Linnéuniversitetet
Kalmar Växjö

Report

Assignment 3

1DV701

Linnéuniversitetet
Kalmar Växjö

[redacted]

Contents

1 Problem 1 - Sending 512 bytes of data

Place your screenshots here

1.1 Introduction

The assignment consisted in creating a TFTP server. To do so, we based our research primarily on the document RFC 1350 [1] which gave the guidelines on how to implement the TFTP server. The documentation allowed us to understand how to organize pieces of data and most importantly, what codes, number of bytes and information we should have inserted.

1.2 Discussion of problem 1

This task consisted in sending and receiving a piece of data of 512 bytes or less, which, according to the RFC 1350, sending and receiving data implies, in the first step, to send or receive either an RRQ or a WRQ.
In the program, we have the capacity to differentiate between a RRQ or a WRQ in the method ParseRQ where we manage to fetch the OPcode and analyze it further.

Once analyzed, we send this opcode in a method called HandleRQ that will take the necessary action according to the opcode (RRQ or WRQ).

- RRQ: This action consists in sending a file. In this, we differentiate the fact that a datagram packet is less or equal to 512 bytes. If less than 512 bytes, this means that this is the last packet sent and that the connection can stop afterwards.
  When receiving an RRQ, the program first fetches the file, if the file is not existing, an error is sent. Once the file has been found, it is appended in a FileInputStream. This stream will determine the number of bytes the file has. Then, in the while loop, the input stream will read 512 bytes by 512 bytes up until the size of the file is less than 512 (in the case of less than 512 bytes, the loop runs only one time)[2]. The array of bytes is sent to the method send_DATA_receive_ACK. In this method, a single datagram packet is sent in a way that it enters a loop if, if the ack is not received under a precise amount of time, it is sent over and over (up to 5 times) before being discarded. Once the packet is sent, a ByteBuffer is instantiated in order to receive the ACK.

- WRQ: This action consists in writing a file. In our implementation, we first fetch the Opcode in the method ParseRQ. Then we read this opcode in HandleRQ where there is an if statement where we take actions if the opcode is equal to the WRQ.
  In this part of the code, we instantiate a FileOutputStream [3] that will locate the file to see if the file is already existing or create it. Then we will first create the ACK of 4 bytes with the associate BlockNumber by creating a datagram packet. This datagram packet will be sent in the method receive_DATA_send_ACK which will both send the ack, receive the packet and return the data to write in the file.

2 Problem 2 - File higher than 512 bytes

Place your screenshots here

2.1 Discussion

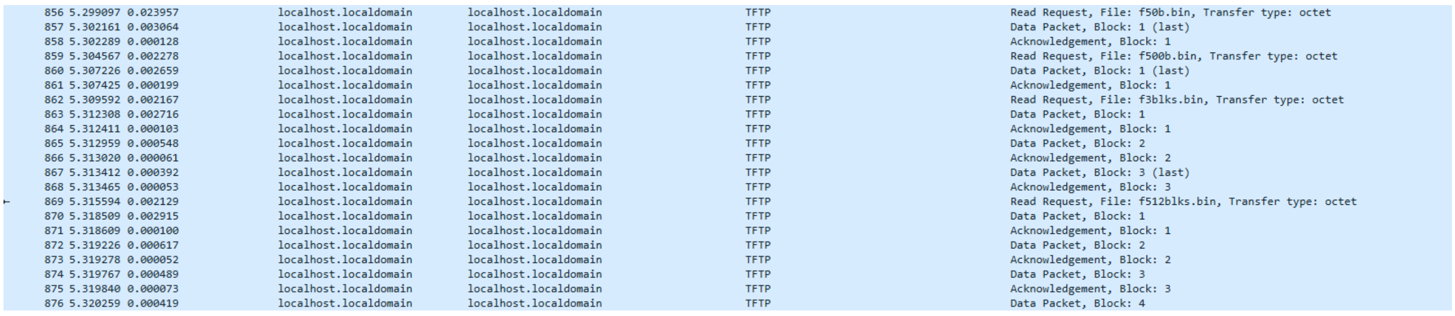
This problem consisted in sending and receiving files bigger than 512 bytes. To achieve this result, what we did was inserting a system of loops on the previous implementation that we made for problem 1.

- RRQ Loop : As explained previously, by using an InputStream, we will be able to read 512 bytes by 512 bytes up until the length of the buffer is less than 512 bytes. This will handle the sending of data almost automatically up until the bytes array is less than 512. Then the loop will stop sending data. This is consistent with the explanation in the RFC1350.

- WRQ Loop: Here, the loop will continue writing up until the size of the arriving buffer will be lower than 512 bytes in conformity with the RFC1350. All of that using the FileOutputStream.

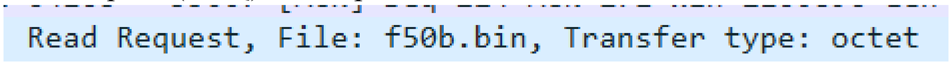
3 Problem 3 - Wireshark analysis

Capture of the information traffic during the execution of the tests:



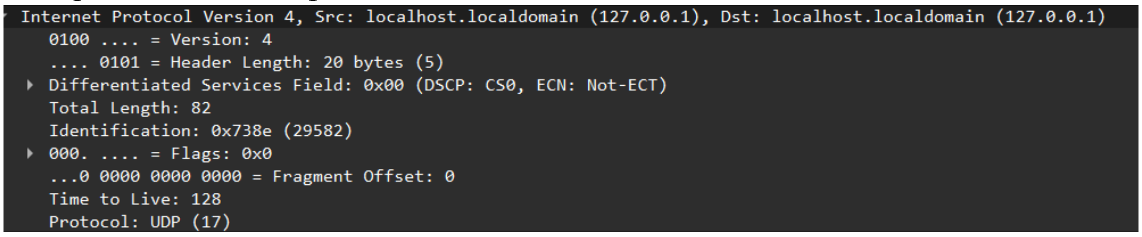
```
856 5.299097 0.023957    localhost.localdomain   localhost.localdomain   TFTP    Read Request, File: f50b.bin, Transfer type: octet
857 5.302161 0.003064    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 1 (last)
858 5.302289 0.000128    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 1
859 5.304567 0.002278    localhost.localdomain   localhost.localdomain   TFTP    Read Request, File: f500b.bin, Transfer type: octet
860 5.307226 0.002659    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 1 (last)
861 5.307425 0.000199    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 1
862 5.309592 0.002167    localhost.localdomain   localhost.localdomain   TFTP    Read Request, File: f3blks.bin, Transfer type: octet
863 5.312308 0.002716    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 1
864 5.312411 0.000103    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 1
865 5.312959 0.000548    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 2
866 5.313020 0.000061    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 2
867 5.313412 0.000392    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 3 (last)
868 5.313465 0.000053    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 3
869 5.315594 0.002129    localhost.localdomain   localhost.localdomain   TFTP    Read Request, File: f512blks.bin, Transfer type: octet
870 5.318509 0.002915    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 1
871 5.318609 0.000100    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 1
872 5.319226 0.000617    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 2
873 5.319278 0.000052    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 2
874 5.319767 0.000489    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 3
875 5.319840 0.000073    localhost.localdomain   localhost.localdomain   TFTP    Acknowledgement, Block: 3
876 5.320259 0.000419    localhost.localdomain   localhost.localdomain   TFTP    Data Packet, Block: 4
```

Read request (RRQ) - First picture



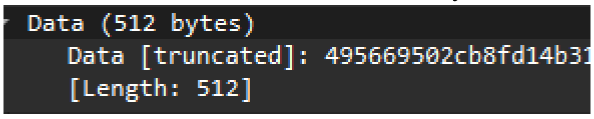
```
Read Request, File: f50b.bin, Transfer type: octet
```

Used protocol - Second picture



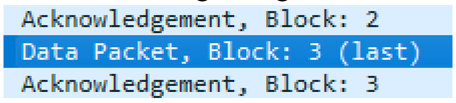
```
Internet Protocol Version 4, Src: localhost.localdomain (127.0.0.1), Dst: localhost.localdomain (127.0.0.1)
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 82
    Identification: 0x738e (29582)
  ▸ 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 128
    Protocol: UDP (17)
```

Shows the data of a chunk of 512 bytes - 3rd Picture



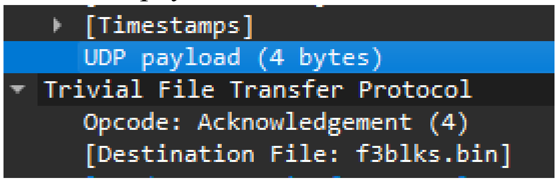
```
Data (512 bytes)
   Data [truncated]: 495669502cb8fd14b31
   [Length: 512]
```

Wireshark recognizing a chunk of bytes lower than 512 bytes - Fourth Picture



```
Acknowledgement, Block: 2
Data Packet, Block: 3 (last)
Acknowledgement, Block: 3
```

Shows the payload of an ACK - Fifth Picture



```
  ▸ [Timestamps]
    UDP payload (4 bytes)
▾ Trivial File Transfer Protocol
    Opcode: Acknowledgement (4)
    [Destination File: f3blks.bin]
```

3.1 Discussion

To run this capture, we set up the mode "Adapter for loopback traffic capture".
We ran the python tests that were passing in order to analyze the situation correctly.

What we can see here corresponds to the explanation in the RFC 1350.
In the first picture, we can see that there is a read request (corresponding to RRQ) to write a file with the encoding of "octet".
In the main capture, we can see that the protocol is "TFTP".
The packet sent, as shown in the second picture, uses UDP to send the data.
In the third picture, we can see that a "normal" chunk of data is 512 bytes. We are writing the term "normal" here because in the fourth picture, we can see that a block of data lower than 512 is considered as "last".
Then, we have the ACK, or acknowledgment block in the fourth and fifth picture which has its own labels and is 4 bytes of payload corresponding to the opcode and the sequence number.

By comparing both reading and writing operation, we can see that both data blocks are the same and contain the same type of information. Which means that the system does not make any difference between the sending or receiving data.

4 Problem 4 - Sending errors

The following errors have been implemented, sending, like defined in the FRC, the error codes with:

- Error 0 : "Error connection"
- Error 1: "File not found"
- Error 2: "Couldn't create file. (AV)"
- Error 4: "Illegale transfer ID"
- Error 6: "File already existing"

All those errors are implemented in the method errorPacket.

5 Testing

To test the code, we used two different methods.

The first method consisted in writing a simple implementation, outside of the template provided, to understand the overall concept of the tftp server. As this method was not considered a real testing file, it still gave us the capacity to test data transferring and byte manipulation using Java.

The second method consisted in using the provided python tests. As explained, we generated the different file executing the genfiles.sh file for testing purposes. Then we set up the test_tftp.py with the right path (hardcoded for easier access) and we used the command python -m pytest.

To test the code:
Please make sure that the file to produce (finishing in .ul) is not present when running the test. Otherwise, tests are failing.

6 Division of work

The major division of work we operated was that Benoit started by testing and tftp implementation outside of the template to test the first and André then adapted the implementation on the template, resolving the conflicts that the tests could show.
André finished the whole coding implementation while Benoit did the Wireshark analysis, all of that in a collaborative manner discussing concepts together.

References:

1 - RFC 1350 - The TFTP Protocol (Revision 2)
2 - Java InputStream
3 - FileOutputStream in Java - GeeksforGeeks