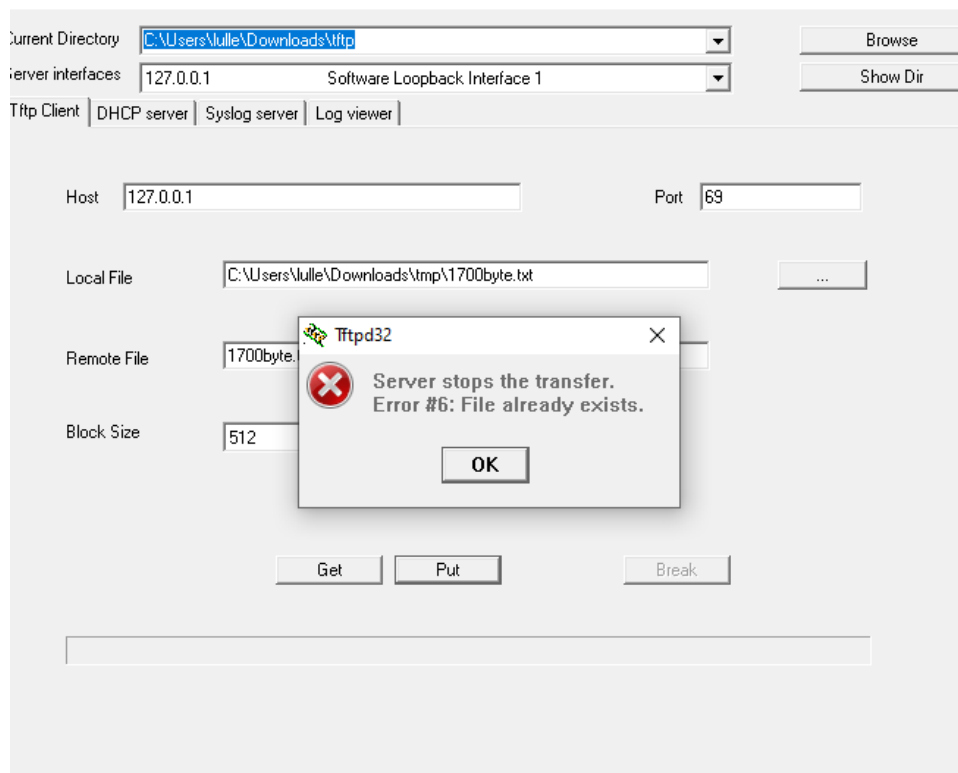


Error 2 Access violation



Error 6 File already exists

File 1700byte.txt already exists in download dir.



3.2 Discussion

Error 0 – Not defined

The first error is a general error I use a timeout exception. Should the client not send any acks or packets this will send an error and close the connection. The error can be considered a general catch all and can be used for multiple purposes.

Error 1 – File not found

A self explanatory error. When the client requests a file the server checks for its existence and return an error if it does not exist.

Error 2 – Access violation

This error can be interpreted differently depending on what OS the server is running on. In my case I used Windows 10 so by denying read access on a file I could try to download that file which would result in the server trying to read it locally and thus throwing an error and returning an access violation error.

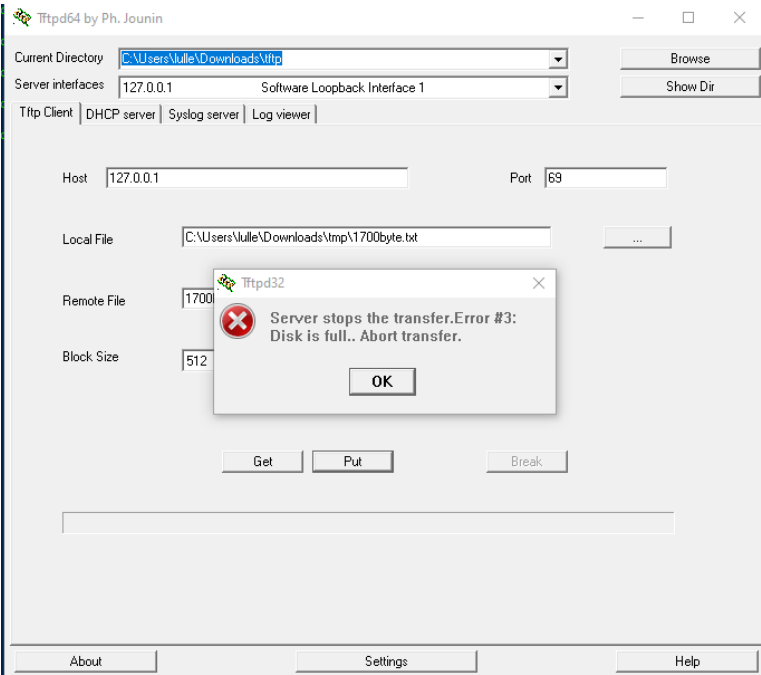
Error 6 – File already exists

A quite straightforward error. To avoid overwriting files on the server it returns an error if a file with an equal filename already exists in the upload directory.

3.3 VG 2

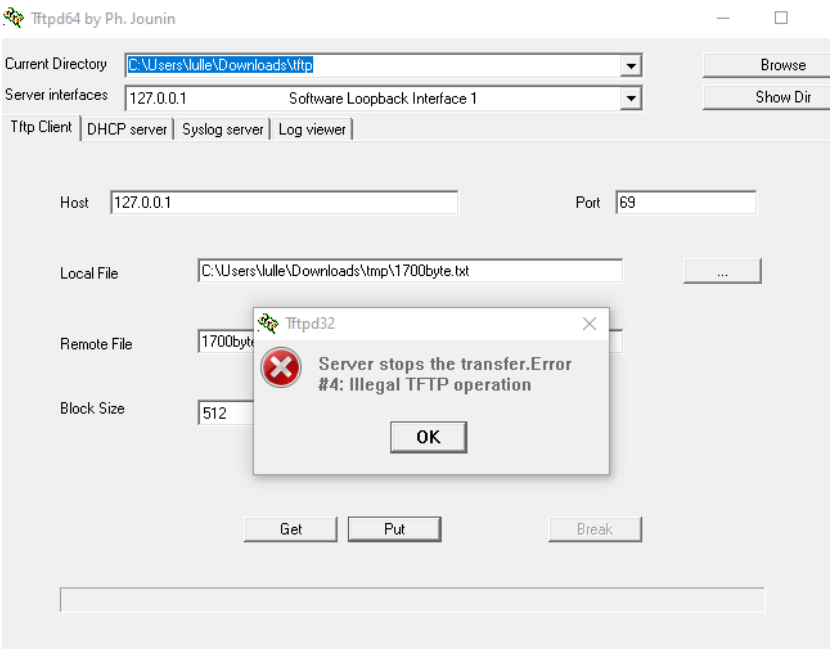
Error 3

Simulating a full disk.



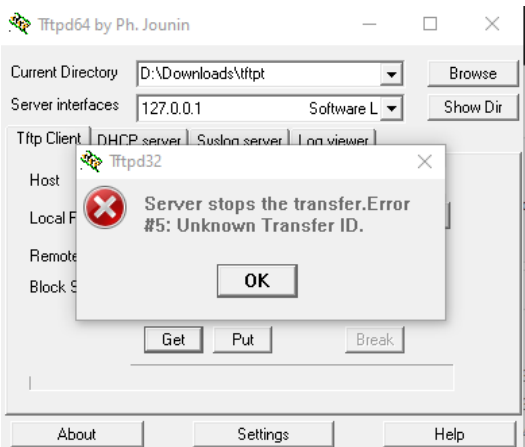
Error 4

This error is thrown when an illegal opcode is sent to the server.



Error 5

Unknown transfer ID caused by mismatching sender/receiving ports.



3.3.1 Discussion

Error 3

Because I did not want to fill my disk I simulated a full disk by changing the retrieved value from `getUsableSpace` at runtime in the bit of code that handles disk space validation. The error is more specific than returning a general error code 0 message.

Error 4

The TFTP illegal operation error is sent when an invalid packet is sent to the server. Either if its too large or if it contains a non existing opcode. When that is the case the server closes the connection.

Error 5

This error covers cases where a packet is sent to the server from an invalid source with a different port number. It returns an error to the sender stating that its id is unknown. The error is easily simulated/reproduced by changing the incoming packets portnumber at runtime by setting a breakpoint in the code and changing the value making the ports mismatch.

