LINNAEUS UNIVERSITY

Laboratory #3

TFTP Server

Phillip Lunyov & Rasmus Gazelius Skedinger

4/17/2017

Contents

Summary

Phillip's summary

After examining the starter's code, we divided our responsobilities to write at least one function with comments. I wrote and tested *receiveFrom* function and *send_DATA_receive_ACK* from scratch, adding some comments to make it much easier to understand (in the code, you will able to see a lot of comments that are hidding some output that is used to see the information that is used for functions). The other functions that are used in code which are from starter's code origins, are written by both of us (*HandleRQ* function and defining input-parameters). On the other hand, Rasmus took responsobility of writing *ParseRQ*, *receive_DATA_send_ACK* and *send_ERR* functions, as well he added some supportive functions to ease the calculation processes (*toBytes* and *fromBytes* functions).

Rasmus's summary

The workload in this assignment is divided into 50/50. The code I wrote: *Parse_RQ* and *receive_data_and_send_err*.

In this assignment, we wrote a lot of code, divided code into sections and put code's parts together into final product. It took me one day to walk through the code and fix a few logical and syntax errors to be able to successfully solve the third assignment. The major problem in the code's implementation was a test for returning 1 in ParseRQ function. It took me a moment to realise the issues which correct return-types.

Problem #1

Your task in this assignment is to implement a TFTP server functionality according to RFC1350, the only relaxation being that your implementation is allowed to handle only one transfer mode (octet). A program like this should be developed step by step. In what follows we give a suggested working plan to help you complete the assignment. Try not to start solving the next problem before you have thoroughly tested the previous one.

Download the TFTPServer starter code, open the full TFTP specification and read both of the files. Try to get an overall picture of the work to come.

The main objective from the beginning is to get a program that runs (unlike in Assignment 1, the provided code needs modification to run correctly). A suitable starting point is to handle a single read request. This involves the following steps:

- Get listening on the predefined port by implementing a receiveFrom() method.
- Parse a read request by implementing a ParseRQ() method. Once receiveFrom() has received a message, we must parse it in order to get the information (type of request, requested file, transfer mode). The first 2 bytes of the message contains the opcode indicating type of request. The following approach reads two bytes at a given address and converts it to an unsigned short:

```
import java.nio.ByteBuffer;

byte[] buf;

ByteBuffer wrap= ByteBuffer.wrap(buf);

short opcode = wrap.getShort();

// We can now parse the request message for opcode and requested file as:

fileName = new String(buf, 2, readBytes-2) // where readBytes is the number of bytes read into the byte a
rray buf.
```

Note: the problem of parsing the part of request containing the transfer mode should be solved by yourself.

- Once the parsing is done, we can test our program by sending a read request from the client and printing out the opcode (should be 1), requested file and the transfer mode (should be octet).
- Implement code that opens the requested file. *Hint*: before you can open the file you must add the path (variable READDIR) to the received filename.
- Build a response: add opcode for data (OP_DATA) and block number (1), each an unsigned short of 2 bytes in a network byte order. We suggest a similar approach as for parsing the buffer:

```
byte[] buf;
short shortVal = OP_DATA;
ByteBuffer wrap = ByteBuffer.wrap(buf);
short opcode = wrap.putShort(shortVal);
```

- Read a maximum of 512 bytes from the open file, add it to the response buffer and send it to the client.

- If everything works, the client will respond with an acknowledgment (ACK) of your first package. Receive ACK and parse it.

It is now time for a crucial test: make a read request from the client (request to read a file that is shorter than 512 bytes) and check that everything works properly. Include resulting **screenshot** in your report.

After successfully reading the requests, examine the TFTPServer starter code once more and explain in your report why we used both *socket* and *sendSocket*.

Solution #1

It is now time for a crucial test: make a read request from the client (request to read a file that is shorter than 512 bytes) and check that everything works properly. Include resulting **screenshot** in your report.

On the following screenshot, we used TFTP64 client to connect to our Java TFTP server and read the file “hello” from local directory.

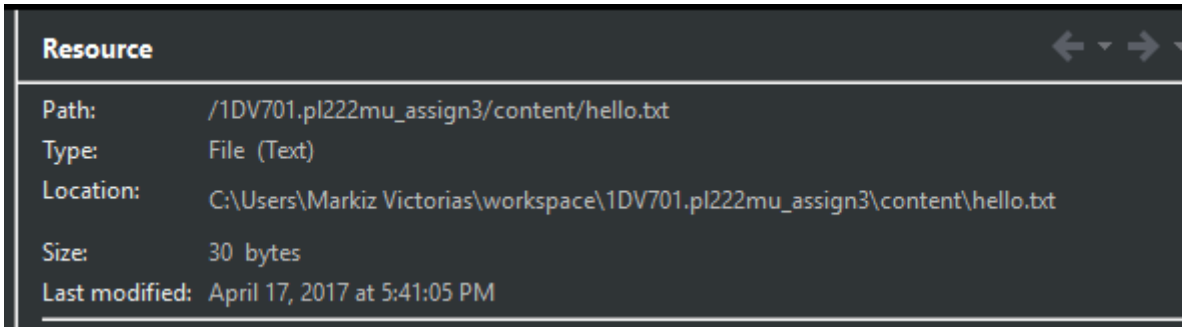

Figure 1.Requesting a file that is less than 512 bytes

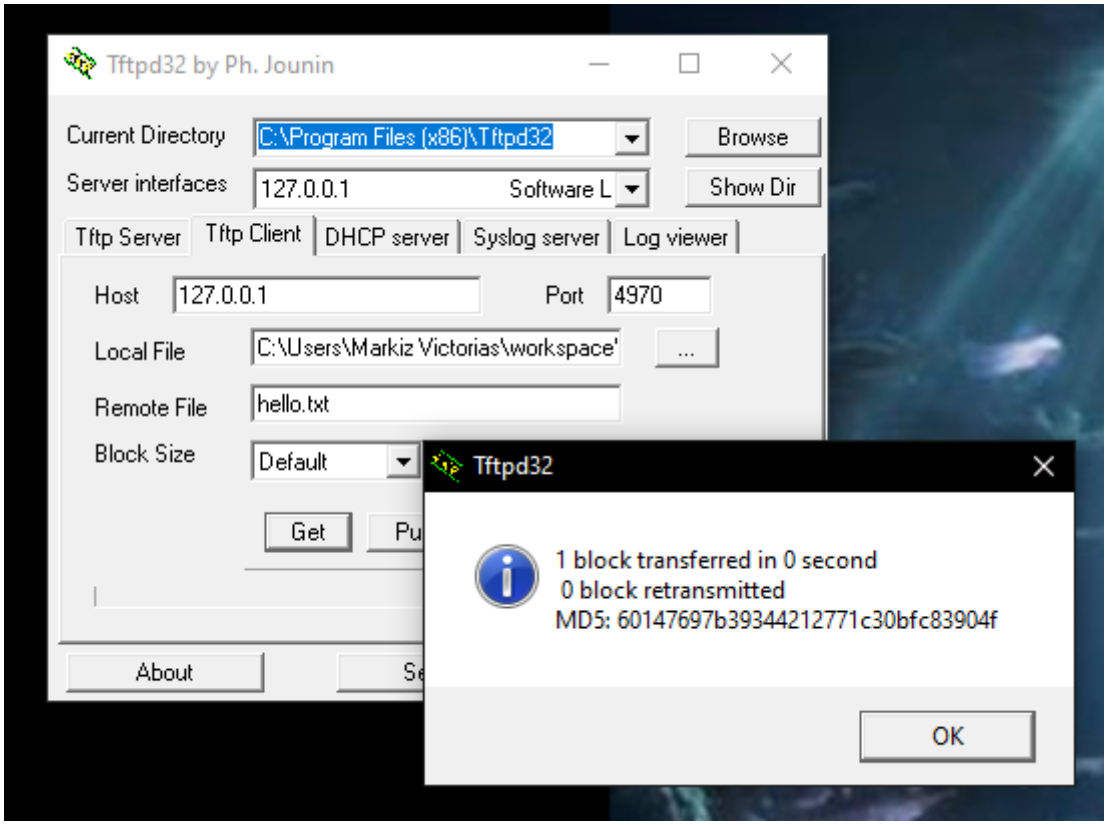

Figure 2. Client's program results

After successfully reading the requests, examine the TFTPServer starter code once more and explain in your report why we used both *socket* and *sendSocket*.

After examining the code, *socket* is used to establish the connection between client and server. As it parses the first datagram, extracts a socket address which client provided, and uses this socket address to determine data ports (read/write operations). On the other hand, *sendSocket* uses information from *socket* to send or receive actual information. At this point, we can say that *socket* is a control port and *sendSocket* is a data port.

Problem #2

Add a functionality that makes it possible to handle files larger than 512 bytes. Include resulting **screenshot** with sending multiple large file in your report.

Implement the timeout functionality. In case of a read request, this means that we should use a timer when sending a packet. If no acknowledgment has arrived before the time expires, we re-transmit the previous packet and start the timer once again. If an acknowledgment of the wrong packet arrives (check the block number), we also re-transmit. *Hint*: make sure that the program does not stuck in the endless re-transmissions.

Once read requests work properly, implement the part that handles write requests. Include resulting **screenshot** in your report.

Solution #2

To test the new requirements, we took a pdf-file (roughlt, 372 Kb) to transfer via TFTP.

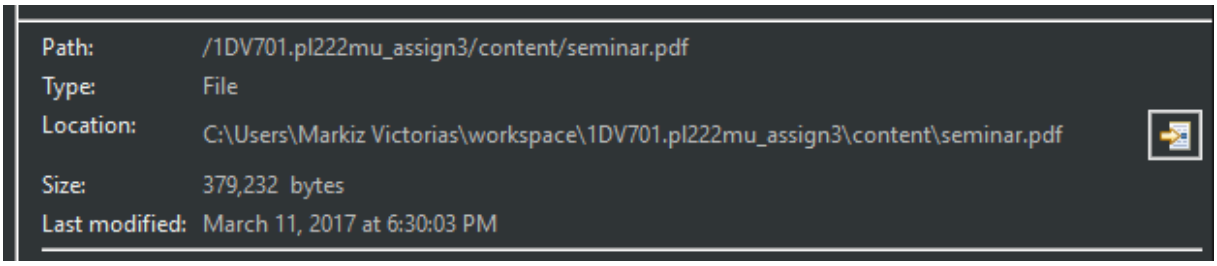

Figure 3. Properties of file

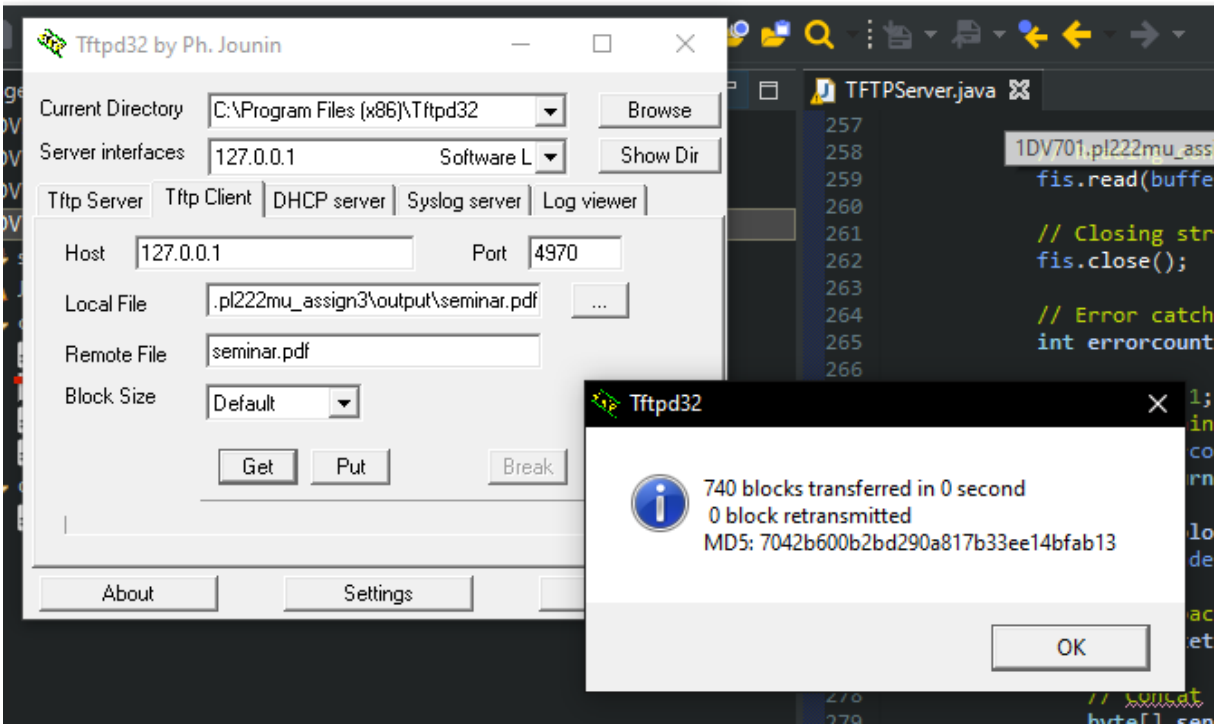

Figure 4. Results of transmission

It took 740 blocks to transfer the whole file to output-directory. Once we finished with reading operations, we implemented put-operation (write) in our code.

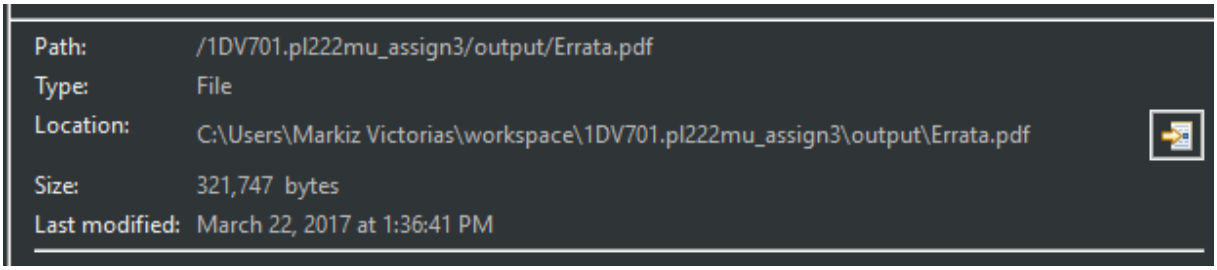

Figure 5. File's properties

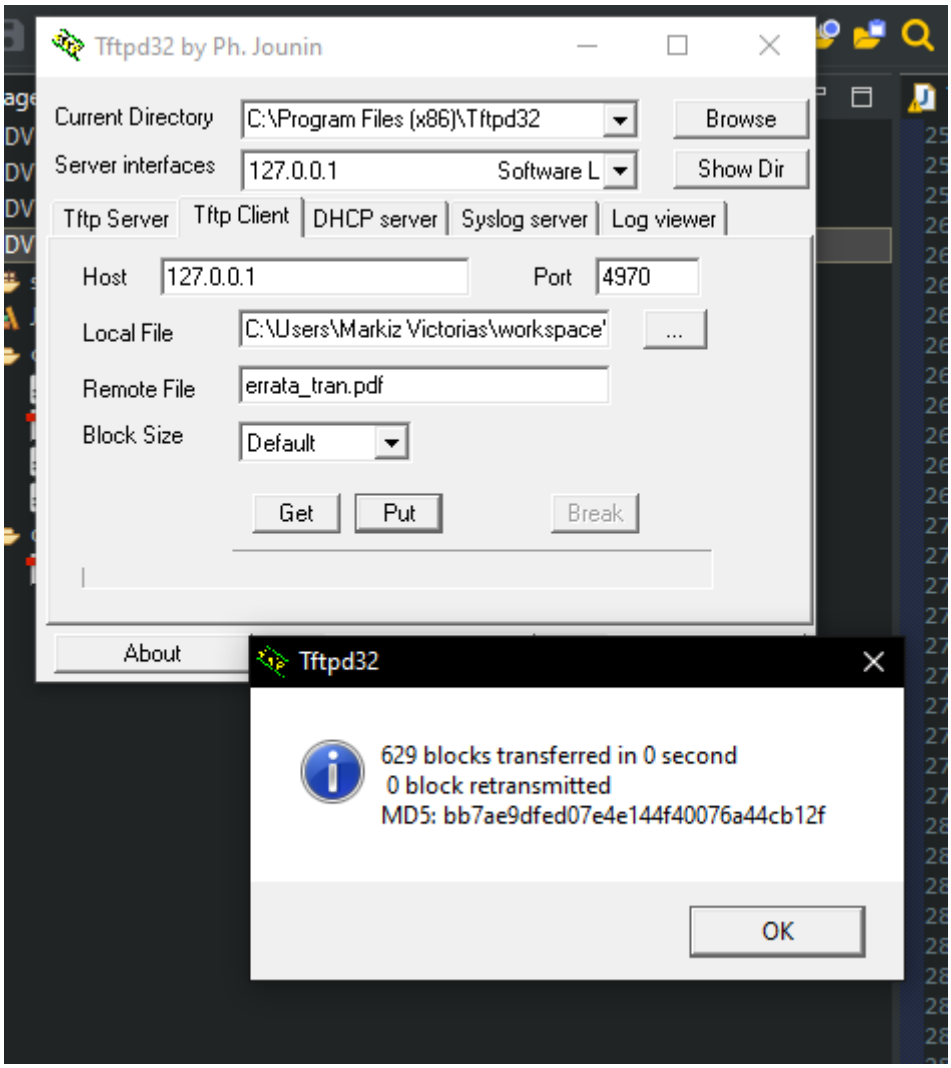

Figure 6. Results of writing operation

According to the Figure 6, it took 629 to write file from client to server.

Problem #3

Implement the TFTP error handling for error codes 0, 1, 2 and 6 (see RFC1350 specification). Include resulting **screenshots** with those exceptions in your report.

Hint: RFC1350 specifies a particular type of packets used to transport error messages as well as several error codes and err messages. For example, an error message should be sent if the client wants to read a file that doesn't exist (errcode = 1, errmsg = File not found), or if the client wants to write to a file that already exists (errcode = 6, errmsg = File already exist). More generally, an error message should be sent every time the server wants to exit a connection. Remember also to check all packets that arrive to see if it is an error message. If that is the case, the client is dead and the server should exit the connection.

Note: "Access violation" or "No such user" errors are related to UNIX file permission / ownership model. It is OK if your implementation returns one of these codes on generic IOException (after checking the cases for codes 1 and 6).

Solution #3

From RFC1350 specification:

```
Error Codes

   Value     Meaning

   0         Not defined, see error message (if any).
   1         File not found.
   2         Access violation.
   3         Disk full or allocation exceeded.
   4         Illegal TFTP operation.
   5         Unknown transfer ID.
   6         File already exists.
   7         No such user.
```

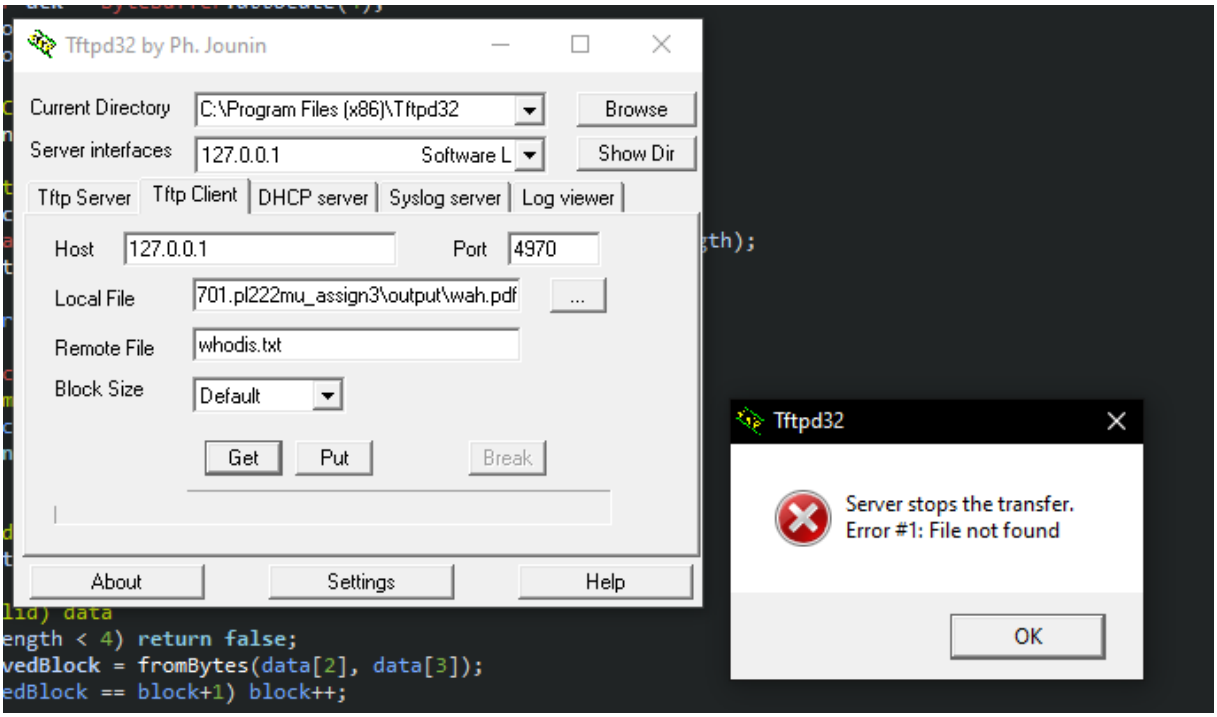

Figure 7. Error #1 - File not found

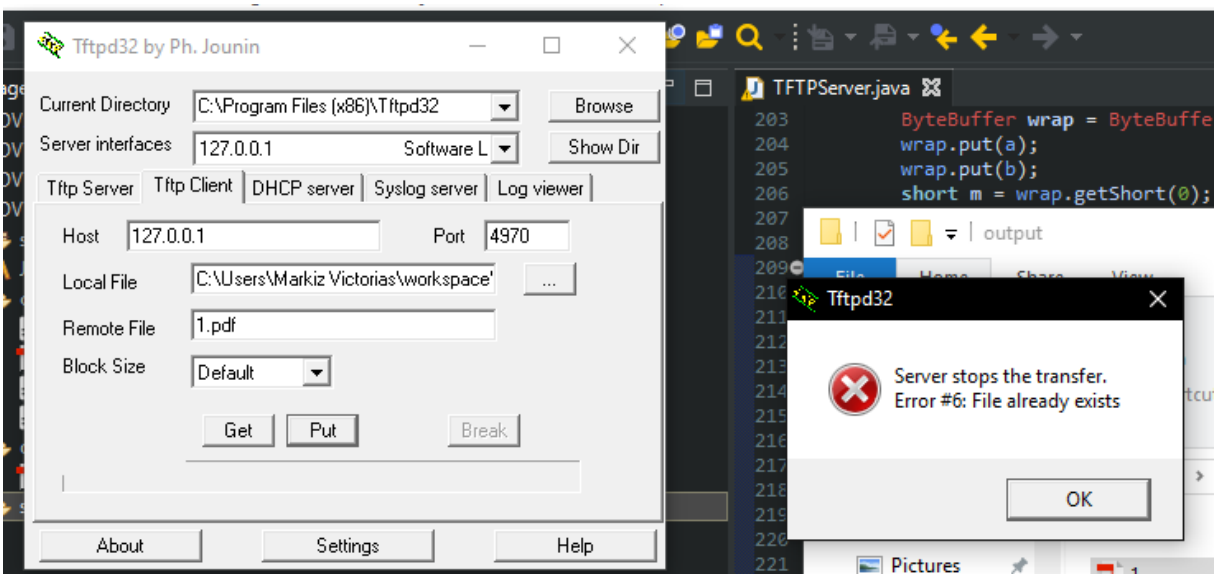

Figure 8. Error #6 - File already exists

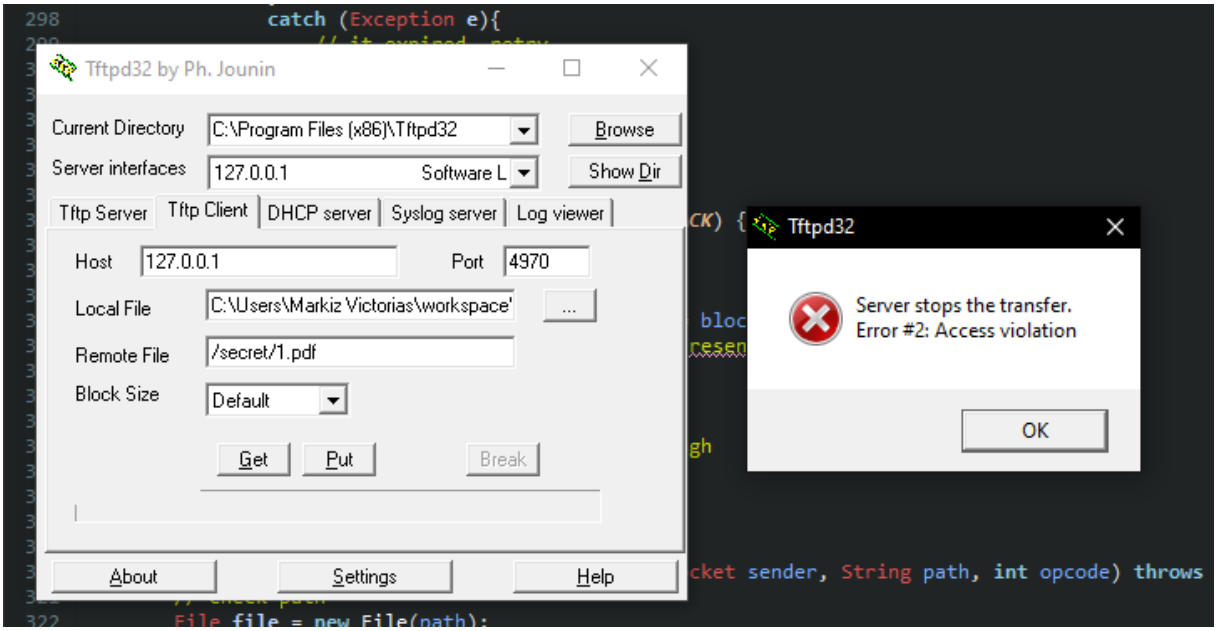

Figure 9. Error #2 - Access violation