

TCP & UDP FILE TRANSFER SERVER

Created by:

Nathan McDonald-Fortier 40134141

Tim Freiman 40091639

Submitted:

27/04/2023

OVERVIEW

The goal of this project is to make a server and a client program that can communicate using either UDP or TCP protocols. When the programs are initialized appropriate ports must be given, and the protocol to be used and whether to run in debug mode be specified. The server will service 4 types of requests, PUT, GET, CHANGE and HELP. See assignment documentation for specifics of each of these commands.

Server Program Design

The server creates sockets based on the type designated by the line arguments given. It will run with a buffer size of 4096 bytes, meaning any messages between clients or servers larger than the buffer will need to be broken up and sent over several messages. They will then be reassembled. In TCP mode if the client uses the Bye command the server will detect the connection loss and go back to waiting for a new connection. Debug mode enables the print() statements.

main(args): starts the server.

1. Passes arguments to be decoded by *parse_cli* function
2. Initializes the server in either TCP or UDP mode based on conn (connection type)
3. If in TCP mode the server connects to the socket
 - a. Sets the server to keep the connection alive
 - b. Bind the given port at given IP
 - c. Begins listening for client
 - d. When 1 client is heard from it accepts the connection
 - e. Create a client socket
 - f. Receives data and passes it to *handle_request* function
 - g. Repeat until the client socket connection is closed
 - h. When closed wait for new client
4. If In UDP mode the server connects to the socket
 - a. Bind the given port at the given IP
 - b. Wait for a request and pass to handle the request
 - c. loop

parse_cli(args): parses the command line arguments extracting connection type, IP, Port and debug state.

The socket is created here and passed to main(), the type is determined by the selection between UDP and TCP.

handle_request(client_socket, client_address, data, debug):
Handles incoming requests and sends back a response.

OPCODE

B7,b6,b5 are selected and read to select the appropriate request.

PUT REQUEST (000)

The client will send the data for the PUT request in chunks of the BUFFER_SIZE (4096 bytes), so the handle request intercepts the next incoming messages based on the size of the file.

GET REQUEST(001)

Reads the file and sends the file in chunks the size of the BUFFER_SIZE

CHANGE REQUEST(010)

Checks if the name exists and changes the name

HELP REQUEST(011)

Sends help message string.

Client Program Design

The server creates sockets based on the type designated by the line arguments given. It will run with a buffer size 4096 bytes, meaning any messages between clients or servers larger than the buffer will need to be broken up and sent over several messages. They will then be reassembled by the other party. Debug mode enables the `print()` statements.

main(args): starts the server.

1. Passes arguments to be decoded by *parse_cli* function
2. Initializes the server in either TCP or UDP mode based on conn (connection type)
3. If in TCP
 - a. Attempt to connect to a server, if not wait 1 sec and try again
 - b. When connected prompt the user for input
 - c. Parse and send to appropriate function,
 - d. Repeat
4. If in UDP mode prompt used for input
 - a. Parse and send to the appropriate function
 - b. Repeat

parse_cli(args): parses the command line arguments extracting connection type, IP, Port and debug state.

The socket is created here and passed to *main()*, the type is determined by the selection between UDP and TCP.

put(command)

1. Extract the filename and name length
2. Find the file and read
3. Build request
4. Give request to `send()`
5. `listen()` for response

get(command)

1. Extract the filename and length
2. Build request
3. Give request to `send()`
4. `listen()` for response

change(command)

1. Extract old and new filenames and lengths
2. Build request
3. Give request to send()
4. listen() for response

help()

1. Build request
2. Give request to send()
3. listen() for response

bye()

1. Close the current socket end program.

listen()

1. Wait for response from server
2. Extract upcode
3. Decode and print-related messages.
4. For get request:
 - a. The server will send the data for the Get request in chunks of the BUFFER_SIZE (4096 bytes), so the handle request intercepts the next incoming messages based on the size of the file.

send(data)

1. Based on size of request calculate number of chunks needed to send
2. Send data in chunks to server as needed

Wireshark Analysis

TCP

Question 1

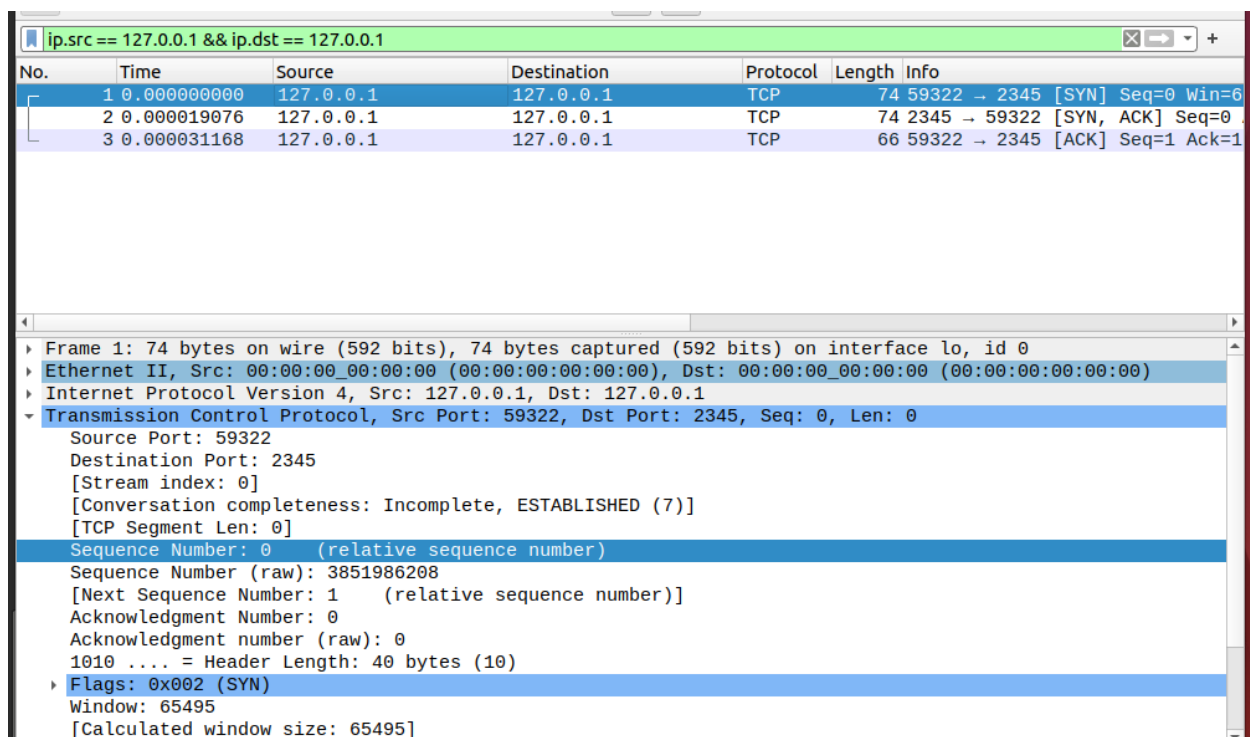
The TCP SYN segment that is used to initiate the TCP connection has the segment number 0.

Question 2

The first two sequence numbers of the TCP connection are both 0.

The first one was sent at time 0. The second one was sent at time 0.000019076.

The ACK for the first segment arrived at 0.000019076. The second ACK arrived at 0.00031168



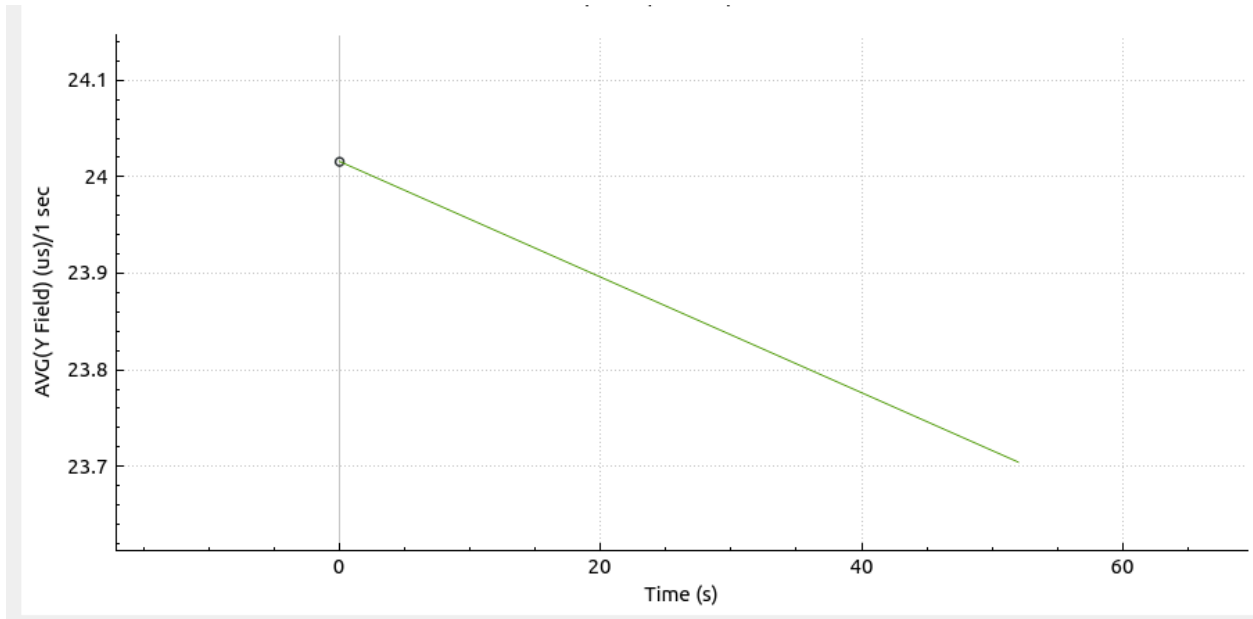
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	59322 → 2345 [SYN] Seq=0 Win=6
2	0.000019076	127.0.0.1	127.0.0.1	TCP	74	2345 → 59322 [SYN, ACK] Seq=0
3	0.000031168	127.0.0.1	127.0.0.1	TCP	66	59322 → 2345 [ACK] Seq=1 Ack=1

Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 59322, Dst Port: 2345, Seq: 0, Len: 0
Source Port: 59322
Destination Port: 2345
[Stream index: 0]
[Conversation completeness: Incomplete, ESTABLISHED (7)]
[TCP Segment Len: 0]
Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 3851986208
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 0
Acknowledgment number (raw): 0
1010 = Header Length: 40 bytes (10)
Flags: 0x002 (SYN)
Window: 65495
[Calculated window size: 65495]

Question 3

RTT1 = 0.000019347

RTT2 = 0.000022311



RTT Graph

Question 4

The receiver typically acknowledges all of the previous bytes sent.

Question 5

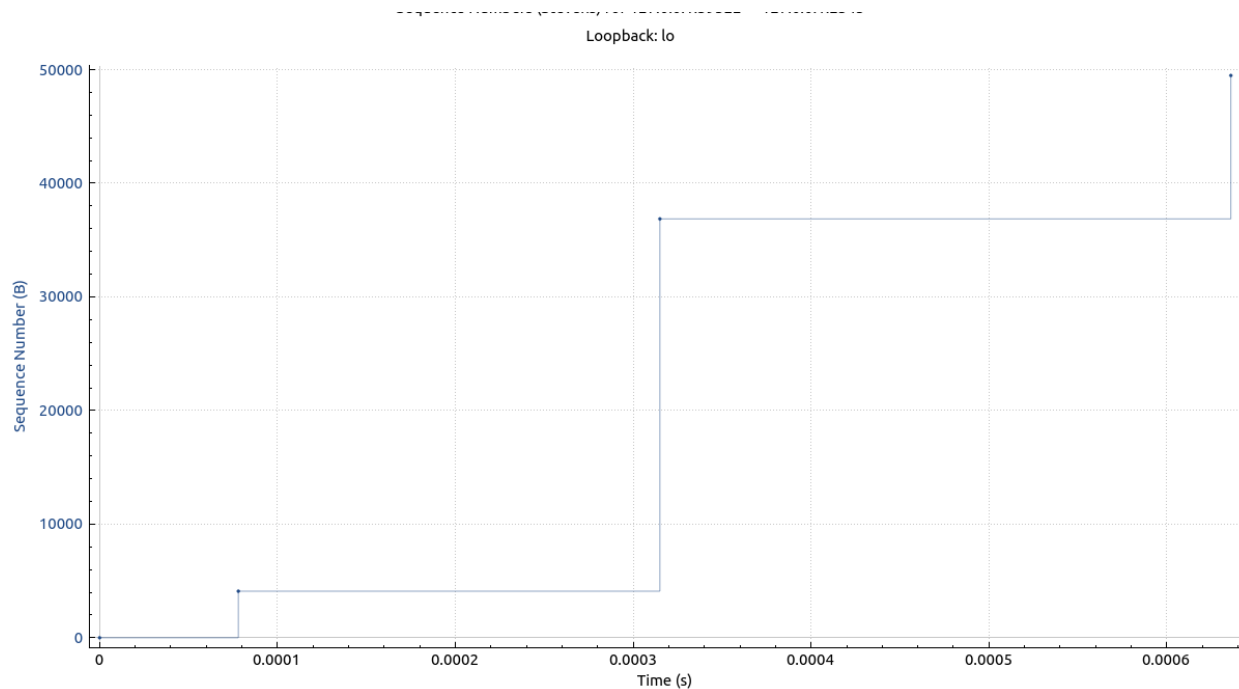
No, there are no instances where there was a retransmission of data. We would know there was a retransmission if the ACK number went down. It does not go down.

Question 6

The throughput of the TCP connection is $49473 \text{ bytes} / 0.000636748 \text{ seconds} = 77\,696\,357 \text{ Bps}$. To calculate this value, the file size is divided by the amount of time it took to transfer the file. This ensures that we are not counting the TCP overhead into the throughput.

Question 7

Time-Sequence Graph (Stevens)



UDP

Question 1

The UDP overhead can be calculated to be $4104 - 4096 = 8$ bytes.

Question 2

The maximum UDP payload size is 65527 bytes.

Source: <https://datatracker.ietf.org/doc/html/rfc768>

Question 3

The UDP protocol number is 17 in decimal and 0x11 in hexadecimal

Source: <https://datatracker.ietf.org/doc/html/rfc768>