|  |
| --- |
| Assignment3 Introduction to Network Administration |
| Jonathan Didde-Esteban Henry Pap17VT - 1DV701March 3, 2017 |

Group effort – Comment Section:

Jonathan M. Didde-Esteban **40%** Henry Pap **60%.**

# Comments Section:

**Jonathan Didde-Esteban -** While the project was much more extensive, the workload was still relatively balanced. Henry had the greater grasp of expertise and insight in setting some of the core elements of the program but much of the time was distributed in meet ups and discussions on the programs functionality and process.

**Henry Pap –** The project was weird at first glance, once you read the TFTP document “rfc1250” (slightly) it was quite easy. The hardest problem was parsing 2bytes to *Int* or *short*.

Explanations

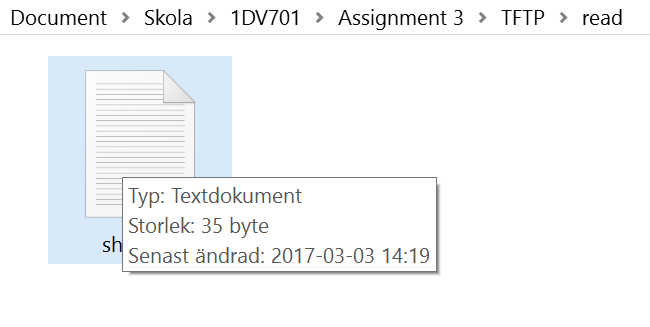
“*GetCanonicalPath ()*” this method [from the File class] will take a path e.g. “root/dir/../a.txt” and calculate the new path which is “root/a.txt”. As this can be a major access issue I use it to check if we are still in the root folder.

Getting bytes to int: You may have seen “((receive [2] << 8) | (receive [3] & 0x00ff))” or something similar. What this does is: getting the value from 2bytes, it starts with the highest byte and moves the bits 8 step to the left (left shift), the second part removes the higher bits. Why use it? Well to make it faster and honestly because it looks more clean then creating a ByteBuffer, allocating space and adding the bytes and convert it (it is enough that I do it when converting int to 2bytes).

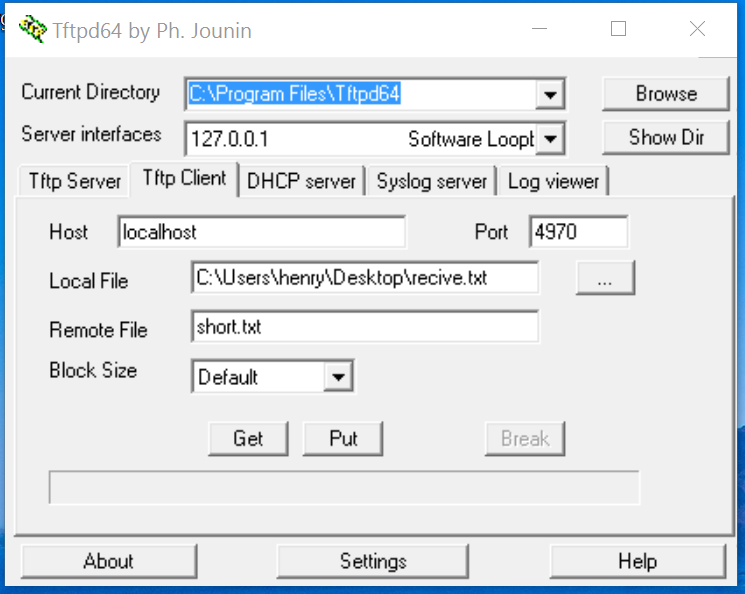
*Arrays.copyOfRange*: is used to create and Array with (value, start, end) parameters. Which in the case of getting the byte array from the datagram is the best way to do so. Since datagram actually is the size of 512 but only read for example 321 bytes, the method getLength of the datagram gives the 321 but the getData gives all (the useless 191 bytes too) so copyOfRange lets you specify the start and the end.

Problem 1

Receiving a file that is less than 512 bytes.

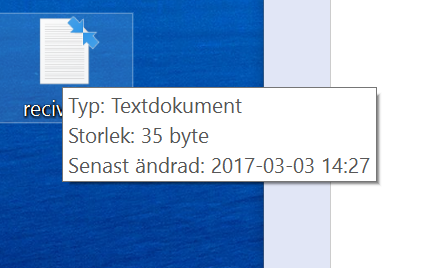


[A file that is 35bytes]



[The TFTP setup]

Now we do a get request and we will check if our empty recive.txt will be 35bytes



[The received file after Get request]

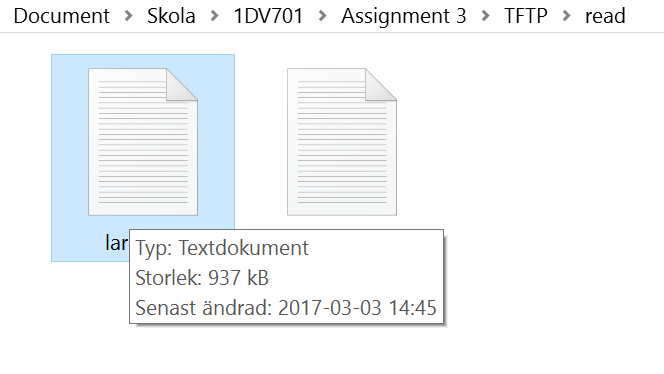
And as we can see, all the 35bytes which is less than 512 was sent.

So why are we using both socket and sendSocket?

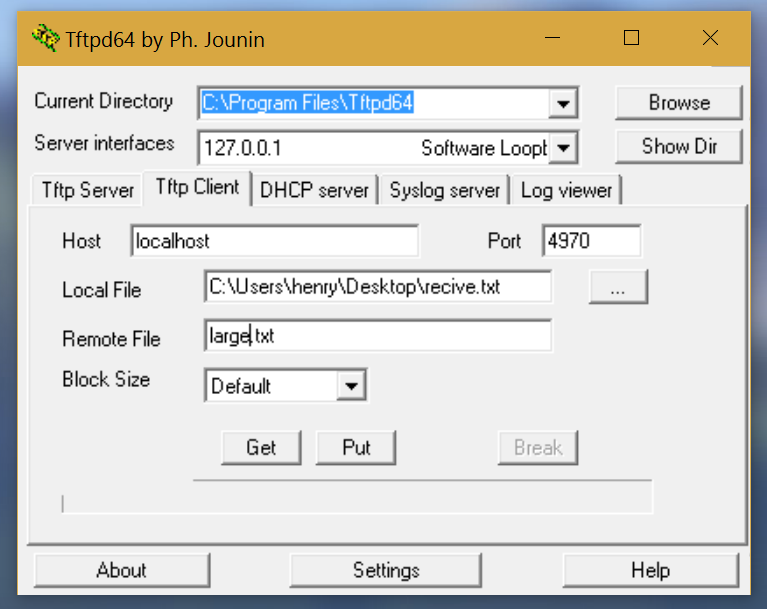
Because we use the socket for listening and when a new client connects we create a new connection for that specific client which we call sendSocket.

Problem 2

Receiving files that is larger than 512 bytes.

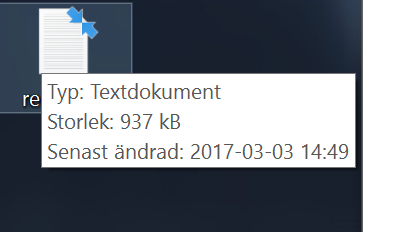


[A file that is larger than 512bytes]



[The TFTP setup]

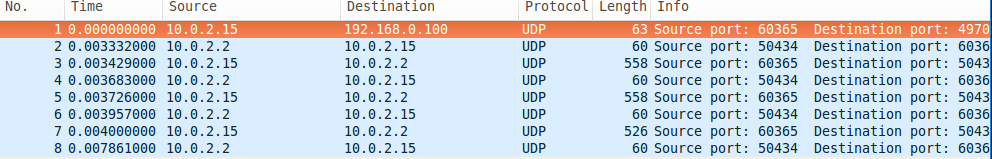
Now we do another get request and check if our recive.txt will be equivalent to our large.txt file on the remote side.

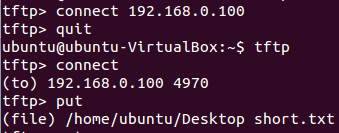


[TFTP Result]

Which it is, now we can send files that is larger, less or equal to 512 bytes.

# VG-task #1



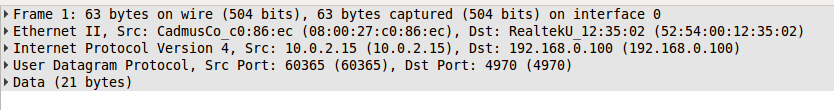


Notary: there are two important distinction to be made about traffic analysis from Wireshark.

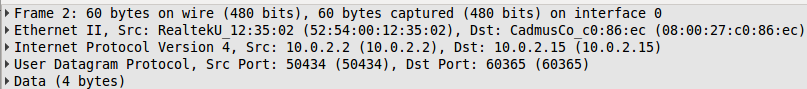
1. TFTP has its own distinct recognizable protocol listings within wire sharks frame – however due to the nature of our code and the scope of the assignment we have our TFTP layered over UDP as the foundation for the server’s coding. So, what we see from the protocol listing is the UDP foundation.
2. The testing between virtual box and our actual client/host computer creates the odd IP addresses 10.0.2.2, which is a special alias of the virtual machines IP for accessing the 127.0.0.1 feedback loop – likewise 10.0.2.15 is our actual host machine for the emulator (which functions as the server in this instance when using tftp client request from the Ubuntu linux machine).

Using a capture of the put/write to our server code I use to break down the details of the packets.

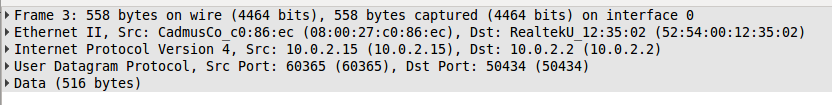
1. The first frame isn’t typical due to testing environment on a virtual machine – this packet basically rings out and helps to establishes the destinations for the connection between the virtual machine and our actual computer.



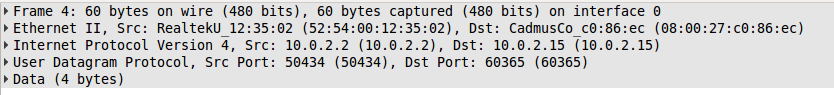
2. Write Request Packet; reaching our server with just 4bytes of data for the TFTP header.



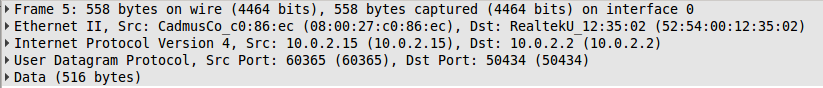
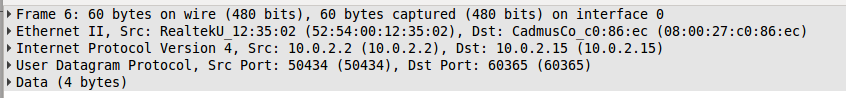
3. Server sending back the allowed file specifically as a data packet – 512bytes for data and 4bytes for the header TFTP standard.



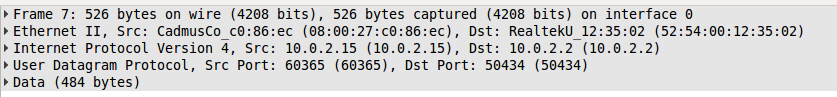
4. The client sends an acknowledgement packet confirming the data’s arrival – matching the first block and represent the first step of the proper data arriving – has no data just a TFTP header modelled for acks types, (such as the #block which the data packets have as well).



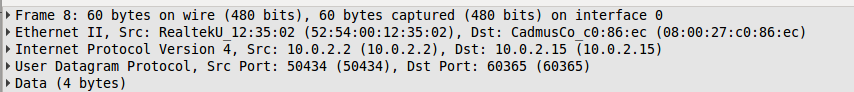
5 & 6 – Another cycle of a sent data packet and ack packet for confirmation between the server and the client…



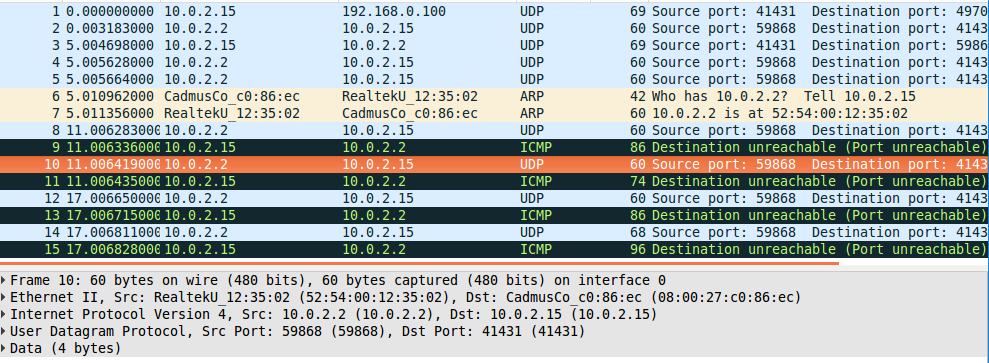
7 A data packet as well however because of its value at 484bytes not a full 512 packet we know or the server knows this marks the end of the data to be sent.



8 The server sends its final acknowledgement packet to the client knowing no more should be sent; typically, there is a delay between the two before closing connections as acknowledgments can be resent if not received this is especially important for the final closing packet and not just the sequences of data.



There are no true differences between a read and write method in TFTP other than the first set of bytes in a header and the sequence of steps for how data is exchanged between the server and client; the formats for data, acknowledgements, errors, size restrictions and so on use the same template, a major reason for TFTP having an appreciably simplistic approach. The notable differences between a read/get and write/put is in the sequence – with a read request having the server send return packets of data to the client and a write request having the client packets of data to the server. Methods for checking the data and handling errors between a client and server can be different but the underlying structure is the same and not handled separately or uniquely different from each other.

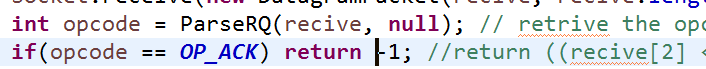


Here is a get method that fails halfway through – Frame 10 which is focused upon, is an error packet sent out to the server – however the server can’t reach the client, (specifically the port due to it being unrecognizable in a virtual machine).

Problem 3

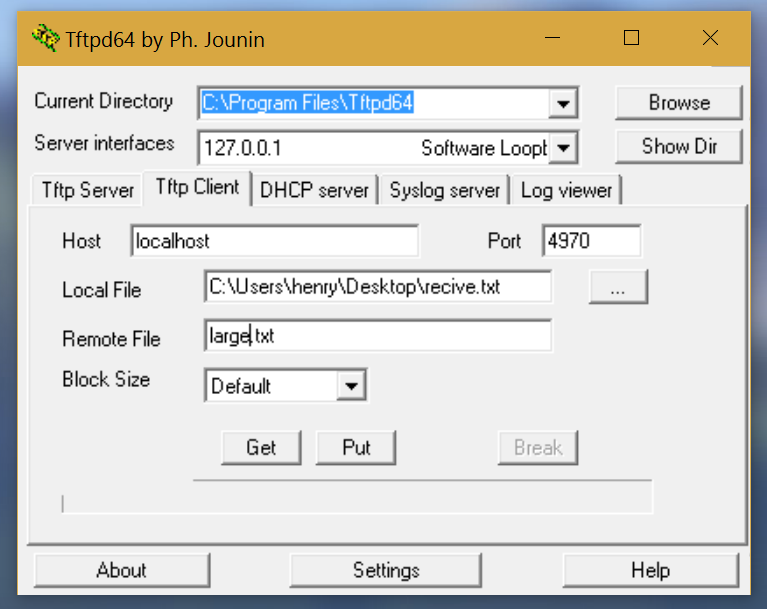
Showing the different errors (0, 1, 2, 4, 5, 6, and 7). VG-task #2 completed

**Error 0:** Not defined. This happens when retransmissions are too many, which we will fake with a timeout of 0ms.

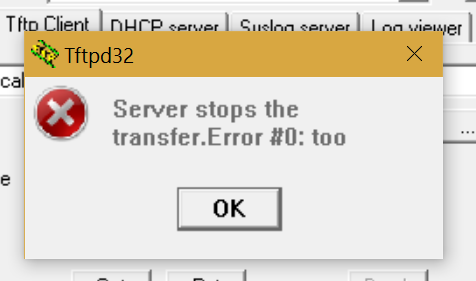


[Faking image]

Requesting the same large.txt file again

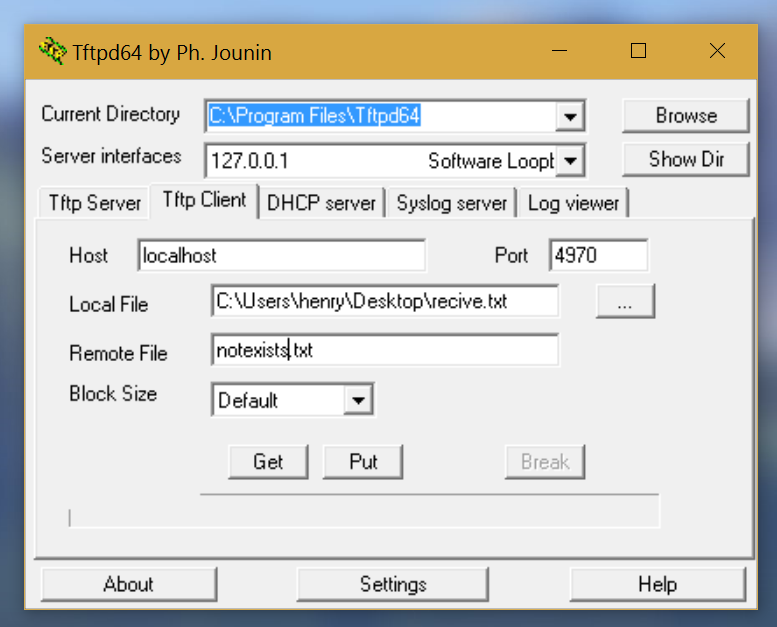


[TFTP setup]

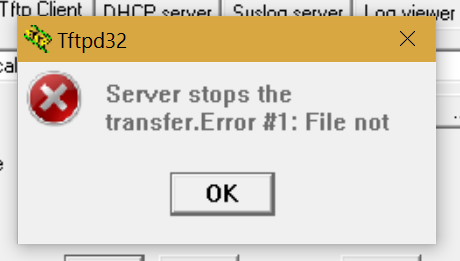


[Result image for error 0]

**Error code 1:** File not found

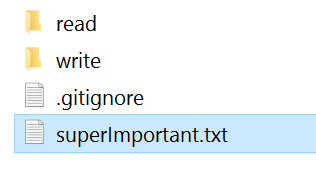


[TFTP setup]

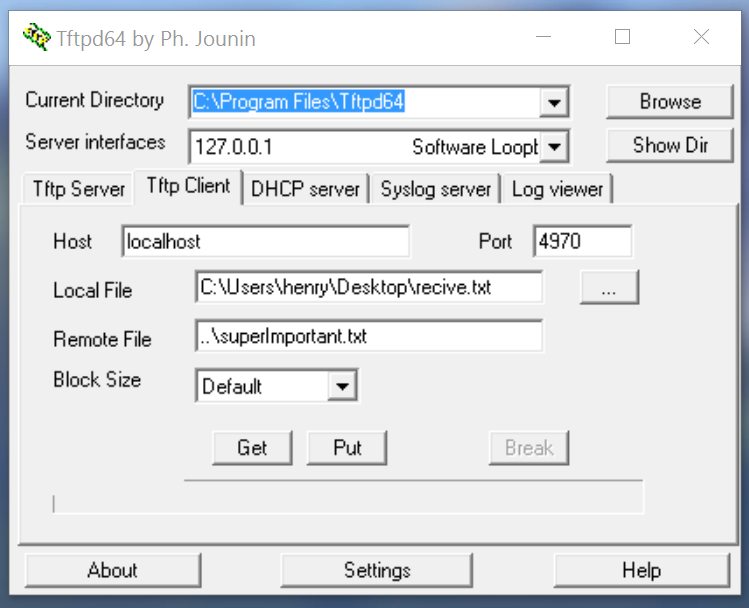


[The result]

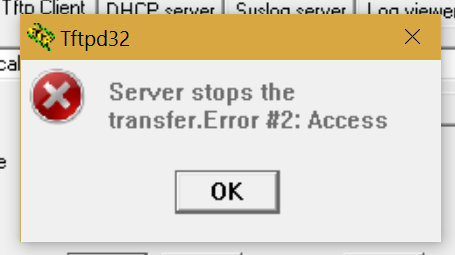
**Error code 2:** Access violation



[Important file created outside]



[TFTP setup]

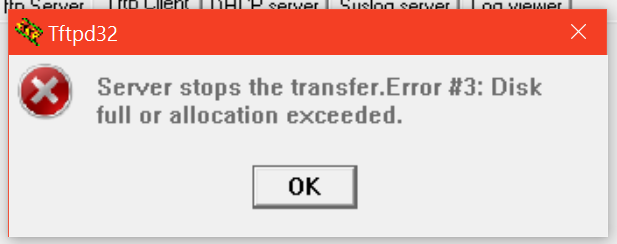


[Result image]

**Error code 3:** Disk full or allocation exceeded

The file size limit for this server is now set to 800kb

So by sending a file larger then that gives the error 3.

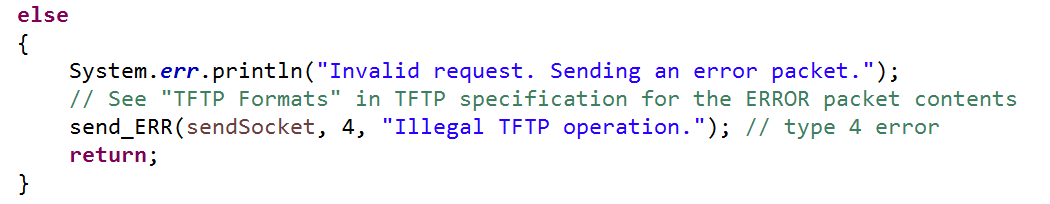


[Result from putting a larger file than 800kb]

**Error code 4:** Illegal TFTP operation

This error is caused if you make another operation than Write or Read.

If you are using Linux then maybe you can try this, but since we only have windows we can’t.



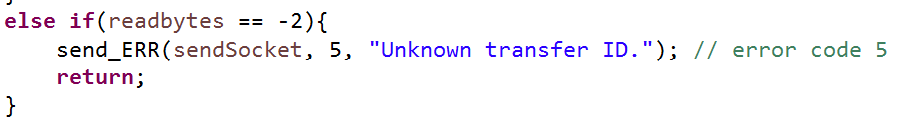
[Code that shows error code 4]

Error code 5: Unknown transfer ID

Caused if the port are not the same when receiving the ACK

3

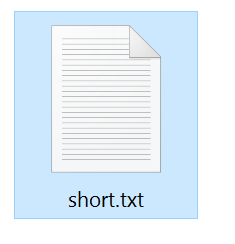
[The port checking]



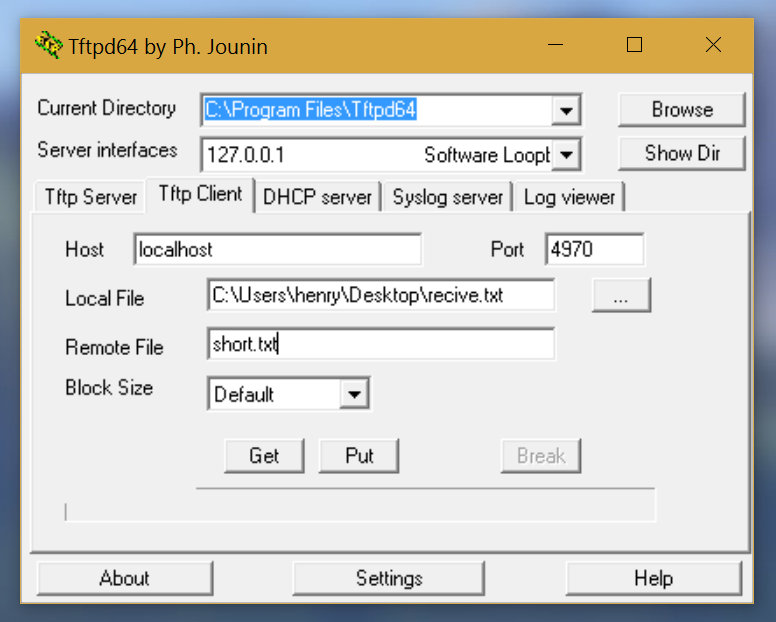
[And the result if this happens]

**Error code 6:** File already exists

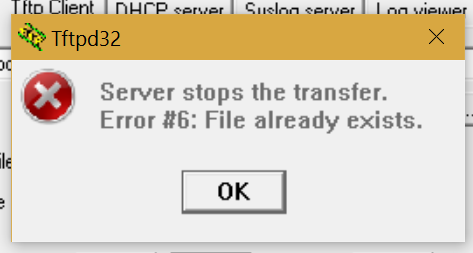
This will be achieved with a PUT request.



[File created in write folder]



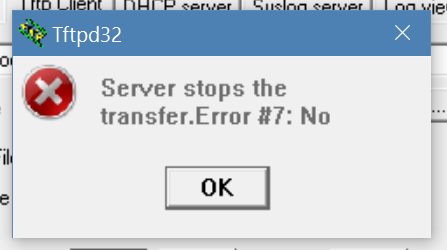
[TFTP setup]



[Result image]

**Error code 7:** No such user

Will result if you want to access /admin folder which only IP 1.3.3.7 can



[Error 7 result image]