



SOFE 3850: Computer Networks

Project Report

Group: Project Groups 4

Members:

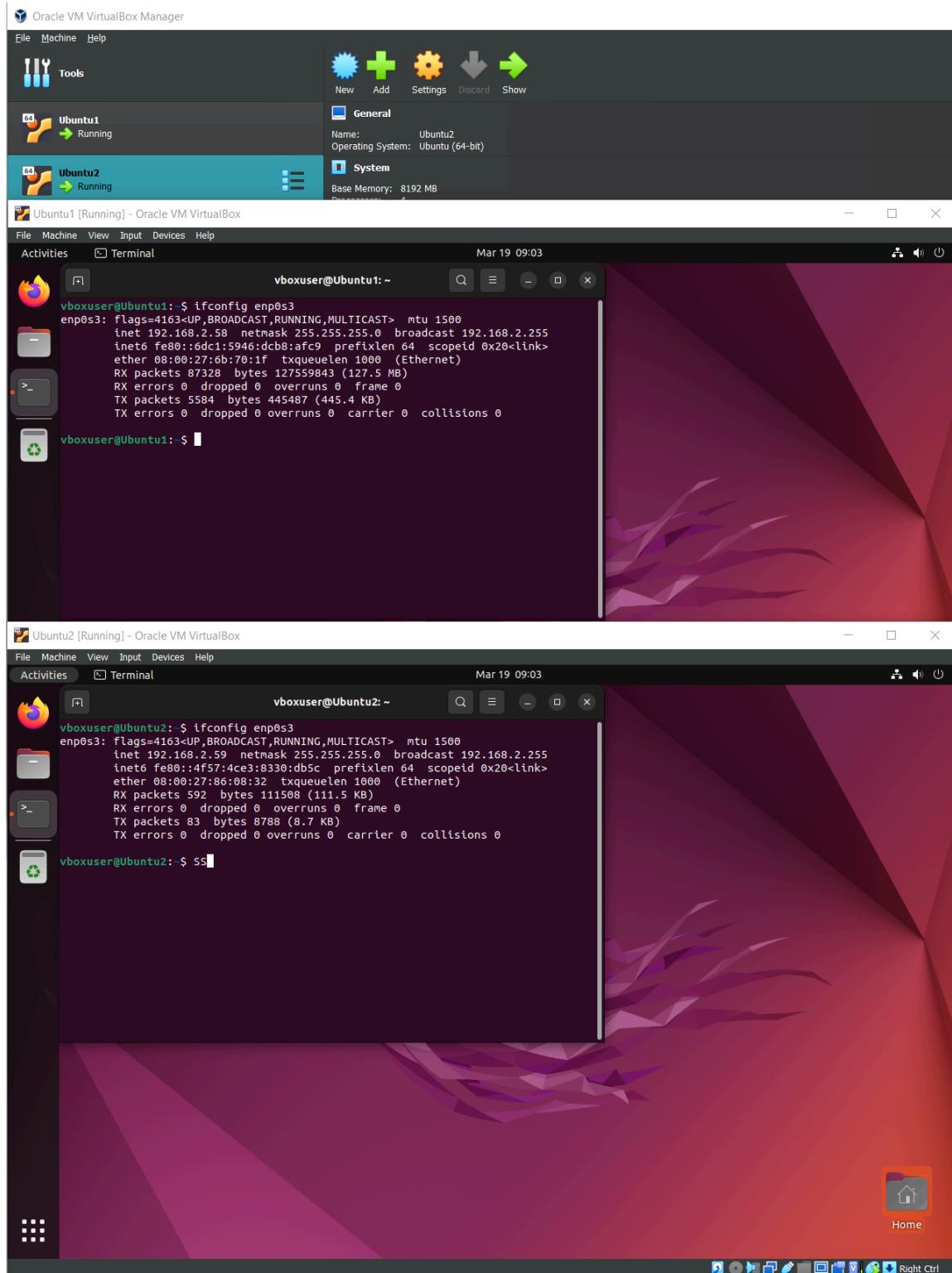
Name	Student ID
Jason Stuckless	100248154
Okiki Ojo	100790236

Table of Contents

Part 1: Setup of VMs and the Virtual Network Environment	3
1. Virtual Machines	3
2. Successful Pings	4
3. Web Server	5
4. Wireshark Captured Data	6
Part 2: TCP Server	8
1. Wireshark Captured Data	8
2. Child Processes	8
3. Part III Programs	9
Part 3: File Download Application Based on TCP	10
1. File Download	10
2. Error Message	11
3. Program Explanation	11
Part 4: UDP Server Implementation	13
1. Steps 1 to 6	13
2. Question 7	15
3. UDP vs TCP	15
4. Concurrent vs. Non-Concurrent	15
5. Program Demonstration	16
6. Program Explanation	22

Part 1: Setup of VMs and the Virtual Network Environment

1. Virtual Machines



2. Successful Pings

The screenshot displays three separate windows from Oracle VM VirtualBox, each showing a terminal session on an Ubuntu host. The top window, titled "Ubuntu1 [Running] - Oracle VM VirtualBox", shows the output of the "ifconfig" command for interface enp0s3 and a successful ping to 192.168.2.59. The middle window, titled "Ubuntu2 [Running] - Oracle VM VirtualBox", shows the output of the "ifconfig" command for interface enp0s3 and a successful ping to 192.168.2.58. The bottom window, titled "Command Prompt", shows the output of the "ping" command from a Windows host (C:\Users\jason) to 192.168.2.58 and 192.168.2.59, both of which return successful responses.

```
vboxuser@Ubuntu1:~$ ifconfig enp0s3
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.2.58 netmask 255.255.255.0 broadcast 192.168.2.255
          inet6 fe80::6dc1:5946:dbcb:af9 prefixlen 64 scopeid 0x20<link>
            ether 08:00:27:6b:70:1f txqueuelen 1000  (Ethernet)
              RX packets 493 bytes 267987 (267.9 KB)
              RX errors 0 dropped 0 overruns 0 frame 0
              TX packets 297 bytes 55526 (55.5 KB)
              TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vboxuser@Ubuntu1:~$ ping 192.168.2.59
PING 192.168.2.59 (192.168.2.59) 56(84) bytes of data.
64 bytes from 192.168.2.59: icmp_seq=1 ttl=64 time=0.286 ms
64 bytes from 192.168.2.59: icmp_seq=2 ttl=64 time=0.188 ms
64 bytes from 192.168.2.59: icmp_seq=3 ttl=64 time=0.202 ms
64 bytes from 192.168.2.59: icmp_seq=4 ttl=64 time=0.195 ms
64 bytes from 192.168.2.59: icmp_seq=5 ttl=64 time=0.192 ms
64 bytes from 192.168.2.59: icmp_seq=6 ttl=64 time=0.164 ms
64 bytes from 192.168.2.59: icmp_seq=7 ttl=64 time=0.151 ms
^C
--- 192.168.2.59 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6184ms
rtt min/avg/max/mdev = 0.151/0.196/0.286/0.040 ms
vboxuser@Ubuntu1:~$

vboxuser@Ubuntu2:~$ ifconfig enp0s3
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.2.58 netmask 255.255.255.0 broadcast 192.168.2.255
          inet6 fe80::4f57:4ce3:8330:db5c prefixlen 64 scopeid 0x20<link>
            ether 08:00:27:86:08:32 txqueuelen 1000  (Ethernet)
              RX packets 183 bytes 37230 (37.2 KB)
              RX errors 0 dropped 0 overruns 0 frame 0
              TX packets 70 bytes 10013 (10.0 KB)
              TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vboxuser@Ubuntu2:~$ ping 192.168.2.58
PING 192.168.2.58 (192.168.2.58) 56(84) bytes of data.
64 bytes from 192.168.2.58: icmp_seq=1 ttl=64 time=0.172 ms
64 bytes from 192.168.2.58: icmp_seq=2 ttl=64 time=0.196 ms
64 bytes from 192.168.2.58: icmp_seq=3 ttl=64 time=0.247 ms
64 bytes from 192.168.2.58: icmp_seq=4 ttl=64 time=0.170 ms
64 bytes from 192.168.2.58: icmp_seq=5 ttl=64 time=0.187 ms
^C
--- 192.168.2.58 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4075ms
rtt min/avg/max/mdev = 0.170/0.194/0.247/0.028 ms
vboxuser@Ubuntu2:~$ s

[Output from Command Prompt]
C:\Users\jason>ping 192.168.2.58

Pinging 192.168.2.58 with 32 bytes of data:
Reply from 192.168.2.58: bytes=32 time<1ms TTL=64

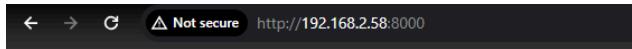
Ping statistics for 192.168.2.58:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\jason>ping 192.168.2.59

Pinging 192.168.2.59 with 32 bytes of data:
Reply from 192.168.2.59: bytes=32 time<1ms TTL=64

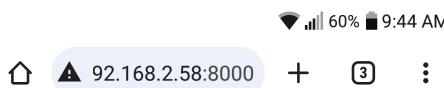
Ping statistics for 192.168.2.59:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

3. Web Server



Directory listing for /

-
- [.bash_history](#)
 - [.bash_logout](#)
 - [.bashrc](#)
 - [.cache/](#)
 - [.config/](#)
 - [.local/](#)
 - [.profile](#)
 - [.sudo_as_admin_successful](#)
 - [Desktop/](#)
 - [Documents/](#)
 - [Downloads/](#)
 - [Music/](#)
 - [Pictures/](#)
 - [Public/](#)
 - [snap/](#)
 - [Templates/](#)
 - [Videos/](#)
-

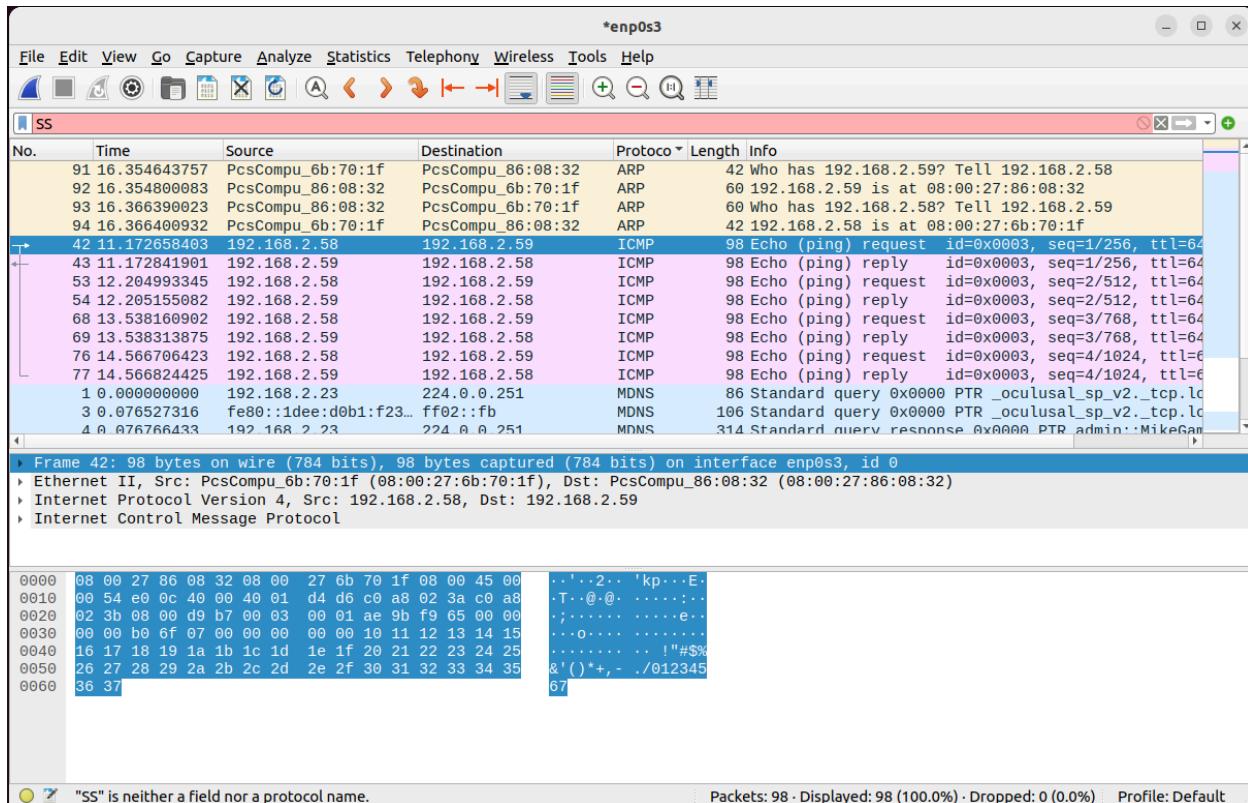


Directory listing for /

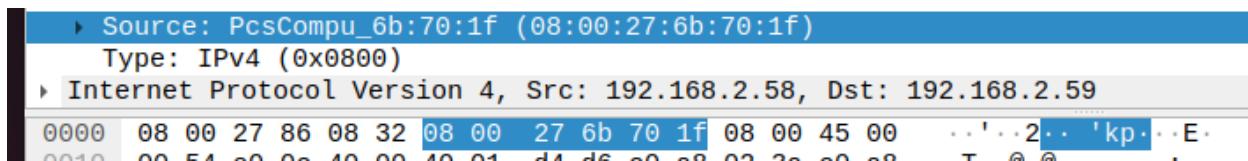
-
- [.bash_history](#)
 - [.bash_logout](#)
 - [.bashrc](#)
 - [.cache/](#)
 - [.config/](#)
 - [.local/](#)
 - [.profile](#)
 - [.sudo_as_admin_successful](#)
 - [Desktop/](#)
 - [Documents/](#)
 - [Downloads/](#)
 - [Music/](#)
 - [Pictures/](#)
 - [Public/](#)
 - [snap/](#)
 - [Templates/](#)
 - [Videos/](#)
-



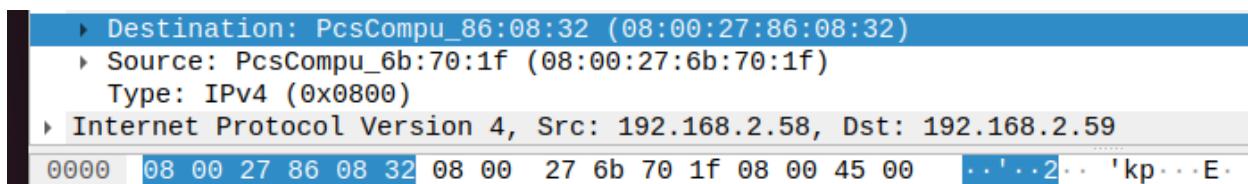
4. Wireshark Captured Data



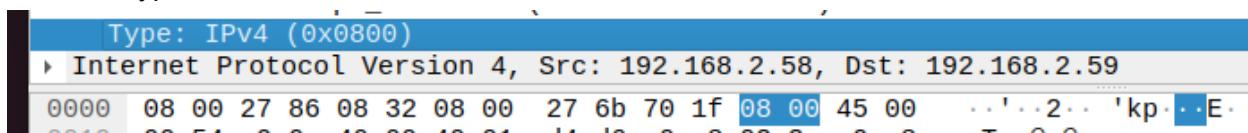
Source Ethernet Address: 08:00:27:6b:70:1f



Destination Ethernet Address: 08:00:27:86:08:32



Ethernet Type Field: 08:00



Source IP Address: c0:a8:02:3a (192.168.2.58)

Source Address: 192.168.2.58	
Destination Address: 192.168.2.59	
Internet Control Message Protocol	
0000	08 00 27 86 08 32 08 00 27 6b 70 1f 08 00 45 00
0010	00 54 e0 0c 40 00 40 01 d4 d6 c0 a8 02 3a c0 a8
0020	02 3b 08 00 d9 b7 00 03 00 01 ae 9b f9 65 00 00

Destination IP Address: c0:a8:02:3b (192.168.2.59)

Destination Address: 192.168.2.59	
Internet Control Message Protocol	
0000	08 00 27 86 08 32 08 00 27 6b 70 1f 08 00 45 00
0010	00 54 e0 0c 40 00 40 01 d4 d6 c0 a8 02 3a c0 a8
0020	02 3b 08 00 d9 b7 00 03 00 01 ae 9b f9 65 00 00

IP Protocol Field: 01 (ICMP)

Protocol: ICMP (1)	
Header Checksum: 0xd4d6 [validation disabled]	
[Header checksum status: Unverified]	
Source Address: 192.168.2.58	
0000	08 00 27 86 08 32 08 00 27 6b 70 1f 08 00 45 00
0010	00 54 e0 0c 40 00 40 01 d4 d6 c0 a8 02 3a c0 a8
0020	02 3b 08 00 d9 b7 00 03 00 01 ae 9b f9 65 00 00

ICMP Type Field, Ping Request: 08

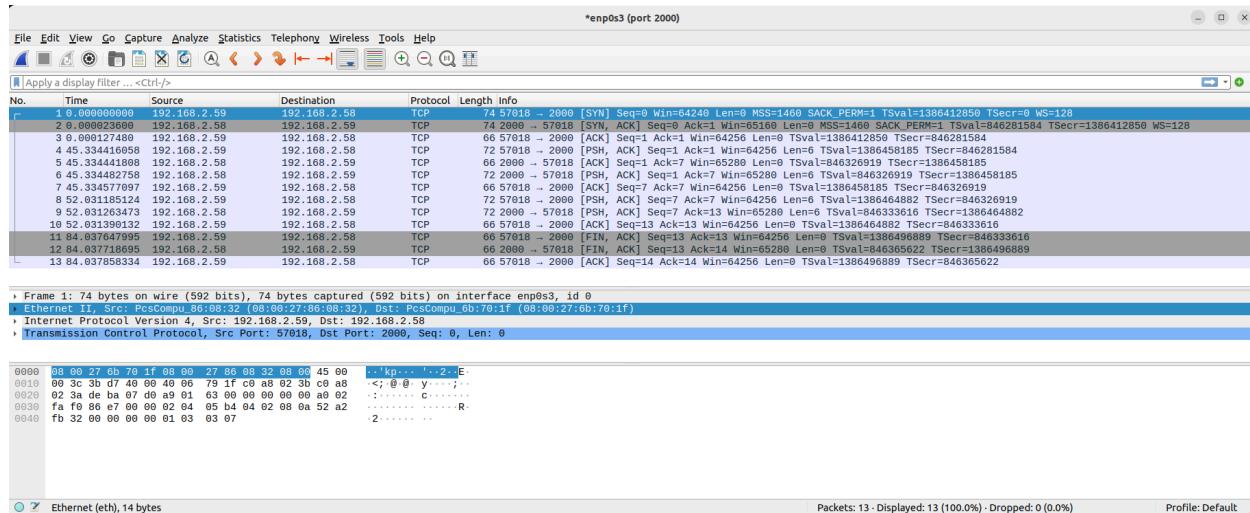
Internet Control Message Protocol	
Type: 8 (Echo (ping) request)	
Code: 0	
0000	08 00 27 86 08 32 08 00 27 6b 70 1f 08 00 45 00
0010	00 54 e0 0c 40 00 40 01 d4 d6 c0 a8 02 3a c0 a8
0020	02 3b 08 00 d9 b7 00 03 00 01 ae 9b f9 65 00 00

ICMP Type Field, Ping Response: 00

Internet Control Message Protocol	
Type: 0 (Echo (ping) reply)	
Code: 0	
0000	08 00 27 6b 70 1f 08 00 27 86 08 32 08 00 45 00
0010	00 54 bb f2 00 00 40 01 38 f1 c0 a8 02 3b c0 a8
0020	02 3a 00 00 e1 b7 00 03 00 01 ae 9b f9 65 00 00
0030	00 00 b0 6f 07 00 00 00 00 00 10 11 12 13 14 15

Part 2: TCP Server

1. Wireshark Captured Data



TCP Connection Packets: Packets No. 1-3

TCP Termination Packets: Packets No. 11-13

2. Child Processes

```
vboxuser@Ubuntu1:~$ ps -a
  PID TTY      TIME CMD
 1363 tty2    00:00:00 gnome-session-b
 2784 pts/0    00:00:00 echo_server
 2818 pts/1    00:00:00 ps
vboxuser@Ubuntu1:~$ ps -a
  PID TTY      TIME CMD
 1363 tty2    00:00:00 gnome-session-b
 2784 pts/0    00:00:00 echo_server
 2819 pts/0    00:00:00 echo_server
 2820 pts/1    00:00:00 ps
vboxuser@Ubuntu1:~$ ps -a
  PID TTY      TIME CMD
 1363 tty2    00:00:00 gnome-session-b
 2784 pts/0    00:00:00 echo_server
 2819 pts/0    00:00:00 echo_server
 2860 pts/0    00:00:00 echo_server
 2861 pts/1    00:00:00 ps
vboxuser@Ubuntu1:~$ netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 Ubuntu1:cisco-sccp      Ubuntu2:55838        ESTABLISHED
tcp      0      0 Ubuntu1:cisco-sccp      Ubuntu2:36416        ESTABLISHED
```

3. Part III Programs

The image shows two terminal windows side-by-side. The top window, titled 'vboxuser@Ubuntu1: ~', displays the command `vboxuser@Ubuntu1:~$./hello_server 3000`. The bottom window, titled 'vboxuser@Ubuntu2: ~\$', displays the command `vboxuser@Ubuntu2:~$./hello_client 192.168.2.58 3000`, followed by the output 'Hello'.

hello_server.c and hello_client.c are copies of echo_server.c and echo_client.c except for the following changes:

Line 67 in echo_server.c changed in hello_server.c:

```
exit(sendHello(new_sd));
```

Added in hello_server.c before break; on line 70 of echo_server.c:

```
exit(0);
```

sendHello() function:

```
int sendHello (int sd) {
    char *bp, buf[BUFSIZE];

    exit(write(sd, "Hello", 5));
    close(sd);

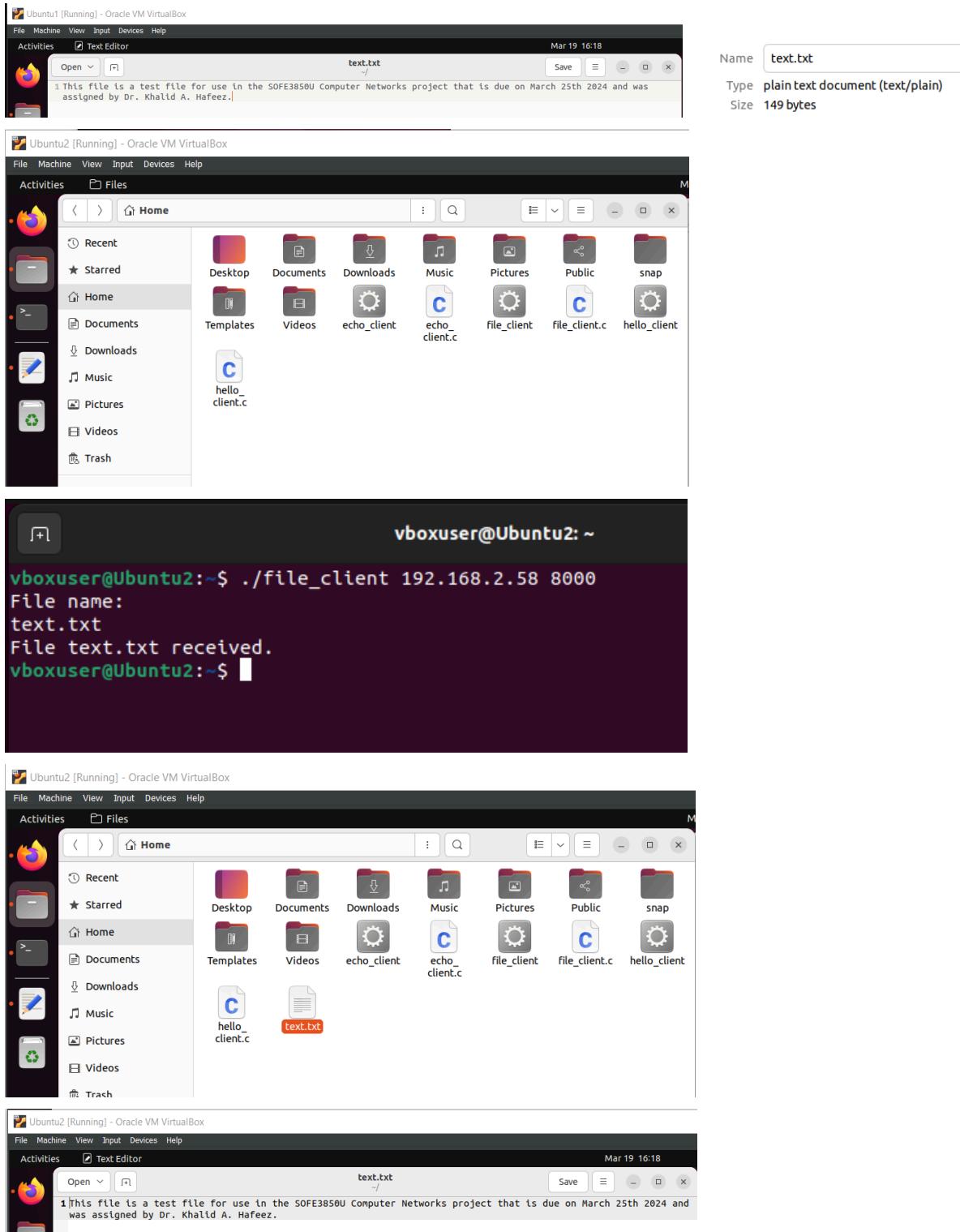
    return(0);
}
```

Line 61 to 73 in echo_client.c changed in hello_client.c:

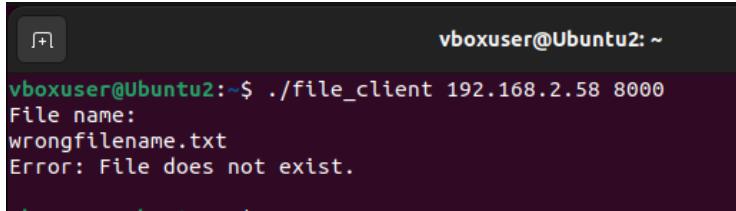
```
while(n=read(sd, sbuf, BUFSIZE)){
    printf("%s\n", sbuf);
    close(sd);
    break;
}
```

Part 3: File Download Application Based on TCP

1. File Download



2. Error Message



```
vboxuser@Ubuntu2:~$ ./file_client 192.168.2.58 8000
File name:
wrongfilename.txt
Error: File does not exist.
```

3. Program Explanation

Again, file_server.c is a copy of echo_server.c except line 67 changed to:

```
exit(transferFile(new_sd));
```

And new function transferFile is:

```
int transferFile(int sd) {
    char *fileContents = 0, *bp, filename[BUFLEN], outgoing[BUFLEN];
    int n, bytes_to_read, fileLength;
    FILE *fptr;
    long length;
    while(n = read(sd, filename, BUFLEN)) {
        fptr = fopen(filename, "rb");
        break;
    }
    if (fptr == NULL) {
        write(sd, "eError: File does not exist.\n", n);
    }
    else {
        fseek (fptr, 0, SEEK_END);
        length = ftell(fptr);
        fseek (fptr, 0, SEEK_SET);
        fileContents = malloc(length);
        if (fileContents)
            fread(fileContents, 1, length, fptr);
        snprintf(outgoing, sizeof outgoing, "%s", fileContents);
        fileLength = strlen(outgoing);
        for (int i = 0; i < fileLength; i += 100) {
            memmove(outgoing, outgoing+i, fileLength);
            write(sd, outgoing, 100);
        }
    }
    return(0);
}
```

And file_client.c is a copy of echo_client.c except after Line 60 the code is changed to:

```
FILE *fptr;
printf("File name: \n");
fflush(stdout);
scanf("%s", sbuf);
write(sd, sbuf, BUFLEN);

while(n=read(sd, rbuf, BUFLEN)) {
    if(rbuf[0] == 'e') { // Error message
        memmove(rbuf, rbuf+1, strlen(rbuf));
        printf("%s", rbuf);
    }
    else if(rbuf[0] == 'f') { // File transfer
        memmove(rbuf, rbuf+1, strlen(rbuf));
        fptr = fopen(sbuf, "a");
        fprintf(fptr, "%s", rbuf);
        while(n=read(sd, rbuf, BUFLEN)) { // Read other packets
            fprintf(fptr, "%s", rbuf);
            printf("File %s received.\n", sbuf);
        }
    }
}
```

How server transfers the data from the file to the client:

```
for (int i = 0; i < fileLength; i += 100) {
    memmove(outgoing, outgoing+i, fileLength);
    write(sd, outgoing, 100);
}
```

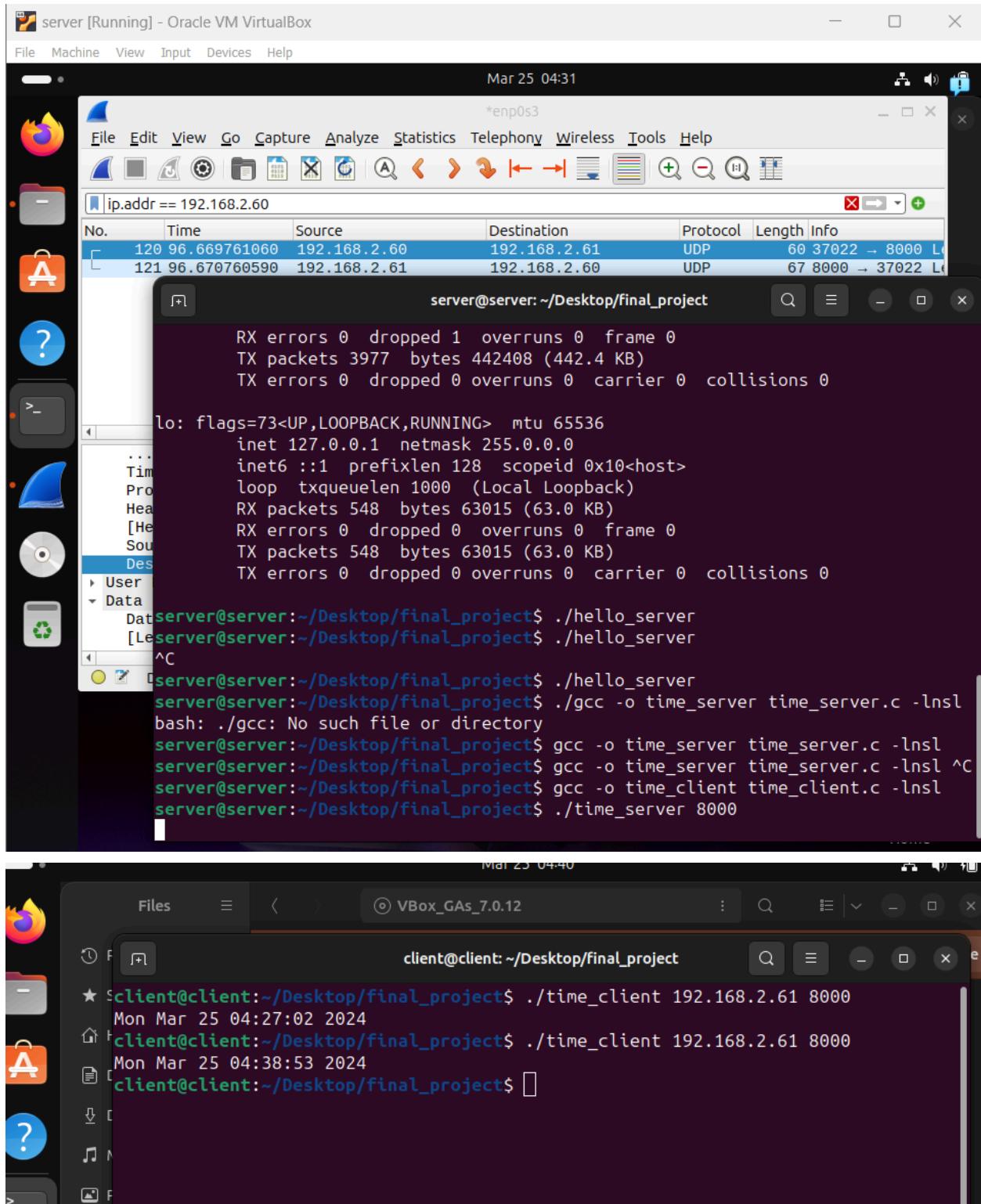
The server stops transmitting and terminates connection after either sending an error message or after it has sent the file in 100 byte increments and reached the end of the file (i < fileLength in for loop).

The client stops receiving once it has received an error message or the requested file.

Part 4: UDP Server Implementation

1. Steps 1 to 6

Compiling and Running the time server and client



The screenshot displays two terminal windows running on a Linux desktop. The top window is titled "server [Running] - Oracle VM VirtualBox" and shows a terminal session for a UDP server. The bottom window is titled "client@client: ~/Desktop/final_project" and shows a terminal session for a UDP client.

Server Terminal Output:

```
server@server:~/Desktop/final_project$ ./hello_server
server@server:~/Desktop/final_project$ ./hello_server
server@server:~/Desktop/final_project$ ./gcc -o time_server time_server.c -lnsl
bash: ./gcc: No such file or directory
server@server:~/Desktop/final_project$ gcc -o time_server time_server.c -lnsl
server@server:~/Desktop/final_project$ gcc -o time_server time_server.c -lnsl ^C
server@server:~/Desktop/final_project$ gcc -o time_client time_client.c -lnsl
server@server:~/Desktop/final_project$ ./time_server 8000
```

Client Terminal Output:

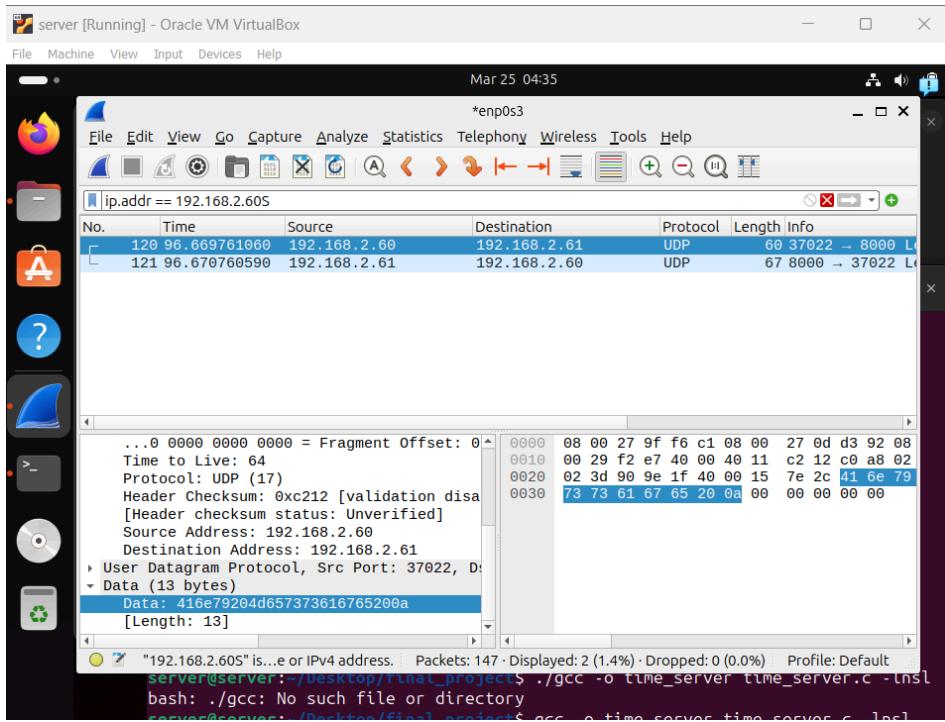
```
client@client:~/Desktop/final_project$ ./time_client 192.168.2.61 8000
Mon Mar 25 04:27:02 2024
client@client:~/Desktop/final_project$ ./time_client 192.168.2.61 8000
Mon Mar 25 04:38:53 2024
client@client:~/Desktop/final_project$
```

Both terminals also show network traffic capture and statistics. The server terminal shows the following statistics:

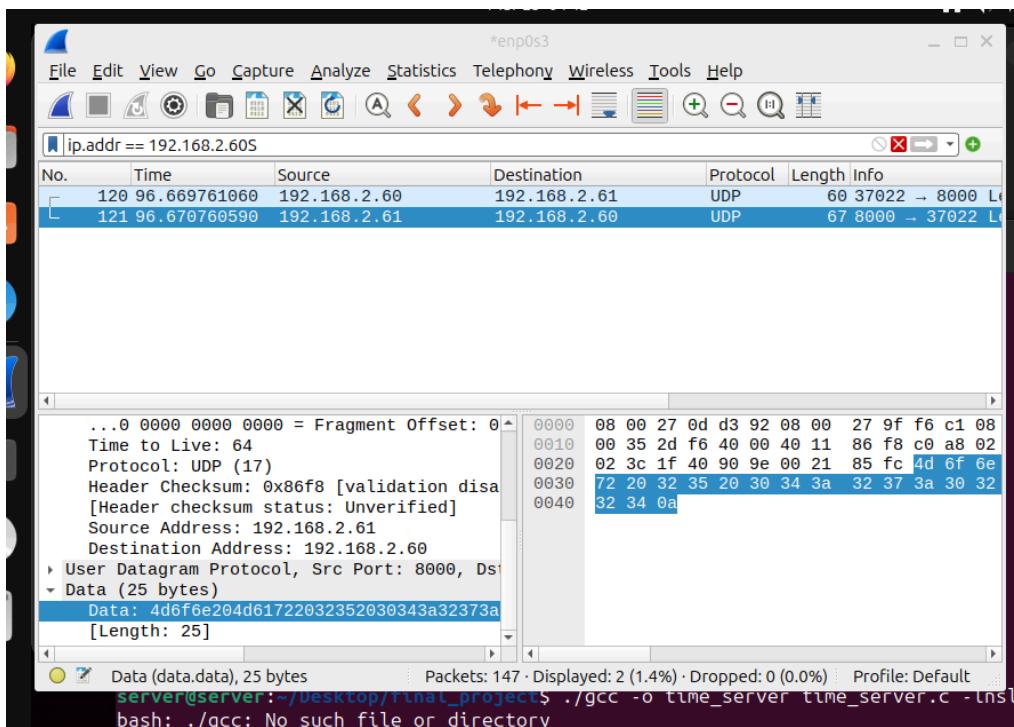
```
RX errors 0 dropped 1 overruns 0 frame 0
TX packets 3977 bytes 442408 (442.4 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
      inet6 ::1 prefixlen 128 scopeid 0x10<host>
          loop txqueuelen 1000 (Local Loopback)
          RX packets 548 bytes 63015 (63.0 KB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 548 bytes 63015 (63.0 KB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Using wireshark to view the message sent from the client `192.168.2.60` to the server `192.168.2.61`



Using wireshark to view the vice-versa the datetime sent to the client `192.168.2.60` from the server `192.168.2.61`



The data sent between both the client and server are as so

Hex to String

A screenshot of a web-based tool titled "Hex to String". The interface includes a text area at the top containing two hex strings:

```
416e79204d657373616765200a  
4d6f6e204d61722032352030343a32373a303  
220323032340a
```

Below the text area are three buttons: "Auto" (with a checked checkbox), "Hex to String" (highlighted in green), and "File..". There is also a "Load URL" button. At the bottom left, there is a timestamp: "Any Message" followed by "Mon Mar 25 04:27:02 2024".

2. Question 7

It's a means of notifying the server that the client is ready for the datetime information as there is no formal connection handshake between the server and client, so having those messages help a lot in ensuring that both the client and server are ready.

3. UDP vs TCP

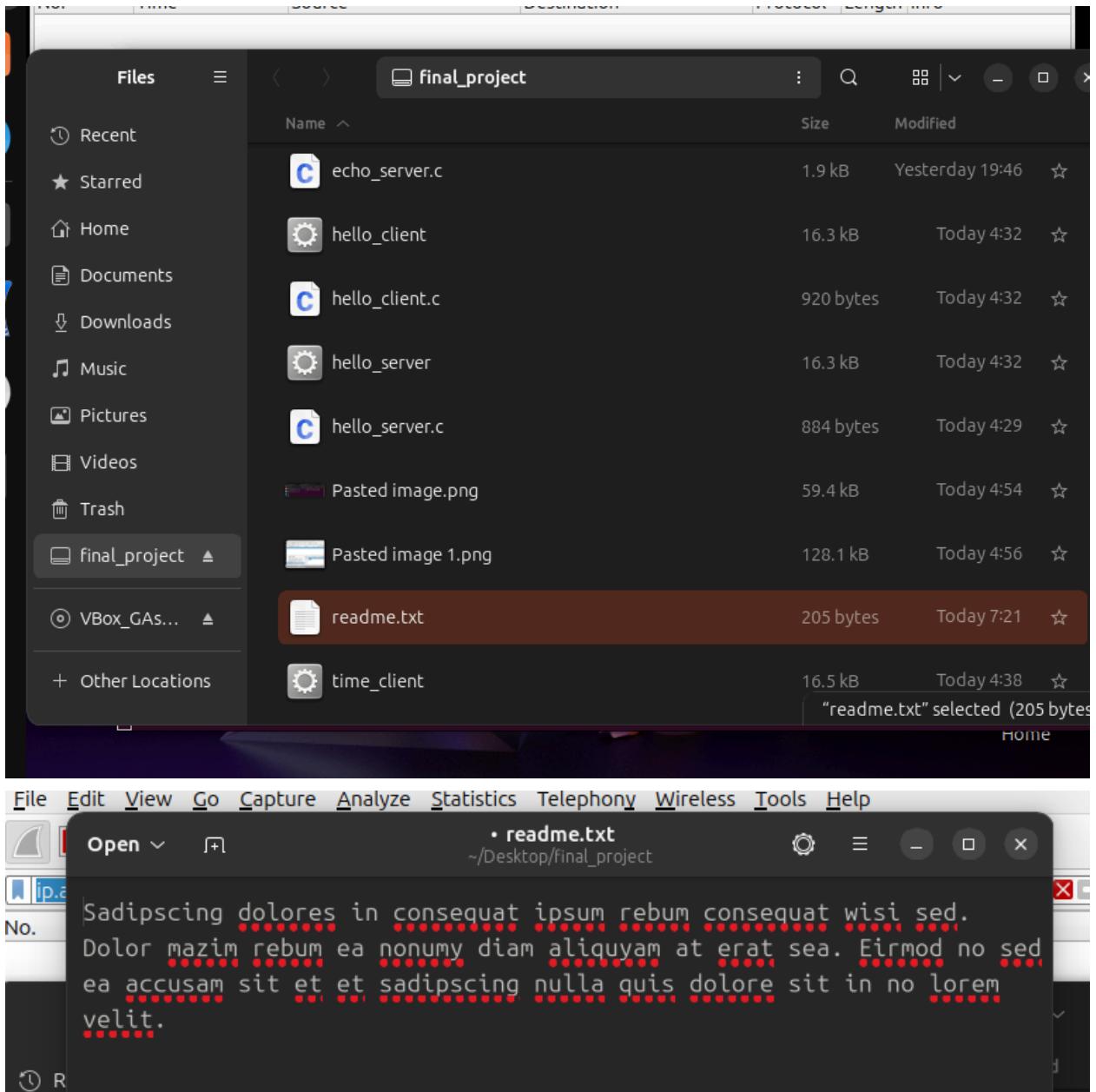
UDP is the more appropriate protocol, as the up to date information is more important than verifying that the information has been received, if a packet gets dropped in UDP there will be another packet sent that will be more accurate to the current date, but if we used TCP the request would need to be re-sent again and will cause the client to be out of sync with the time server.

4. Concurrent vs. Non-Concurrent

Concurrency isn't as necessary in UDP as it is in TCP as UDP is connectionless so we never really need to wait for a client to receive the up to date response before we move on to the next client, but to ensure all clients have the most up to date information at any given time creating a number of processes to allow for UDP responses to multiple clients at the same time is very useful, so I would lean toward concurrent. I should note that this would add quite some complexity to the entire process and would potentially require a clients table, ipc connections, semaphores and/or mutex locks, to ensure each concurrent process handles a different client, as well as to have continuity with clients.

5. Program Demonstration

The readme.txt file to download from the server (we used lorem ipsum text, its larger than 100 bytes)



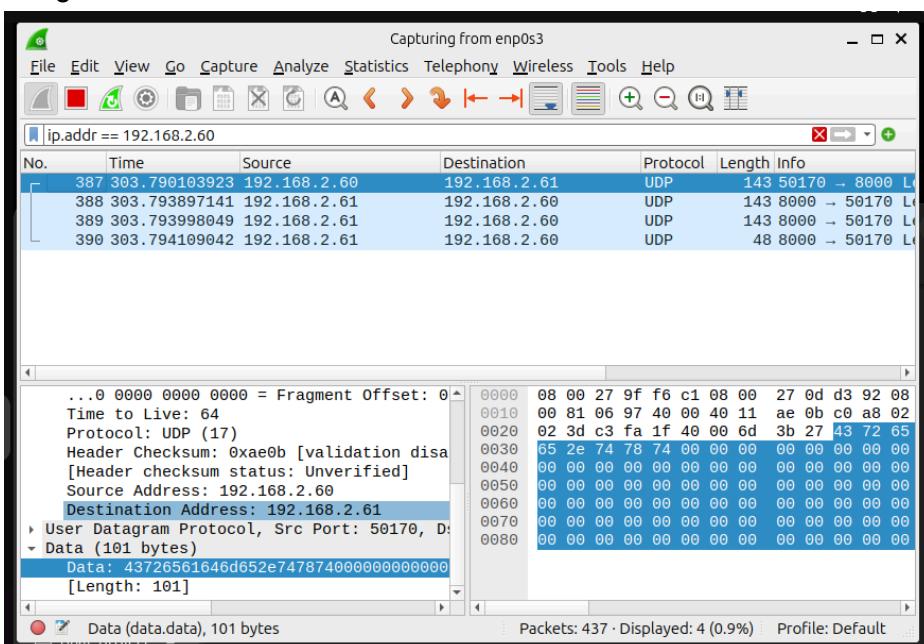
Startup the udp server with port 8000

```
server@server:~/Desktop/final_project$ gcc -o udp_server udp_server.c -lssl && g  
cc -o udp_client udp_client.c -lssl  
server@server:~/Desktop/final_project$ ./udp_server 8000  
- Error transferring file  
- Error transferring file  
^C  
server@server:~/Desktop/final_project$ gcc -o udp_server udp_server.c -lssl && g  
cc -o udp_client udp_client.c -lssl  
server@server:~/Desktop/final_project$ ./udp_server 8000  
^C  
server@server:~/Desktop/final_project$ gcc -o udp_server udp_server.c -lssl && g  
cc -o udp_client udp_client.c -lssl  
server@server:~/Desktop/final_project$ ./udp_server 8000
```

Startup the udp client with port 8000 and the ip address of the server. We then enter `readme.txt` as the filename of the file to download

```
client@client:~/Desktop/final_project$ ./udp_client 192.168.2.61 8000
Enter filename: readme.txt
File transfer complete.
Enter filename: 
```

Wireshark network capture of the Filename PDU with the filename readme.txt as the pdu data being sent from client to server



Hex to String

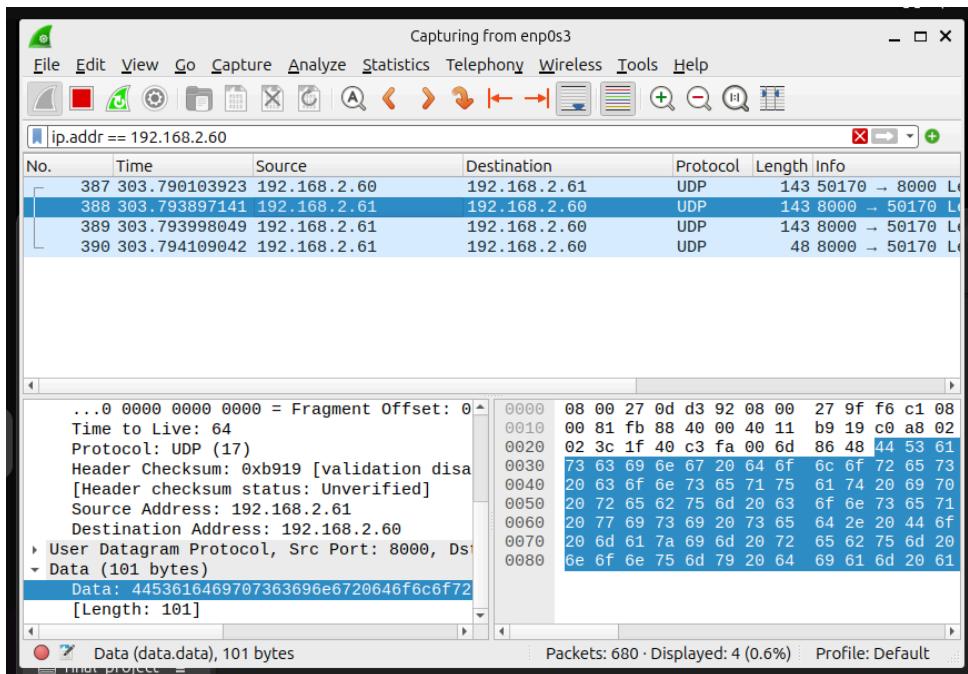
43726561646d652e7478740000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000

Auto  Hex to String  File..

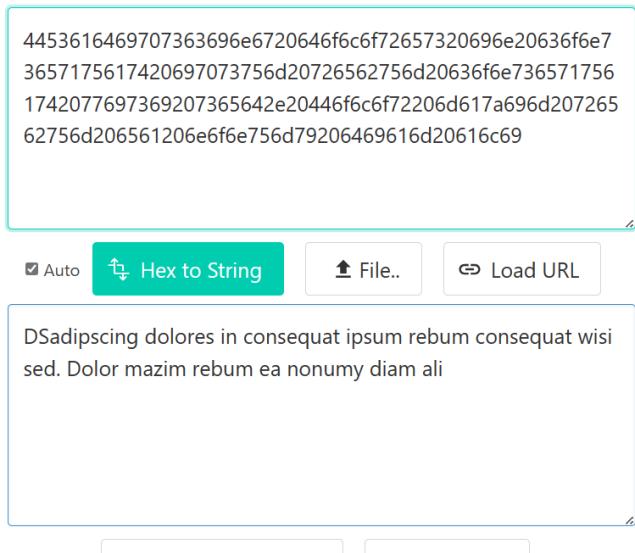
 Load URL

Creadme.txt

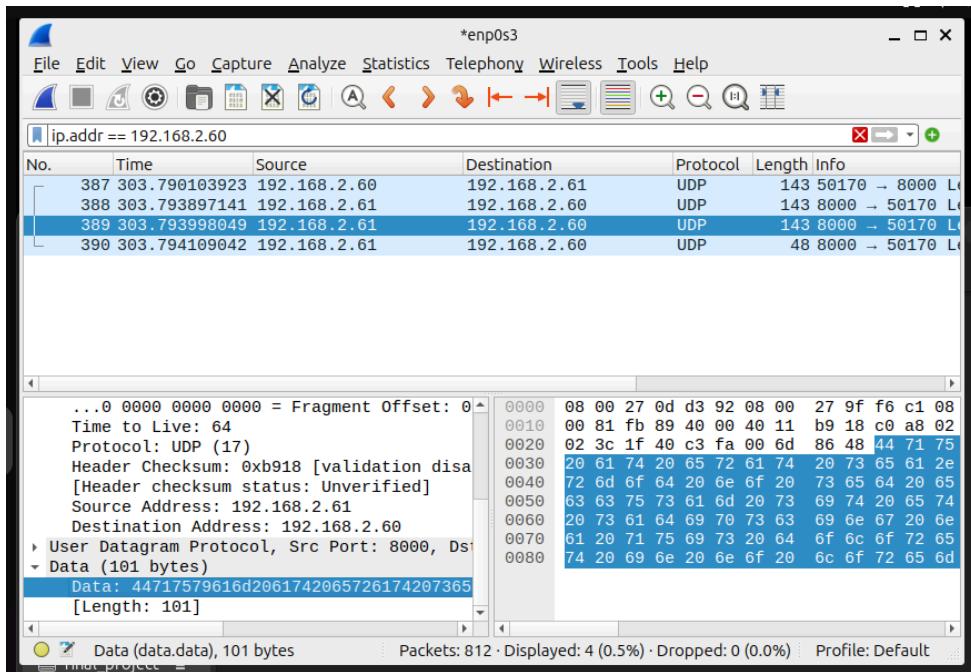
The first 100 bytes of the readme.txt file being sent from the server to the client using the Data PDU type



Hex to String



The second 100 bytes of characters from readme.txt from the server to the client as a Data PDU type



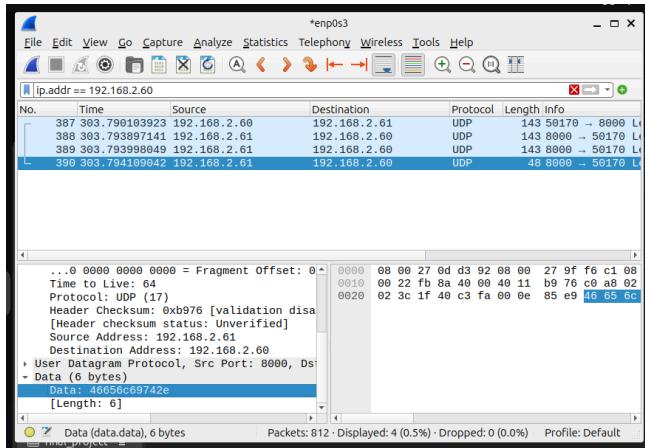
Hex to String

```
44717579616d2061742065726174207365612e204569726d6f64  
206e6f20736564206561206163637573616d2073697420657420  
65742073616469707363696e67206e756c6c6120717569732064  
6f6c6f72652073697420696e206e6f206c6f72656d2076
```

Auto Hex to String File.. Load URL

Dquyam at erat sea. Eirmod no sed ea accusam sit et et
sadipscing nulla quis dolore sit in no lorem v

The last couple bytes of the readme.txt file from the server to the client as a Final PDU type



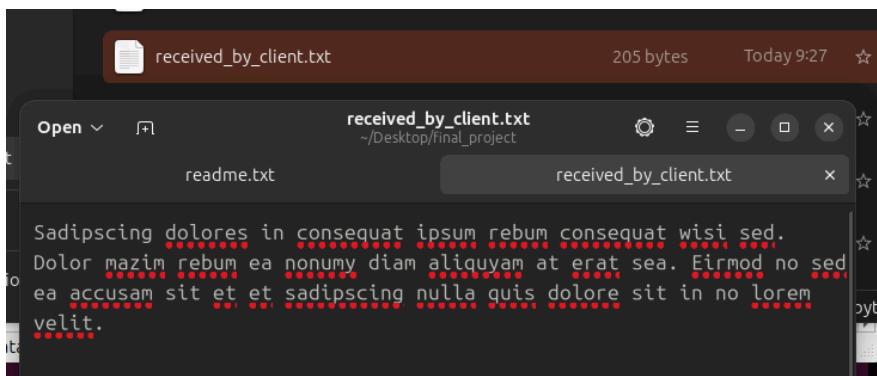
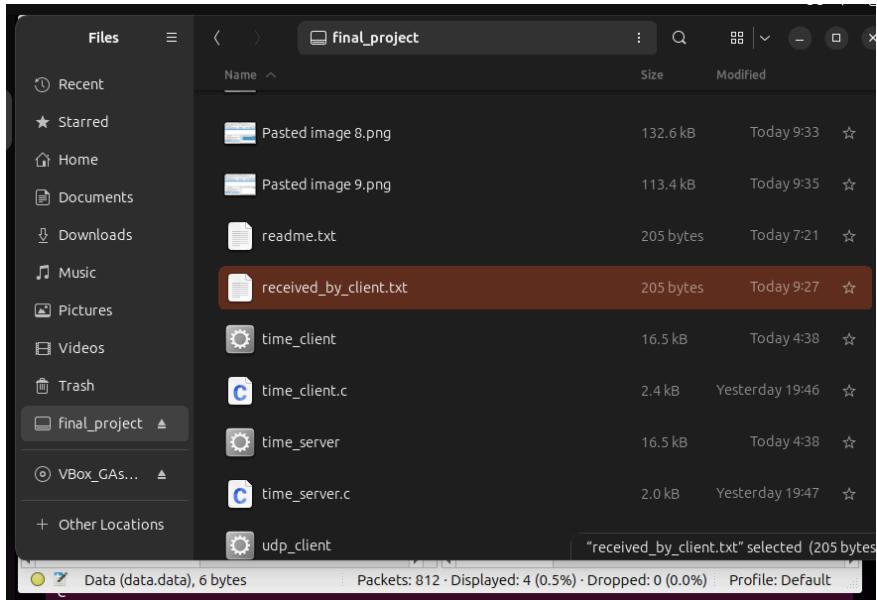
Hex to String

```
46656c69742e
```

Auto Hex to String File.. Load URL

Felit.

Here is the new file created from the server to client download it's called received_by_client.txt



Downloading multiple separate files

```

client@client:~/Desktop$ ./udp_client 192.168.2.61
8000
Enter filename: readme.txt
File transfer complete.
Enter filename: ^
client@client:~/Desktop/final_project$ cd ..
client@client:~/Desktop$ ./final_project/udp_client 192.168.2.61
8000
Enter filename: readme.txt
File transfer complete.
Enter filename: readme

```

The terminal window shows the client sending a file named 'readme.txt' to the server and receiving it back. The file content is identical to the one shown in the text editor above.

client@client:~/Desktop

```
-rwxrwx--- 1 root vboxsf 4616 Mar 25 2024 udp_server.c
drwxrwx--- 1 root vboxsf 0 Mar 25 08:23 .vscode
client@client:~/Desktop/final_project$ ./udp_client 192.168.2.61
8000
Enter filename: readme.txt
File transfer complete.
Enter filename: ^C
client@client:~/Desktop/final_project$ cd ..
client@client:~/Desktop$ ./final_project/udp_client 192.168.2.61
8000
Enter filename: readme.txt
File transfer complete.
Enter filename: readme2.txt
File transfer complete.
Enter filename: 
```

received_by_client.txt

Open ~ Desktop

```
-----  
Ipsum placerat vero sed accusam dolor kasd clita diam. Voluptua  
eirmod et aliquyam ullamcorper. Nibh aliquyam ipsum takimata et est  
labore rebum ut diam lorem dolor luptatum feugait gubergren elitr  
amet lorem. Duis ea kasd sed. Tempor velit exerci sed nonumy et  
accumsan consetetur no vero rebum eu no eos ad et nisl. Dolore  
adipiscing minim magna ea kasd ea ipsum nonumy clita. Gubergren  
suscipit nulla dolor consectetur odio ea ea dolor diam eum et  
voluptua sadipscing eum ea tincidunt et nulla.  
  
-----  
Sadipscing dolores in consequat ipsum rebum consequat wisi sed.  
Dolor mazim rebum ea nonumy diam aliquyam at erat sea. Eirmod no sed  
ea accusam sit et et sadipscing nulla quis dolore sit in no lorem  
velit.
```

received_by_client.txt

Home final_project

readme.txt readme2.txt readme2.txt

File Edit View

```
-----  
Ipsum placerat vero sed accusam dolor kasd clita diam. Voluptua eirmod et  
aliquyam ullamcorper. Nibh aliquyam ipsum takimata et est labore rebum ut diam  
lorem dolor luptatum feugait gubergren elitr amet lorem. Duis ea kasd sed.  
Tempor velit exerci sed nonumy et accumsan consetetur no vero rebum eu no eos  
ad et nisl. Dolore adipiscing minim magna ea kasd ea ipsum nonumy clita.  
Gubergren suscipit nulla dolor consectetur odio ea ea dolor diam eum et  
voluptua sadipscing eum ea tincidunt et nulla.  
  
-----  
Sadipscing dolores in consequat ipsum rebum consequat wisi sed. Dolor mazim  
rebum ea nonumy diam aliquyam at erat sea. Eirmod no sed ea accusam sit et et  
sadipscing nulla quis dolore sit in no lorem velit.
```

Ln 1, Col 1 725 characters 100% Windows (CRLF) UTF-8

abs received_by_client.txt 3/25/2024 9:27 AM

Computer Networks time client 3/25/2024 4:38 AM

Type

- PNG File
- Text Document
- Text Document
- Text Document

^ The second readme file to download

6. Program Explanation

- How the server transfers the complete file to the client.

- The server receives the filename with the FILENAME_PDU, it then scans the received buffer and splits it into the pdu type and the pdu data using sscanf, but we don't want to over scan and not leave space for the termination character, so we limit the sscanf to only scan 1 less character in the buffer to allow for the termination character to be added to the string.

```

13
14 #define BUFLEN 100 /* buffer length */
15 #define DATA_PDU 'D'
16 #define FINAL_PDU 'F'
17 #define ERROR_PDU 'E'
18 #define FILENAME_PDU 'C'
19
20 struct pdu {
21     char type;
22     char data[BUFLEN];
23 };
24
25 void set_pdu_data(struct pdu *pdu, const char *data, size_t len) {
26     memcpy(pdu->data, data, len); // Directly copy the specified length of data
27 }
28
29 int transferFile(int socketd, const char* buf, struct sockaddr *addr, socklen_t addr_len) {
30     char fileContents[BUFLEN]; // Adjusted size to match PDU data field
31     struct pdu result_pdu;
32     int fileLength, bytesRead = 0, bytesToRead;
33
34     // Extract filename from received PDU
35     struct pdu received_pdu;
36     sscanf(buf, "%c%99s", &received_pdu.type, received_pdu.data); // Ensure null-termination
37     if (received_pdu.type != FILENAME_PDU) {
38         // Send error message to client
39         char *error = "Error: Expected filename PDU";
40         result_pdu.type = ERROR_PDU;
41         set_pdu_data(&result_pdu, error, sizeof(error));
42         if (sendto(socketd, &result_pdu, sizeof(char) + sizeof(error), 0, addr, addr_len) == -1) {
43             fprintf(stderr, "sendto failed (when trying to send an error message): - \"%s\"\n", error);
44         }
45
46     }
47     return -1;
48 }
```

- If the pdu type is the FILENAME_PDU we open the file with read permissions

```

49     // Open the file
50     FILE *file = fopen(received_pdu.data, "r");
51     if (file == NULL) {
52         // Send error message to client
53         char *error = "Error: File not found";
54         result_pdu.type = ERROR_PDU;
55         set_pdu_data(&result_pdu, error, sizeof(error));
56         if (sendto(socketd, &result_pdu, sizeof(char) + sizeof(error), 0, addr, addr_len) == -1) {
57             fprintf(stderr, "sendto failed (when trying to send an error message): - \"%s\"\n", error);
58         }
59
60     }
61     return -1;
62 }
63 // Determine file size
```

- Determine the file size (this is because we need to identify when we've read the end of file, so the pdu type is set to FINAL_PDU, identifying the final pdu sent to the client)

```

63     // Determine file size
64     struct stat st;
65     if (stat(received_pdu.data, &st) == 0) {
66         fileLength = st.st_size;
67     } else {
68         fclose(file);
69
70         // Send error message to client
71         char *error = "Error: Cannot determine file size";
72         result_pdu.type = ERROR_PDU;
73         set_pdu_data(&result_pdu, error, sizeof(error));
74         if (sendto(socketd, &result_pdu, sizeof(char) + sizeof(error), 0, addr, addr_len) == -1) {
75             fprintf(stderr, "sendto failed (when trying to send an error message): - \"%s\"\n", error);
76         }
77
78         return -1;
79     }
80

```

- Read the fileContents of the file and based on whether we're about to read all the bytes from the file set the pdu type to DATA_PDU or FINAL_PDU (this is if the chunk of data we're reading from the file is the last chunk of data in the file)

```

81     // Read and send file contents in chunks
82     while ((bytesToRead = fread(fileContents, sizeof(char), BUflen, file)) > 0) {
83         result_pdu.type = (fileLength <= bytesRead + bytesToRead) ? FINAL_PDU : DATA_PDU;
84         set_pdu_data(&result_pdu, fileContents, bytesToRead);
85
86         if (sendto(socketd, &result_pdu, sizeof(char) + bytesToRead, 0, addr, addr_len) == -1) {
87             fprintf(stderr, "sendto failed\n");
88             break;
89         }
90         bytesRead += bytesToRead;
91     }
92

```

- If we encounter errors we send those errors to the client using the pdu type of ERROR_PDU, and the error in the data of the PDU

> We take into account the size of the data to be sent for errors to ensure the right amount of data is sent for the error message

```

// Send error message to client
char *error = "Error: Expected filename PDU";
result_pdu.type = ERROR_PDU;
set_pdu_data(&result_pdu, error, sizeof(error));
if (sendto(socketd, &result_pdu, sizeof(char) + sizeof(error), 0, addr, addr_len) == -1) {
    fprintf(stderr, "sendto failed (when trying to send an error message): - \"%s\"\n", error);
}

return -1;

```

```

52     // Send error message to client
53     char *error = "Error: File not found";
54     result_pdu.type = ERROR_PDU;
55     set_pdu_data(&result_pdu, error, sizeof(error));
56     if (sendto(socketd, &result_pdu, sizeof(char) + sizeof(error), 0, addr, addr_len) == -1) {
57         fprintf(stderr, "sendto failed (when trying to send an error message): - \"%s\"\n", error);
58     }
59
60     return -1;

```

```

    // Send error message to client
    char *error = "Error: Cannot determine file size";
    result_pdu.type = ERROR_PDU;
    set_pdu_data(&result_pdu, error, sizeof(error));
    if (sendto(socketd, &result_pdu, sizeof(char) + sizeof(error), 0, addr, addr_len) == -1) {
        fprintf(stderr, "sendto failed (when trying to send an error message): - \"%s\"\n", error);
    }

    return -1;
}

```

- How the client receives the complete file and handles the error message
 - The client sends the filename entered through stdin, sets the pdu type to FILENAME_PDU and sends the filename to the server

```

15  #define BUflen 100 /* buffer length */
16
17  #define DATA_PDU 'D'
18  #define FINAL_PDU 'F'
19  #define ERROR_PDU 'E'
20  #define FILENAME_PDU 'C'
21
22  struct pdu {
23      char type;
24      char data[BUflen];
25  };
26
27  void set_pdu_data(struct pdu *pdu, const char *data) {
28      strncpy(pdu->data, data, BUflen - 1);
29      pdu->data[BUflen - 1] = '\0';
30  }
31
32  int receiveFile(int socketd, struct sockaddr_in *sin) {
33      struct pdu received_pdu, send_pdu;
34      char filename[BUflen];
35      FILE *file;
36
37      // Get the filename from the user
38      printf("Enter filename: ");
39      fgets(filename, BUflen, stdin);
40      filename[strcspn(filename, "\n")] = '\0'; // Remove newline
41
42      // Prepare FILENAME_PDU to request the file
43      send_pdu.type = FILENAME_PDU;
44      set_pdu_data(&send_pdu, filename);
45
46      // Send filename PDU to server
47      if (write(socketd, &send_pdu, sizeof(send_pdu)) < 0) {
48          fprintf(stderr, "sendto failed (couldn't send filename PDU to server)\n");
49          return -1;
50      }
51

```

- Then the client opens a new file called `received_by_client.txt` with write permissions (specifically we want the file to be overwritten/cleared or created if it doesn't exist), we'll save the results of download into the file

```

// Open file for writing
file = fopen("received_by_client.txt", "w");
if (!file) {
    fprintf(stderr, "Cannot open file to write");
    return -1;
}

```

- We'll then read the data from the server, and break it up into the pdu type and data, from there if the pdu type is DATA_PDU or FINAL_PDU we write the data into received_by_client.txt, if the pdu is the ERROR_PDU we print the error to the terminal

```

// Receive file contents
while (1) {
    ssize_t n = read(socketd, &received_pdu, sizeof(received_pdu));
    if (n < 0) {
        fprintf(stderr, "read failed");
        fclose(file);
        return -1;
    }

    // Check PDU type
    if (received_pdu.type == ERROR_PDU) {
        fprintf(stderr, "Error from server: %s\n", received_pdu.data);
        fclose(file);
        return -1;
    } else if (received_pdu.type == DATA_PDU || received_pdu.type == FINAL_PDU) {
        fwrite(received_pdu.data, sizeof(char), n - sizeof(received_pdu.type), file);
        if (received_pdu.type == FINAL_PDU) {
            break; // last packet
        }
    } else {
        fprintf(stderr, "Unknown PDU type received\n");
        fclose(file);
        return -1;
    }
}

printf("File transfer complete.\n");
fclose(file);
return 0;

```