Assignment 3: TFTP Server

TFTP Server

The third assignment is designed to familiarize 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 people. Groups of 3 (or larger) are not allowed.

Reference material:

- TFTP introduction
- Full TFTP specification (RFC1350)

Useful downloads

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

The Standard TFTP Client

In this assignment you will use the TFTP client supported by the operating system to contact the server we are about to implement. In the following session show show to use the client to fetch a file (RFC1350.txt) from a TFTP server on a remote machine. The port we use here is the default TFTP port (69), and it should be changed to the actual port the server listens to. The client is pre-installed on Linux and macOS; you can download a windows client 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] [portnumber]`). 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 tranfer service, so there is no way to change directories or list files, just upload and download.

Your task in this assignment is to implement a TFTP server functionality according to RFC1350. 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 of the files. Try to get an overall picture of the work to come.
2. Get the provided code to handle a single read request. This involves the following steps:
    1. Listen on the predefined port by implementing a `receiveFrom()` method.
    2. Parse a read request by implementing a `ParseRQ()` method. The first 2 bytes of the message contains the opcode indicating the type of request.
    3. Open the requested file
    4. Create the response packet. Add the opcode for data (`OP_DATA`) and a block number (1). Each of these are unsigned short in network byte order.
    5. Read a maximum of 512 bytes from the file, add these to the packet and send it to the client.
    6. If everything works, the client will respond with an acknowledgment (`ACK`) of your first package.

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 the resulting screenshot in your report.

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

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();
```

Problem 2

Add a functionality that makes it possible to handle files larger than 512 bytes. Include resulting screenshots of sending multiple large file 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 re-transmit the previous packet and restart the. 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 re-transmissions.

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

VG-task 1

Capture and analyze traffic between machines during a read request using Wireshark. Include the resulting screenshots and explain the results in your report. The explanation should include a line-by-line analysis of what is displayed in the Wireshark screenshot, including the contents of each packet. Describe the difference between a read and a write request? Include a confirming Wireshark screenshot in your report.

Problem 3

Implement the 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 exist`). 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: 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).

VG-task 2:

Implement errors 3, 4, and 5. You should be able to demonstrate/reproduce the errors for code 3 and 4. For error code 5, an explanation of your implementation is sufficient, but you are welcome to try to reproduce the error. Include screenshots of the errors in your report.

Submission

If you work in a group of 2, make sure your submission is named `<username1>_<username2>_assign3.zip`, e.g., `aa222bb_cc333dd_assign3.zip`. Note that everyone in the group should submit a solution to Moodle (before the deadline) and that these solutions should be identical. If you submit solutions that differ, you will fail the

assignment and have to redo at a later date. So please agree on the final submission before you submit.

Include a summary 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.