1DV701 Assignment 3 – TFTP Server

The third assignment is designed to familiarise yourself with how networking services are documented and specified. Your task is to develop a TFTP server using Java that follows the specification. You should be able to access your server using a standard TFTP client. This assignment should improve your understanding of UDP, TCP, and how to read RFC documents.

Note: This assignment is performed in groups of 1-2 persons.

Reference material:

- TFTP introduction
- Full TFTP specification (RFC1350)

Useful downloads:

- Starter code for TFTP server
- Suggested TFTP client for Windows users

The TFTP Client and Server

TFTP, the Trivial File Transfer Protocol, is used to upload and download files. The following session shows how to connect to a server running on localhost port 69 and download the file RFC1350.txt. Note that the session uses localhost and the default port, 69; change it to the address and port your server is listening to when you test your server. The example shows how the tftp client works. It is generally pre- installed on Linux and macOS; you can download a GUI client for Windows from the link provided above.

```
# tftp
tftp> connect localhost 69
tftp> mode octet
tftp> get RFC1350.txt
Received 2360 bytes in 0.0 seconds
tftp> quit
```

You begin by launching the program (`tftp`). You then `connect` to a remote endpoint (`connect [hostname or ip] [port]`). Once a connection is established, you can upload or download files using the `get` and `put` commands (`get FILENAME`). You need to use `mode` to specify whether the file should be sent as `ascii` or `octet` (or `binary`). TFTP is a trivial file transfer service, so there is no way to change directories or list files, just upload and download.

Problem 1

Your task is to implement a TFTP server according to RFC1350. The server has only to support the `octet` mode. This is a complex task, so we suggest you break it down into the following steps:

1. Download the TFTPServer starter code, open the full TFTP specification, and read both. Try to get an overall picture of the work to come. Note that the testing tools can also be helpful to have a look at.
2. Get the provided code to handle a single read request. This involves the following steps:
    a. Listen on the predefined port by implementing a `receiveFrom()` method.
    b. Parse a read request by implementing a `ParseRQ()` method. The first 2 bytes of the message contain the opcode indicating the type of request.
    c. Open the requested file.
    d. Create the response packet. Add the opcode for data (`OP_DATA`) and a block number (1). These are unsigned shorts in network byte order.
    e. Read a maximum of 512 bytes from the file, add these to the packet and send it to the client.
    f. If everything works, the client will respond with an acknowledgement (`ACK`) of your first package.
3. Once you have successfully completed the steps, 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 a screenshot of this in your report.
4. Request to **write** a file that is shorter than 512 bytes and check that everything works properly. Include a screenshot of this in your report.
5. After successfully reading the requests, examine the TFTPServer starter code once more and explain in your report why it uses both `socket` and `sendSocket`.
6. Explain how you went about solving this problem in your report.

Hints

The following code sample shows how to read unsigned shorts from the received packet. Use `putShort` if you want to write an unsigned short.

```
import java.nio.ByteBuffer;byte[] buf;
 ByteBuffer wrap= ByteBuffer.wrap(buf); short opcode =
 wrap.getShort();
```

VG Problem 2

Extend the server so it can send and receive files larger than 512 bytes. Include screenshots of sending multiple large files in your report.

Implement the timeout functionality. In case of a read request, this means that you should use a timer when sending a packet. If no acknowledgment has arrived before the time expires, you retransmit the previous packet and restart the timer. If an acknowledgment of the wrong packet arrives (check the block number), you should also retransmit. Make sure that the program does not end up in an endless retransmission loop.

- Describe (in the report) how you tested timeouts and retransmissions.

VG-Problem 3

- Capture and analyse traffic between machines during a read request using Wireshark.
- Include screenshots and explanations of these in your report.
- The explanations should include a line-by-line analysis of what is displayed in the Wireshark screenshots, including the contents of each packet.
- Describe the difference between read- and write-requests.

VG-Problem 4

Implement TFTP error handling for error codes 0, 1, 2 and 6 (see the RFC1350 specification). Include screenshots of errors in your report.

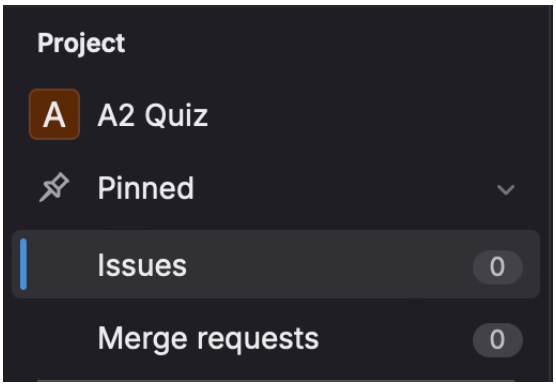
RFC1350 specifies a particular type of packets used to transport error messages, as well as several error codes and error messages. For example, an error message should be sent if the client wants to read a file that does not exist (`errcode = 1, errmsg = File not found`) or if the client wants to write to a file that already exists (`errcode = 6, errmsg = File already exists`). An error message should be sent every time the server wants to exit a connection.

Remember 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: The "`Access violation`" and "`No such user`" errors are related to the 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).

Submission

All submissions are done in GitLab. To submit, please create a new **issue** in your repository and select the TA for your group as an **Assignee** when prompted:



Your repository should contain the following:

- A **PDF** report containing a brief overview of each problem, along with the things specifically outlined to write about.
- All **.java** files that are required to run the program.
- A public folder containing resources.
- A **README.MD** file containing instructions on how to run the program.

Make sure to have a good Git **commit history** to demonstrate your collaboration. Make the commit names meaningful! Example guide:
https://www.freecodecamp.org/news/how-to-write-better-git-commit-messages/
Please agree on the final submission before you submit it.

In the **README**, include a summary of the instructions about how to run the program. In the **report** of what each of the participants of the group contributed and how you split the workload (in percent). For example, student_name_1: 45%, student_name_2: 55%. If these differ too much, the participants might receive different grades for the assignment. If one of the members in the group does not contribute at an acceptable level, please notify the teacher as soon as possible.