Assignment 3: TFTP Server

TFTP Server

The third assignment is dedicated to development of TFTP server following the real-life specification. You will start with provided starter code, implement all of the functionality required by specification and test your server implementation using a standard client.

Reference material

- TFTP introduction
- Full TFTP specification

Required downloads

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

Note: This assignment is performed in groups of 2 people, see submission instructions at the bottom of this page for details. The groups can be found on a separate page.

The Standard TFTP Client

In this assignment we will use the TFTP client supported by the operating system to contact the server we are about to implement. In the following session we show how 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. Unix systems provide a client operating in interactive mode:

```
# tftp (Starts client program)
connect localhost 69 (Defines remote endpoint)
tftp> mode octet (Defines transport mode)
tftp> get RFC1350.txt (Read request for RFC1350.txt)
Received 2360 bytes in 0.0 seconds
tftp> quit (Terminate session)
```

It is as easy as it looks. After we have started the client program, we define the remote endpoint and transport mode to be used. Once this is done we can make one or more requests to the server. We use get filename to read a file from the server and put filename to write a file to the server. If we try to read files that doesn't exist, or write to files that already exist, an error message will be received and displayed. We can not change directory at the remote machine, nor can we list the available files. The only thing we can do is access files from two preassigned directories (read and write) at the machine hosting the server.

If you use Windows, the standard TFTP client lacks such an important option as server port selection. Therefore, we suggest using TFTPD32 application which provides TFTP client (as well as server) implementation with number of options.

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 i
nto the byte array 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*.

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.

VG-task 1: You must capture and analyze traffic between machines during a read request using Wireshark. Include resulting screenshots and explain the results in your report. The explanation should include a line-by-line analysis of what is displayed on the Wireshark screenshot including the contents of each packet. Finally, answer the following question: what is 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 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).

VG-task 2: Implement the remaining error codes. Include resulting **screenshots** with those exceptions in your report.

Note: "Illegal TFTP operation" error is related to client trying to send something other than valid RRQ/WRQ request code. As for the "Invalid Transfer ID", your implementation doesn't have to explicitly send that data structure, but you still have access to (remote port, local port) values for each packet. After connection initialization, server and client are communicating via arbitrary ports X and Y (in "ephemeral" port range). Therefore, you can imagine a situation when "ACK" message arrives from a client, but from a port Z that was not previously used in that particular communication session. In such a situation, server should send an error response with error code 5

Submission

The third assignment is done **in groups** of two people. The groups can be found on a separate page. Name your submission accordingly, e.g. aa222bb_cc333dd_assign3.zip.

In the beginning of your report, write a short summary of work performed by each group participant (one paragraph per person). Also evaluate your work compared to the work of your partner in percentage. For example: student_name_1: 45%, student_name_2: 55%. Mind that if these percentages differ by a big amount, the group members will receive different grades.

Note: if your group member is not able to provide any input for the assignment, you should notify the teacher in advance.

Both people from the group should make a Moodle submission and **it should be identical**! In other words, you need to reach a consensus in terms of your work evaluation. If two group members submit reports with two different solutions, their grade will be drastically decreased.

For other information read the general submission instructions.

[optional] If you like to give an opinion about an assignment (e.g., what could be improved), you can use an anonymous form.

Submission status

Submission status	No attempt
Grading status	Not graded

Due date	Sunday, 12 March 2017, 11:55 PM
Time remaining	16 days 10 hours
Last modified	-
Submission comments	Comments (0)

Add submission