Candidate Number: 249763

Computer Networks Assignment: Implementation of Trivial File Transfer Protocol (TFTP) on top of User Datagram Protocol (UDP) and Transmission Control Protocol (TCP)

Introduction:

Trivial File Transfer Protocol (TFTP) is a protocol that allows files to be transferred from client to server and back. This report will explain how I attempted to implement a simpler version of TFTP on top of UDP and TCP protocols based on RFC 1350. A client and server will be able to communicate with each other through a UDP and TCP connection, and successfully read or write a file, which can be transferred via TFTP. The TFTP protocol will be supported by octet mode, and only handle the error in which a file cannot be found, thereby it will not be able to help with error handling on data duplications. The main port for the TFTP server is 69, but to avoid any errors due to admin rights, a port number of 1024 or above will be used for the server. I will use the five opcodes which TFTP packets support and use the specific format for each different opcode.

TFTP-UDP-Server:

Main Class: The server will be waiting and listening, in an infinite loop, for requests on port 69. If a packet was received, a new thread will be generated to handle the request.

TFTP_Packet class:
I wanted to make a class that handled the production of TFTP packets, for example, if a packet was received from a client, a method would check, which type of packet it is, (ACK, Error, WRQ, RRQ, or DATA), based on that a new TFTP packet would be produced so that the server could reply to the client, depending on the original message, For example, if a server received a data packet, a TFTP packet containing the block number and ACK number would be produced to send the reply. However, I was unable to correctly implement this.

OPCodes class:
This was a simple class that contained all OP codes from RFC 1350.

TFTPRequestHandler class:
I wanted to create this class, as a place where the packet which arrived could be analysed to work out what to do next. For example, if a read request arrived, the server would be able to figure out that it's a read request from its opcode and send data packets accordingly. However, I struggled to implement this in a way that it would compile.

-Write Request:

If the client has requested a write request, the server will have to send an acknowledgment packet to the client, as Data packets are received, the Request handler was supposed to contain a method that would send acknowledgments to the client that the data packets are being successfully transmitted. However, I was unable to successfully implement this idea.

-Read Request:

If the client has requested a read request, the server shall send data packets to the client, and receive acknowledgment packets or error messages from the client.

TFTP-UDP-Client

Main and Run class: The client will be asked to type “1” if they would like to request to write a file or “2” if they would like to request to read a file. Then the client will be asked to provide the server’s IP address to which they want to connect. Finally, they will be asked for their port number, any port number above 1024, and provide the file's name.

Opcode_RFC1350 class:
This was a simple class that contained all OP codes from RFC 1350.

TFTP_Packet class: I tried to implement this class differently from the TFTP_Packet class in the server, as I was unable to create a working class previously. This class contains a few methods which assist the production of the TFTP packets, for the read and, write requests. I tried to use a few helper methods, such as int-to-byte converters, for changing the number of the Opcode to a byte type, so I could concatenate the arrays and create one packet. There are still a few methods that I was unable to correctly code.

-Write Request:

The Write Request (WRQ) writes files to a Server. The method is first going to extract the file name out and it will be scanned to see if the file exists if not a “File not found” error will be thrown. I attempted to create a reader which read 200 bytes of data from the file’s content. This would be stored in “readData” so that it can be used to create a new packet for DATA, which includes, **readData** (Content), **numberOfRead** (number of characters in content), and the **blockNumber** (block number of Data packet). This Data packet is sent to the server using UDP. The protocol takes in the produced data packet, the server’s port number and the IP address of the server. After the data packet is sent, an acknowledgment packet is obtained to inform the client that the packet was successfully transferred. To ensure that the correct packet was transmitted, I carried out a check to see if the block numbers match the data packet and the acknowledgment packet.

Read Request

The Read Request (RRQ) provides clients with files to read from the Server. The method will also extract the file name and be stored in a read request packet alongside its Opcode from

RFC 1350. The method then uses a StringBuilder for the content of the request, as it receives data from the server, the method sends acknowledgments to the server so that the server knows it has successfully sent the packets to the right destination.

TFTP-TCP-Server

I found the server for TCP confusing to implement. I knew that a TCP connection would be more reliable than UDP, as there would be acknowledgment, sequence numbers, and block numbers, but I was confused because it felt as if I was repeating the implementation from the UDP server.

TFTP-TCP-Client

The same goes with the client, I used my previous methods for the UDP client, but instead of sending packets through a UDP connection, I wanted to use a TCP, but I lacked the knowledge to be able to successfully implement this feature.

Conclusion

In conclusion, I struggled with the implementation of the TFTP protocol on top of both transport protocols (UDP and TCP), but I feel as if I got a better understanding of how each protocol is used in theory.