SOFE 3850: Computer Networks

Project Report

Group: Project Groups 4

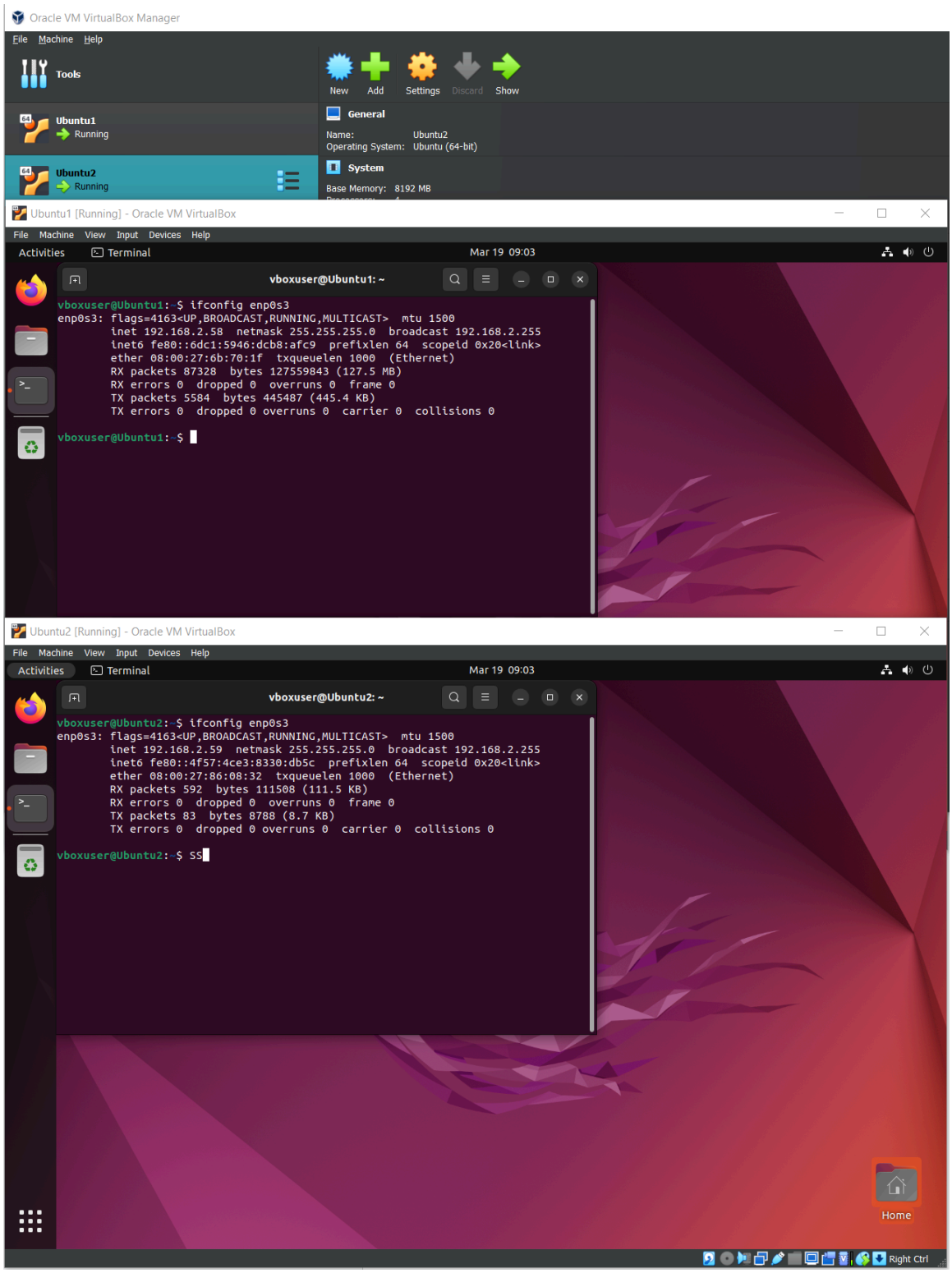
Members:

Name	Student ID
Jason Stuckless	100248154
Okiki Ojo	100790236

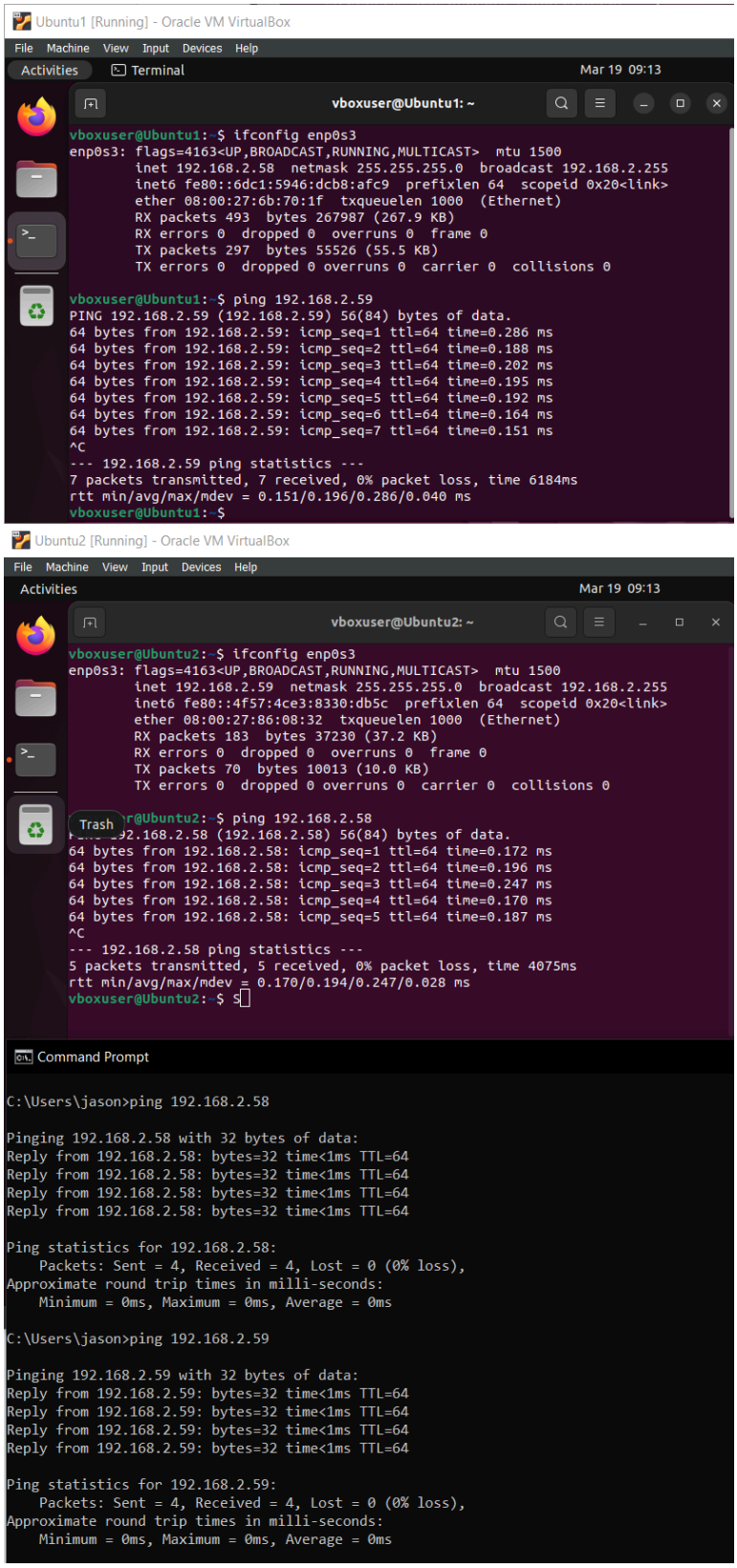
Table of Contents

Part 1: Setup of VMs and the Virtual Network Environment

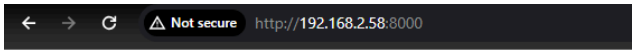
1. Virtual Machines



2. Successful Pings

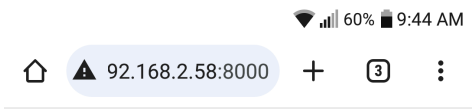


3. Web Server



Directory listing for /

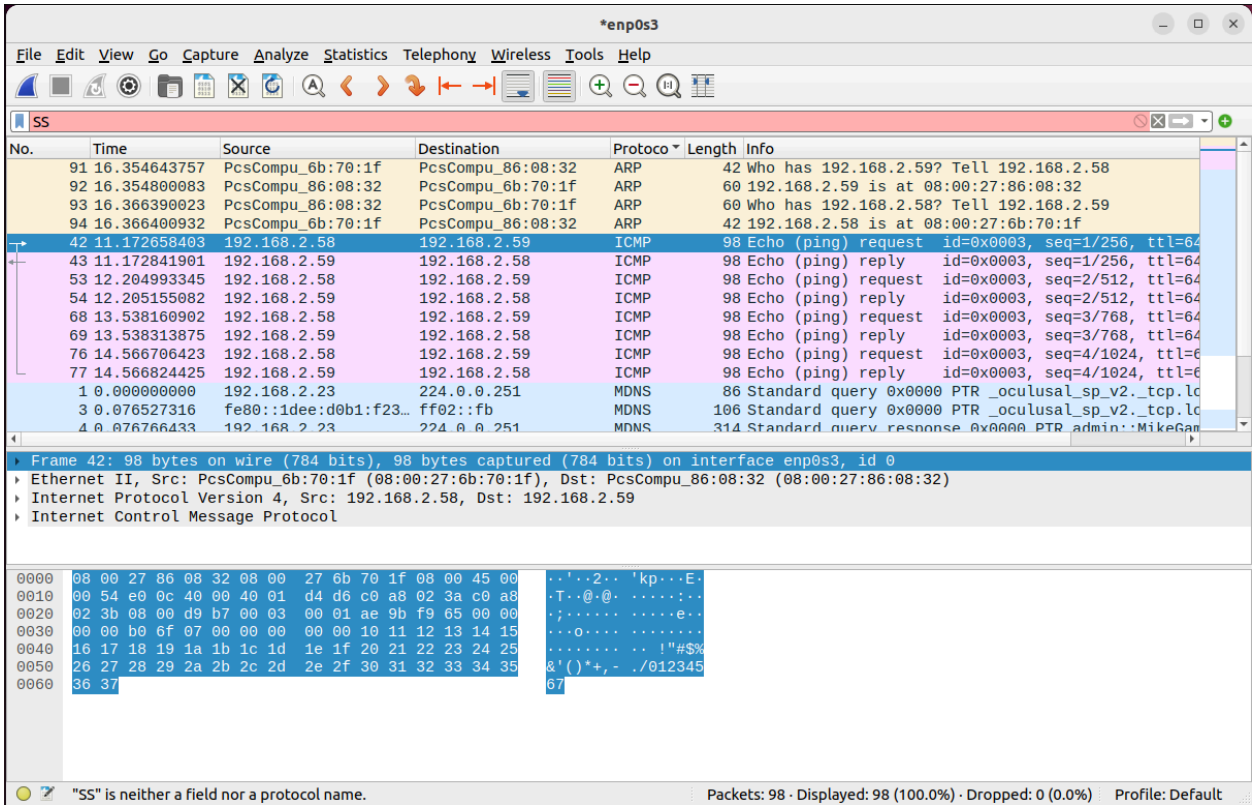
- .bash_history
- .bash_logout
- .bashrc
- .cache/
- .config/
- .local/
- .profile
- .sudo_as_admin_successful
- Desktop/
- Documents/
- Downloads/
- Music/
- Pictures/
- Public/
- snap/
- Templates/
- Videos/



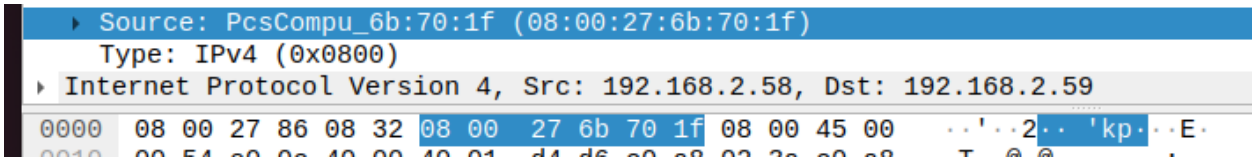
Directory listing for /

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

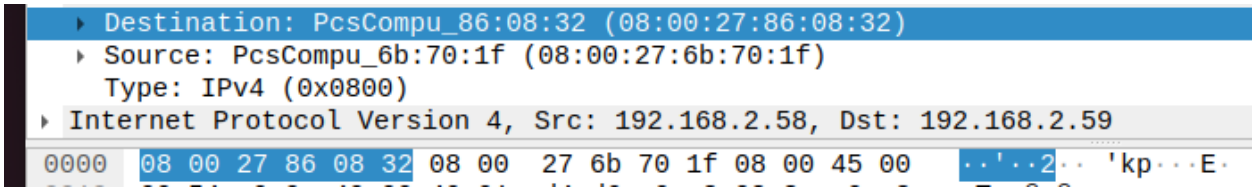
4. Wireshark Captured Data



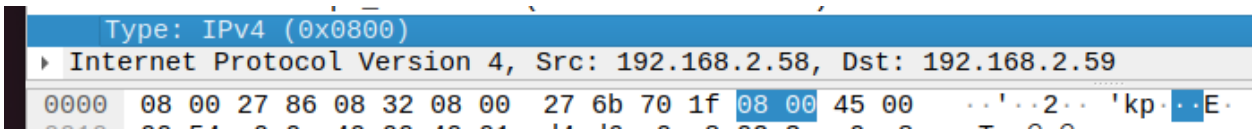
Source Ethernet Address: 08:00:27:6b:70:1f



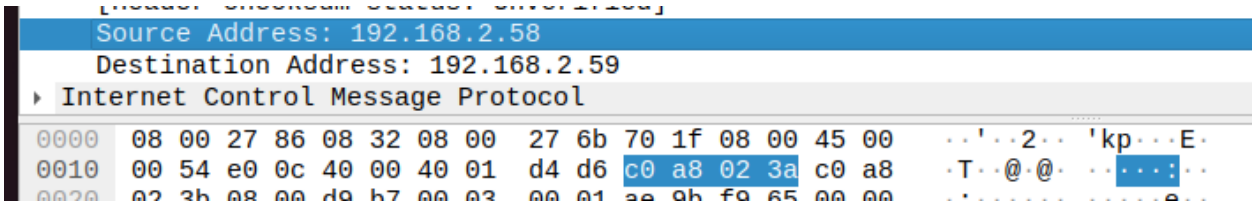
Destination Ethernet Address: 08:00:27:86:08:32



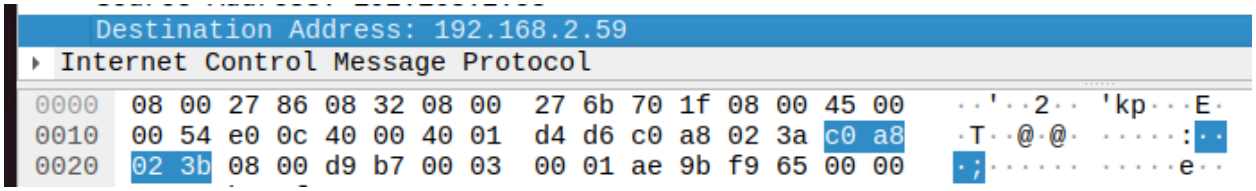
Ethernet Type Field: 08:00



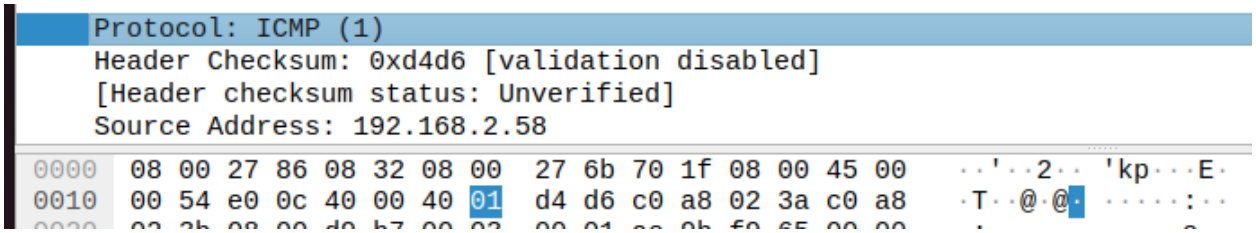
Source IP Address: c0:a8:02:3a (192.168.2.58)



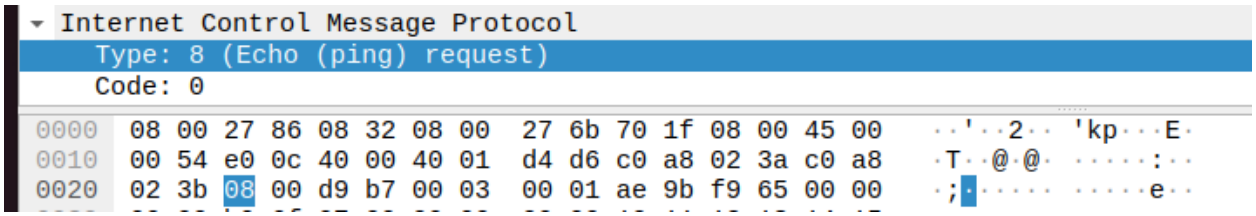
Destination IP Address: c0:a8:02:3b (192.168.2.59)



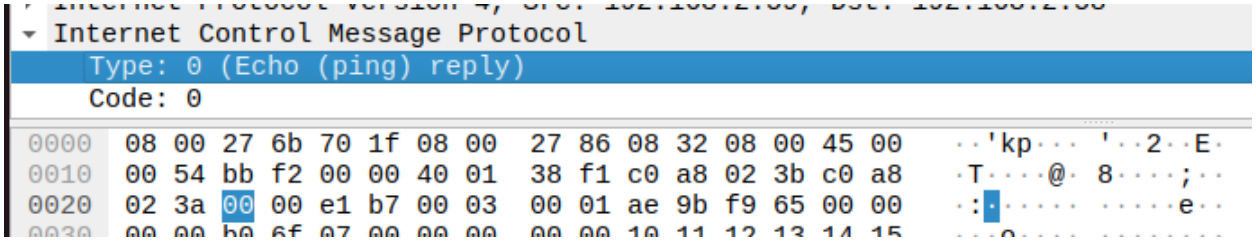
IP Protocol Field: 01 (ICMP)



ICMP Type Field, Ping Request: 08

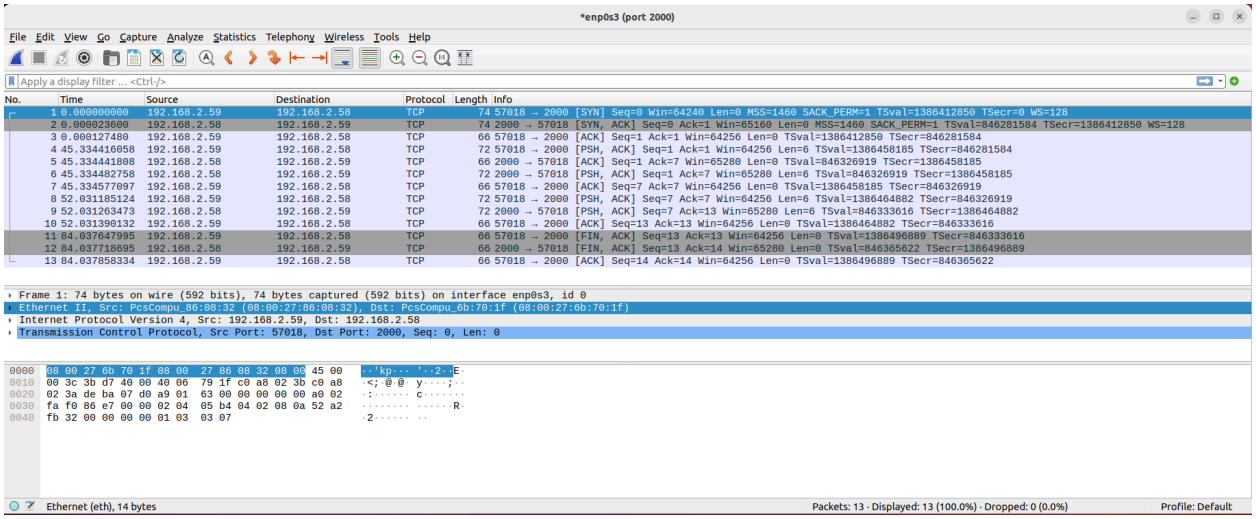


ICMP Type Field, Ping Response: 00



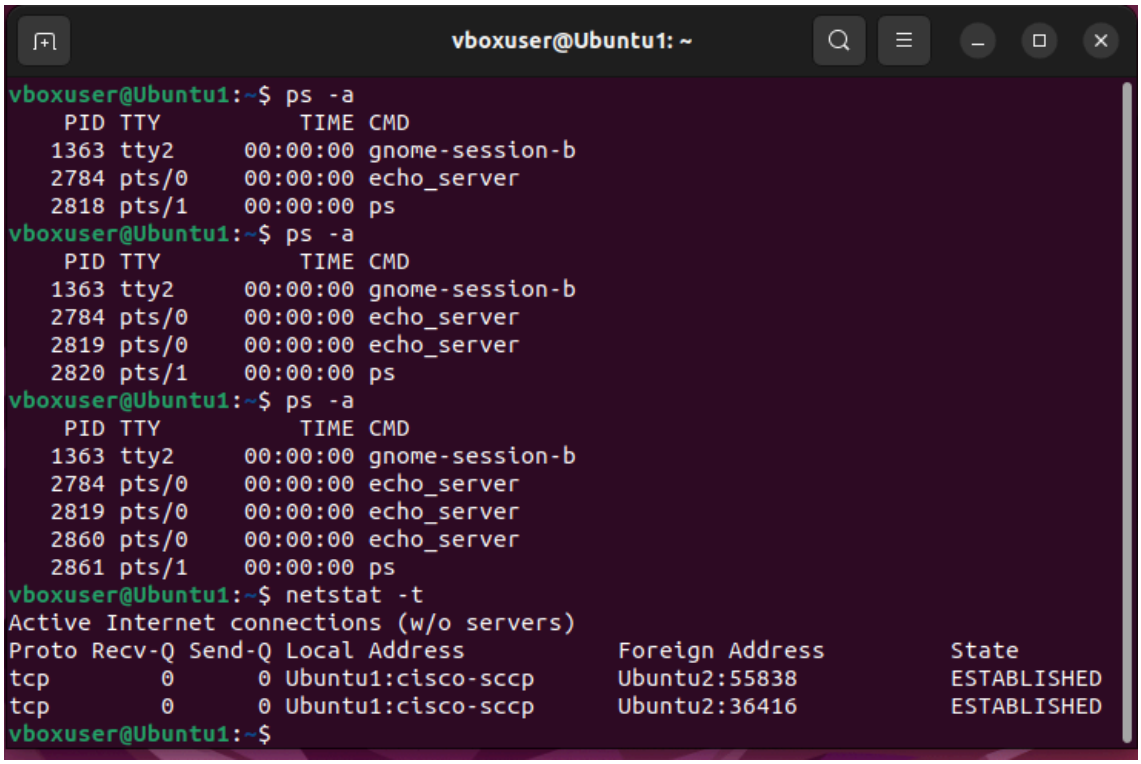
Part 2: TCP Server

1. Wireshark Captured Data

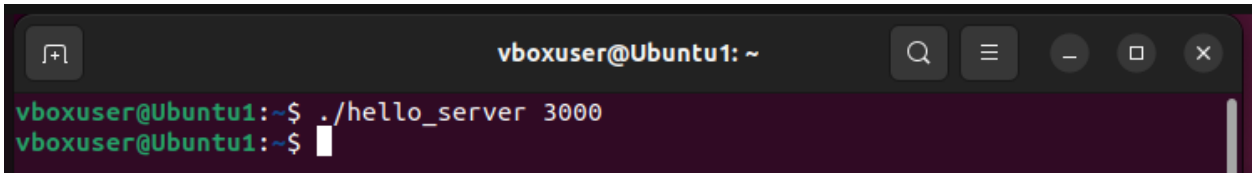


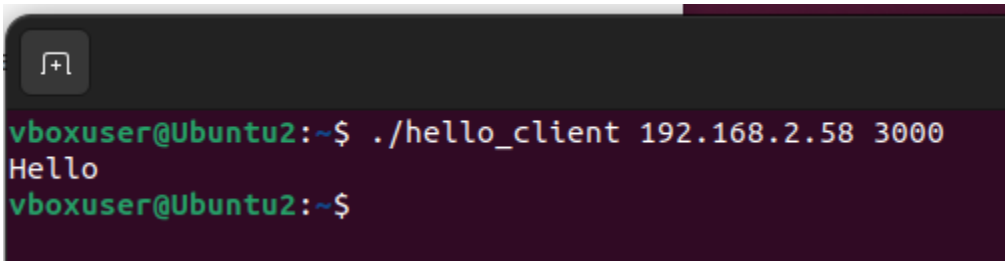
TCP Connection Packets: Packets No. 1-3
TCP Termination Packets: Packets No. 11-13

2. Child Processes



3. Part III Programs





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:

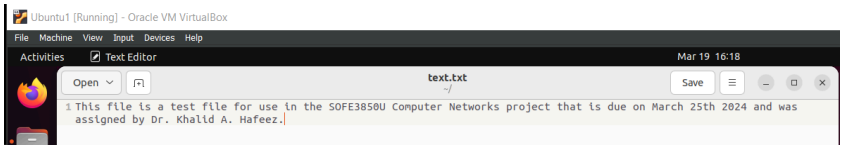
```
int sendHello (int sd) {
      char *bp, buf[BUFLEN];

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

      return(0);
}
```

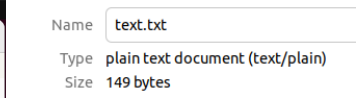
Line 61 to 73 in echo_client.c changed in hello_client.c:

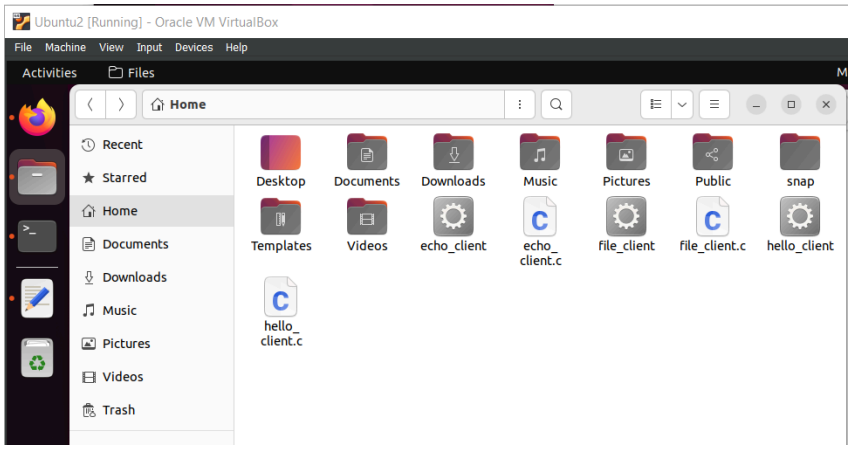
```
while(n=read(sd, sbuf, BUFLEN)){
      printf("%s\n", sbuf);
      close(sd);
      break;
}
```

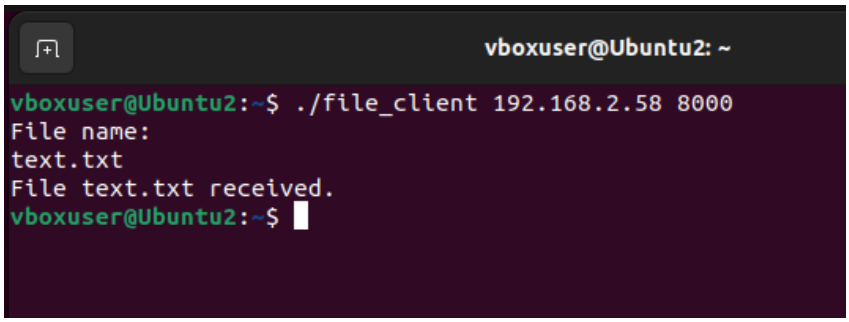
Part 3: File Download Application Based on TCP

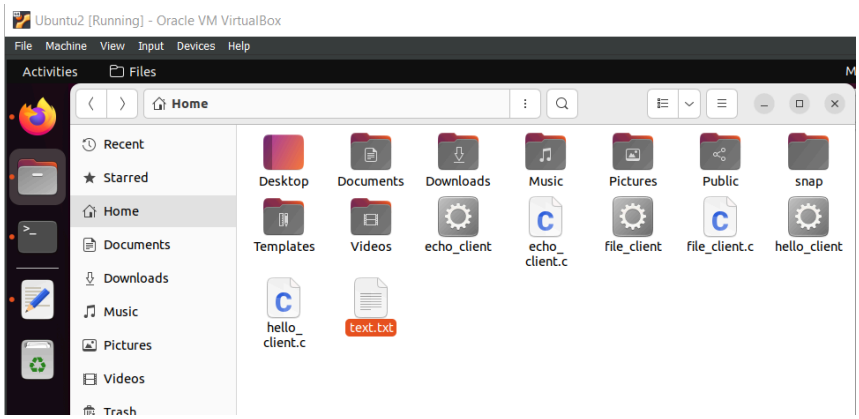
1. File Download

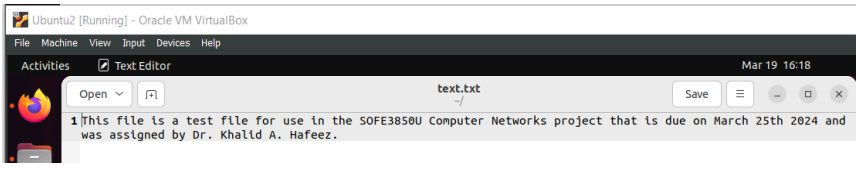




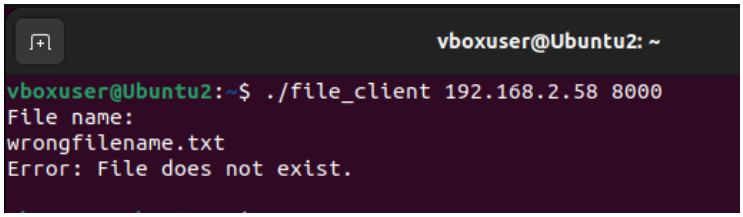








2. Error Message



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, "f%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);
}
```

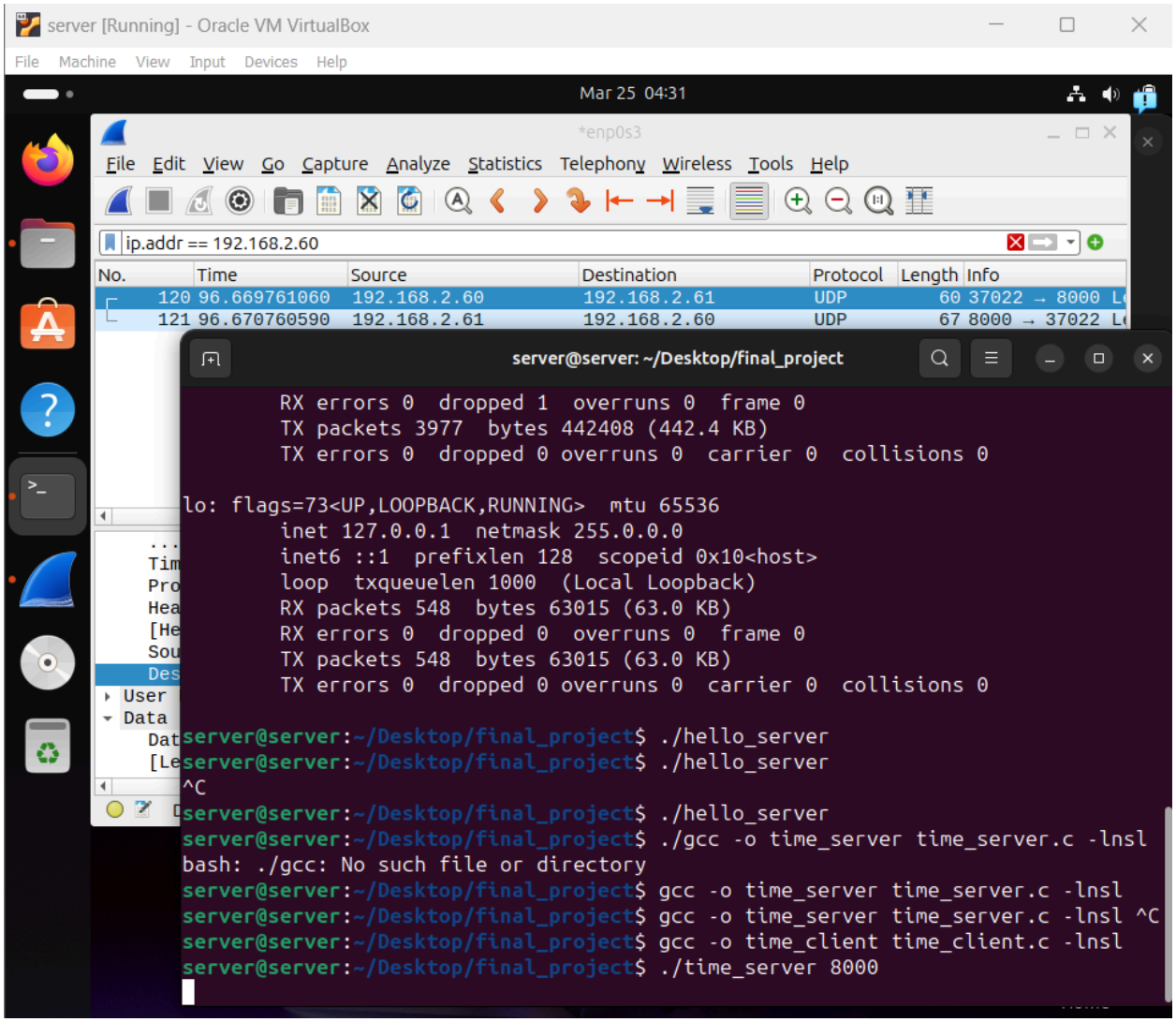
The server stops transmitting and terminates connection after either sending an error message or after it has sent the file in 100 byte increments and reached the end of the file (i < fileLength in for loop).

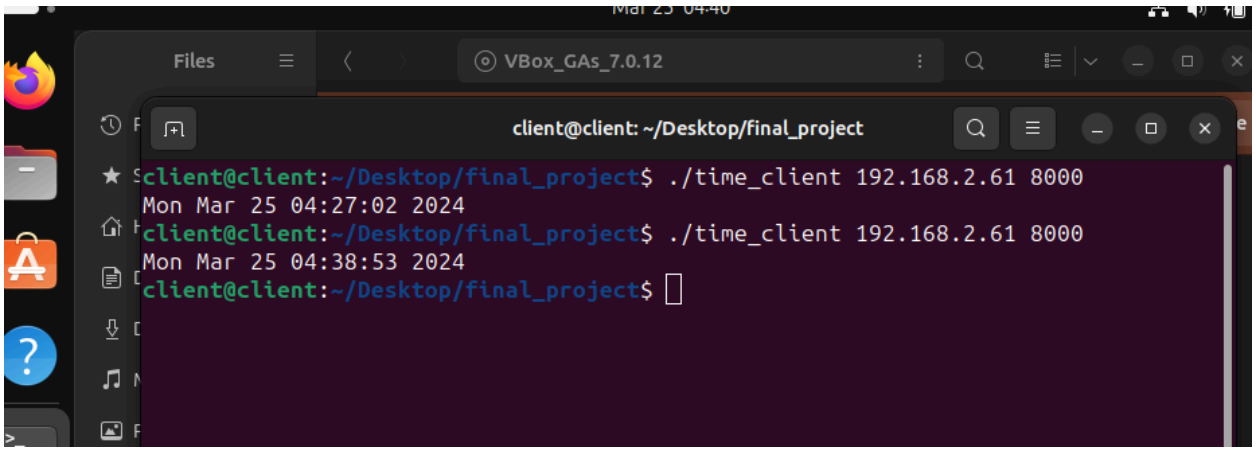
The client stops receiving once it has received an error message or the requested file.

Part 4: UDP Server Implementation

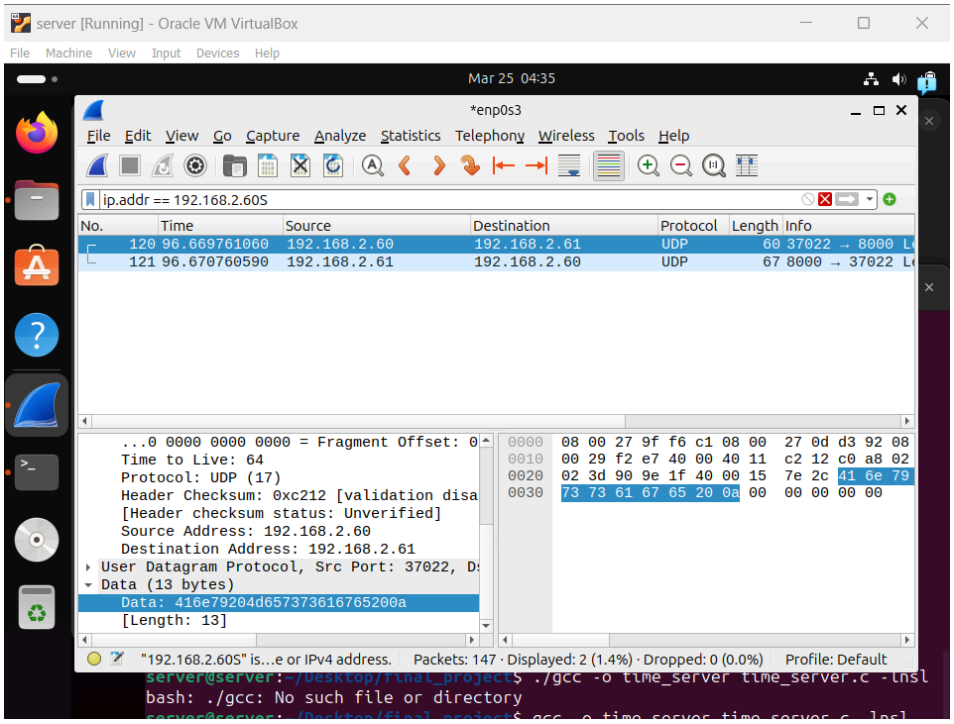
1. Steps 1 to 6

Compiling and Running the time server and client

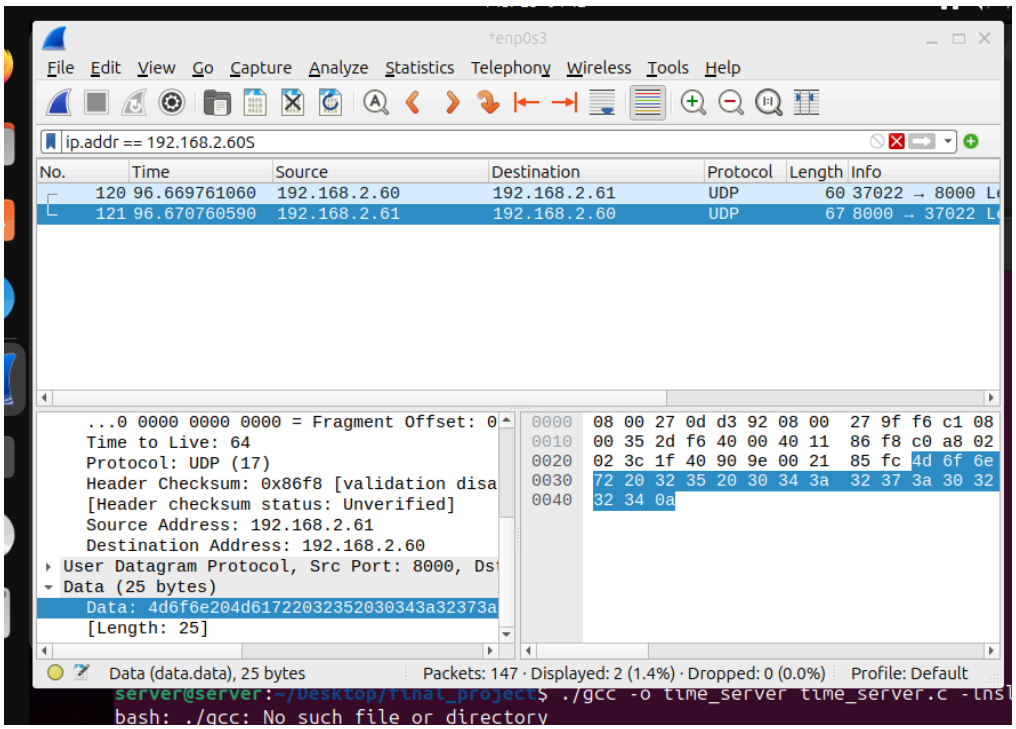




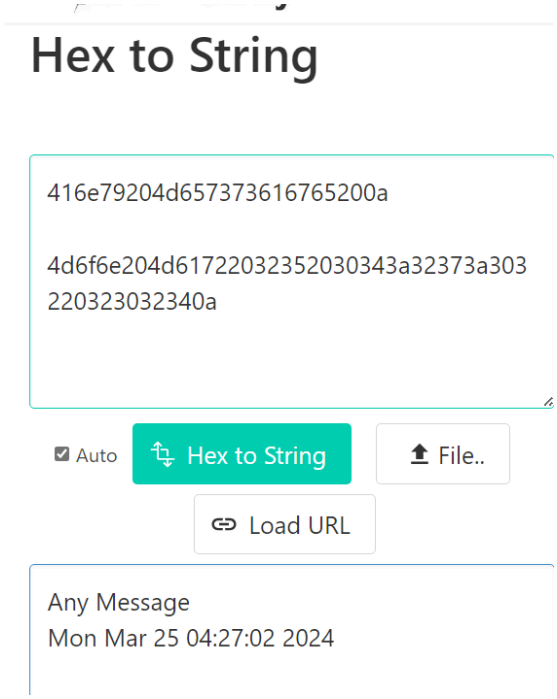
Using wireshark to view the message sent from the client `192.168.2.60` to the server `192.168.2.61`



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



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

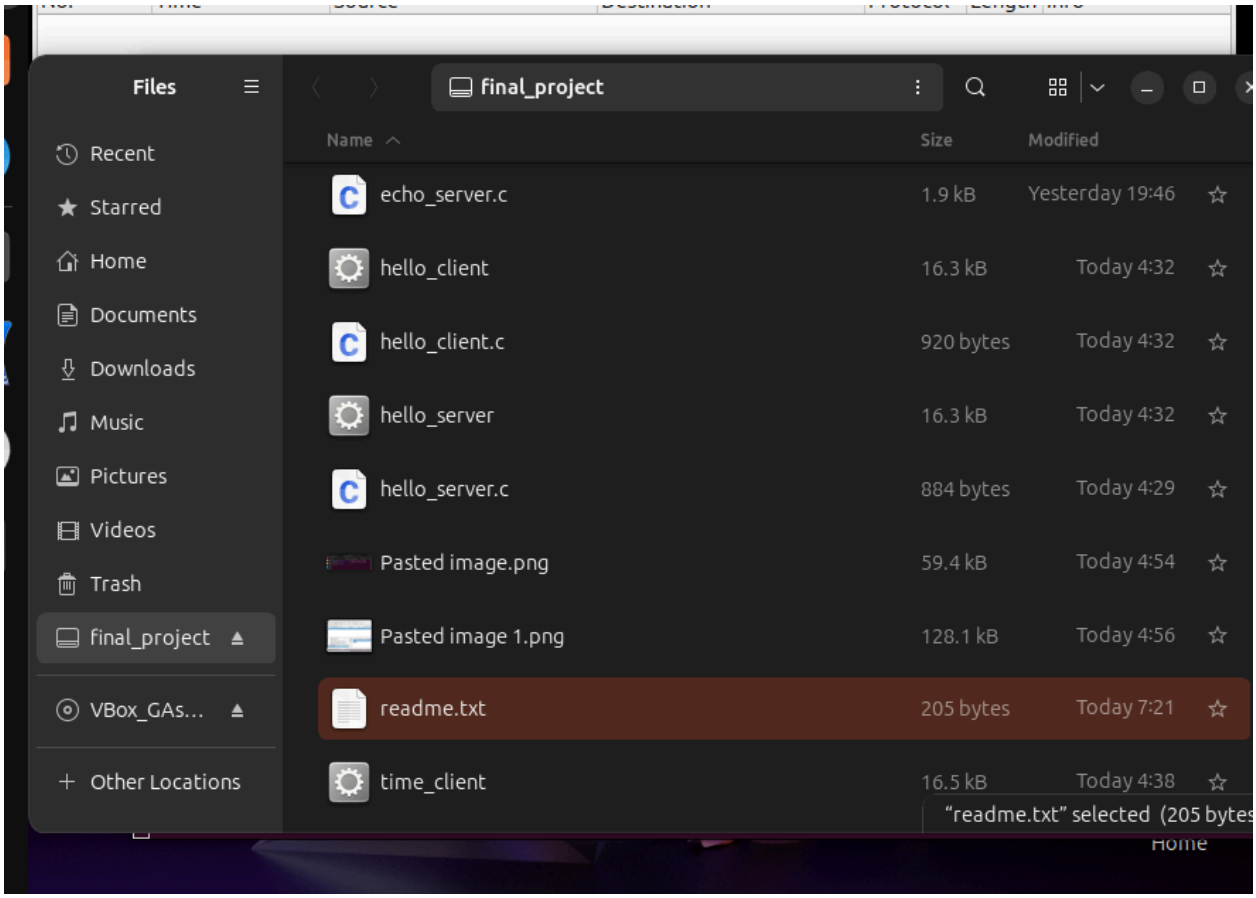
UDP is the more appropriate protocol, as the up to date information is more important than verifying that the information has been received, if a packet gets dropped in UDP there will be another packet sent that will be more accurate to the current date, but if we used TCP the request would need to be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

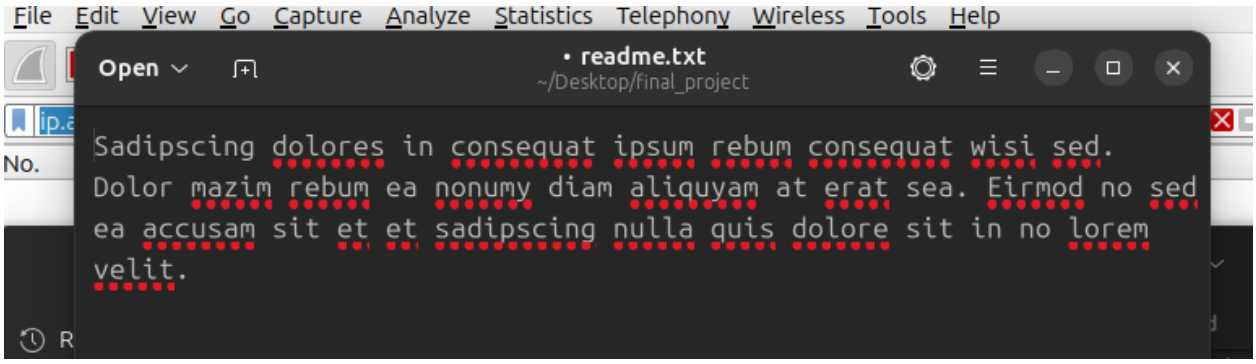
4. Concurrent vs. Non-Concurrent

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)





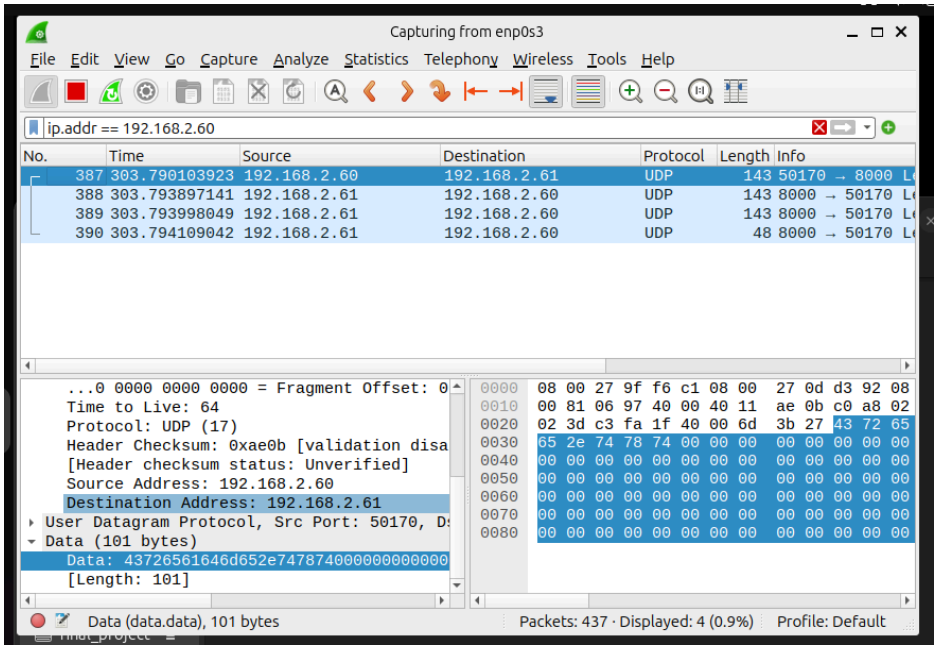
Startup the udp server with port 8000

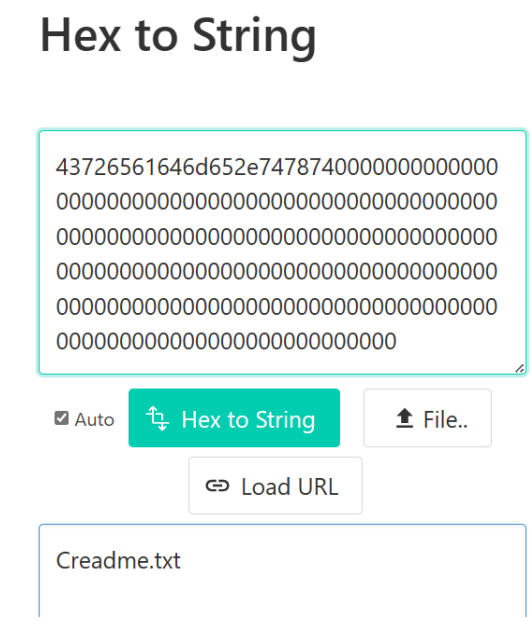
```
server@server:~/Desktop/final_project$ gcc -o udp_server udp_server.c -lnsl && g
cc -o udp_client udp_client.c -lnsl
server@server:~/Desktop/final_project$ ./udp_server 8000
- Error transferring file
- Error transferring file
^C
server@server:~/Desktop/final_project$ gcc -o udp_server udp_server.c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

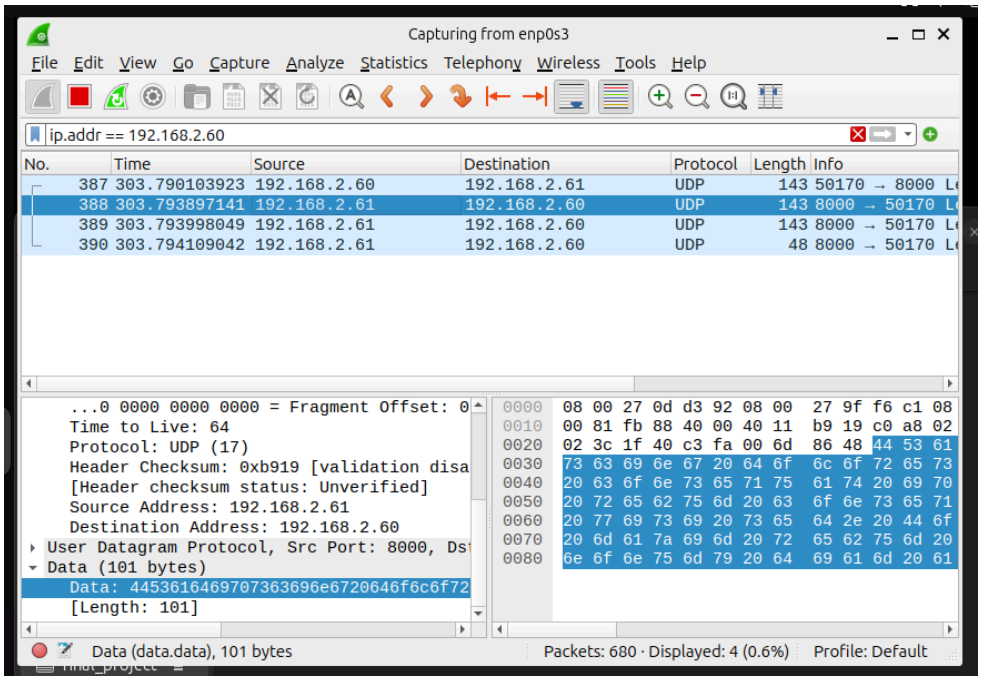
```
client@client:~/Desktop/final_project$ ./udp_client 192.168.2.61 8000
Enter filename: readme.txt
File transfer complete.
Enter filename:
```

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





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



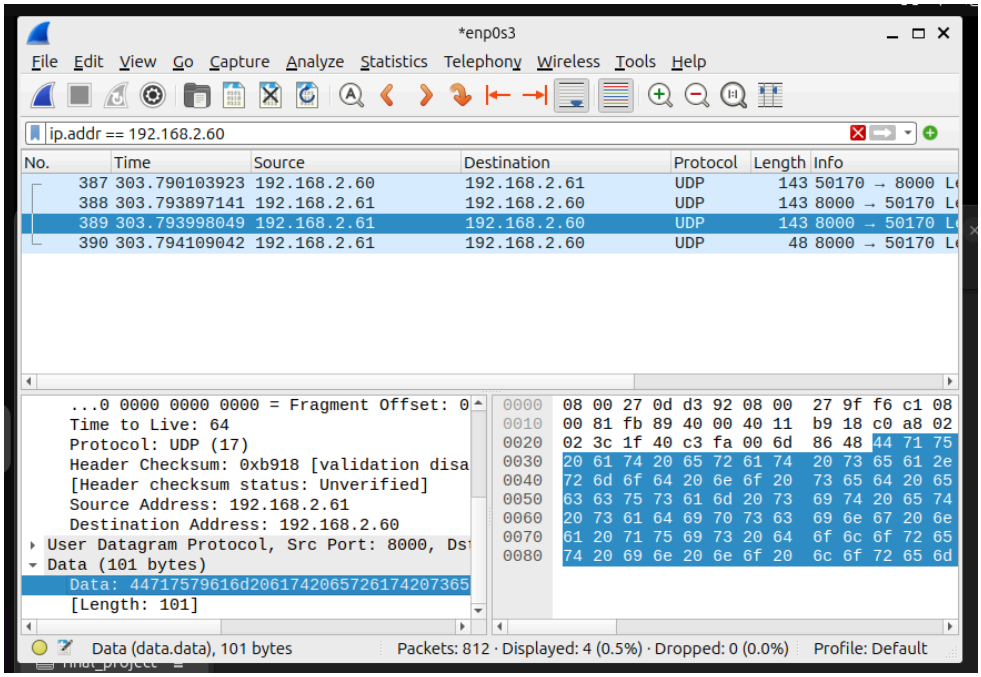
Hex to String

4453616469707363696e6720646f6c6f72657320696e20636f6e7
3657175617420697073756d20726562756d20636f6e736571756
174207769736920736564e20446f6c6f72206d617a696d207265
62756d206561206e6f6e756d79206469616d20616c69

Auto | Hex to String | File.. | Load URL

DSadipscing dolores in consequat ipsum rebum consequat wisi sed. Dolor mazim rebum ea nonumy diam ali

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



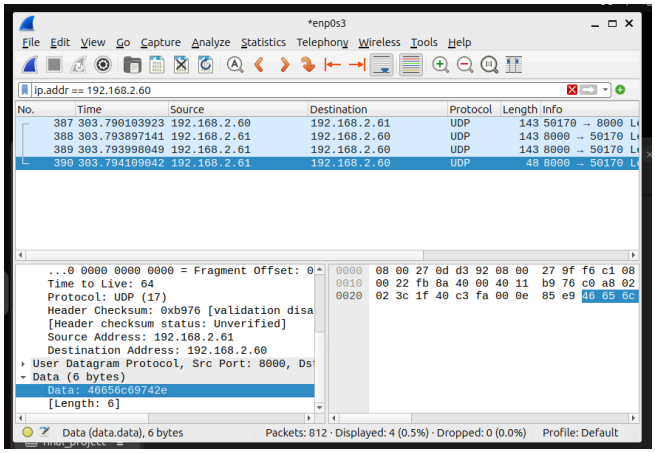
Hex to String

44717579616d20617420657261742073656 12e204569726d6f64206e6f207365642065612061636375736 16d2073697420657420657420736164697073636 96e67206e756c6c61207175697320646f6c6f726520736974206 96e206e6f206c6f72656d2076

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



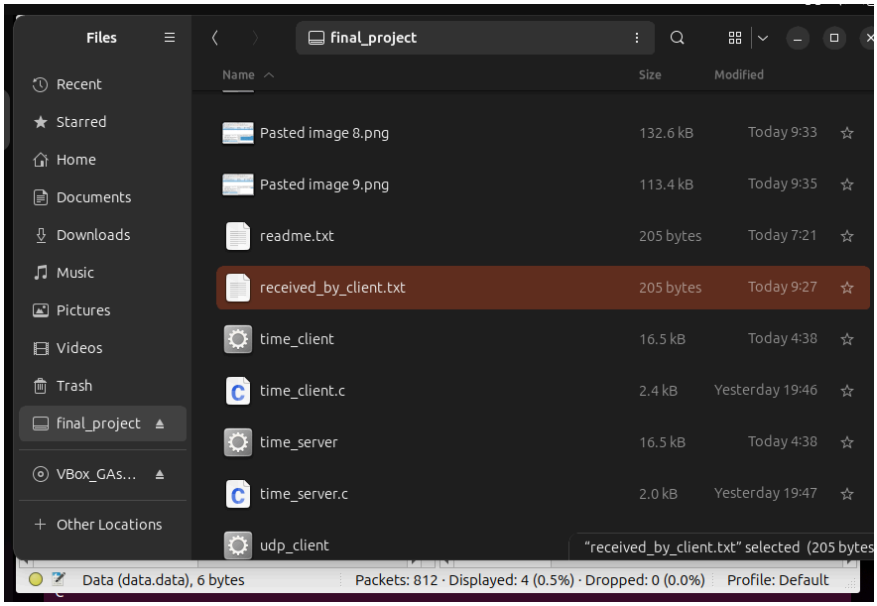
Hex to String

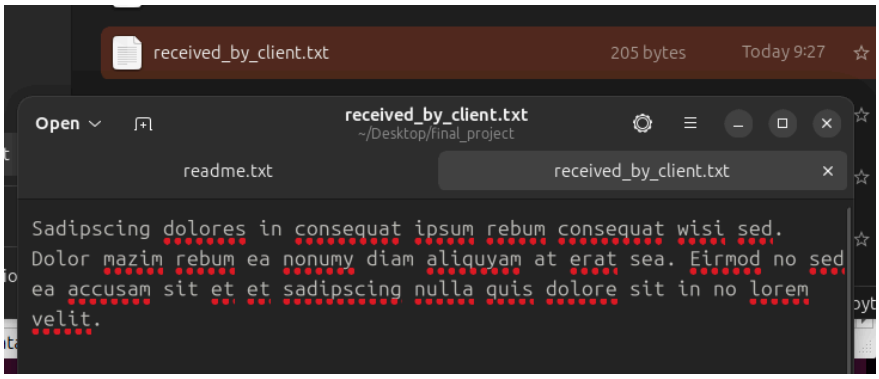
46656c69742e

Auto | Hex to String | File.. | Load URL

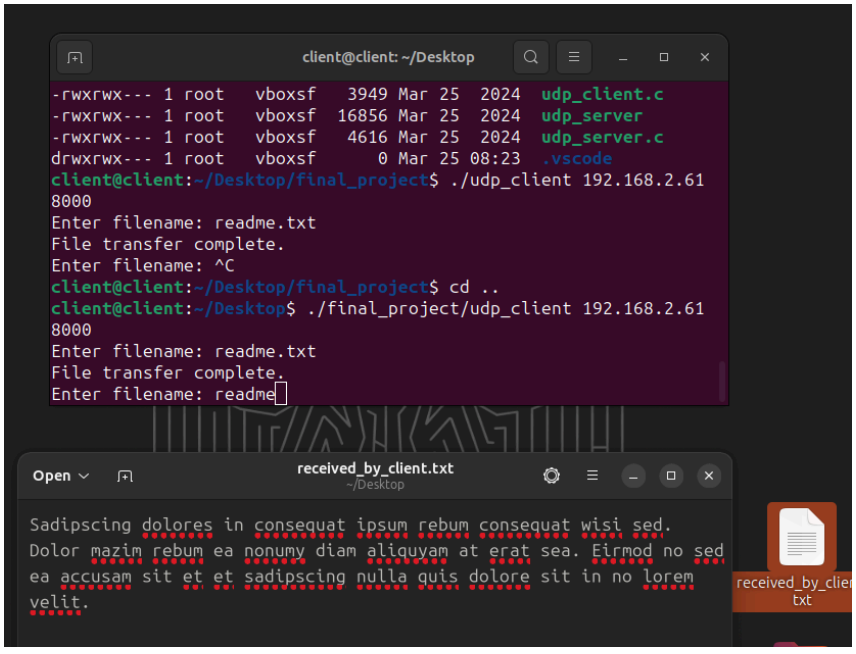
Felit.

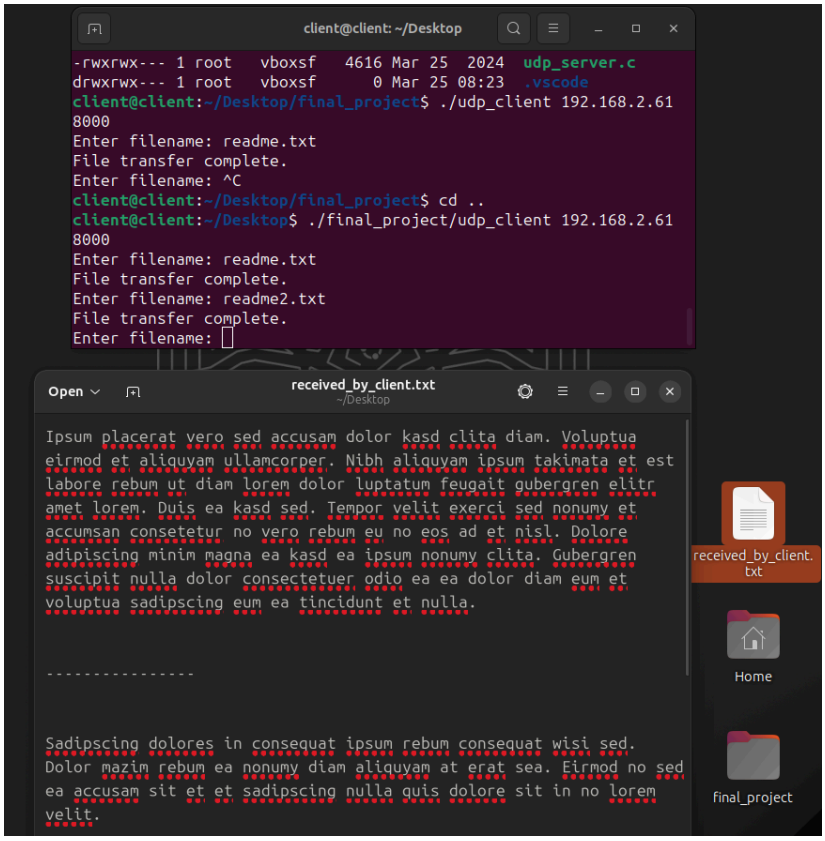
Here is the new file created from the server to client download it's called recieved_by_client.txt

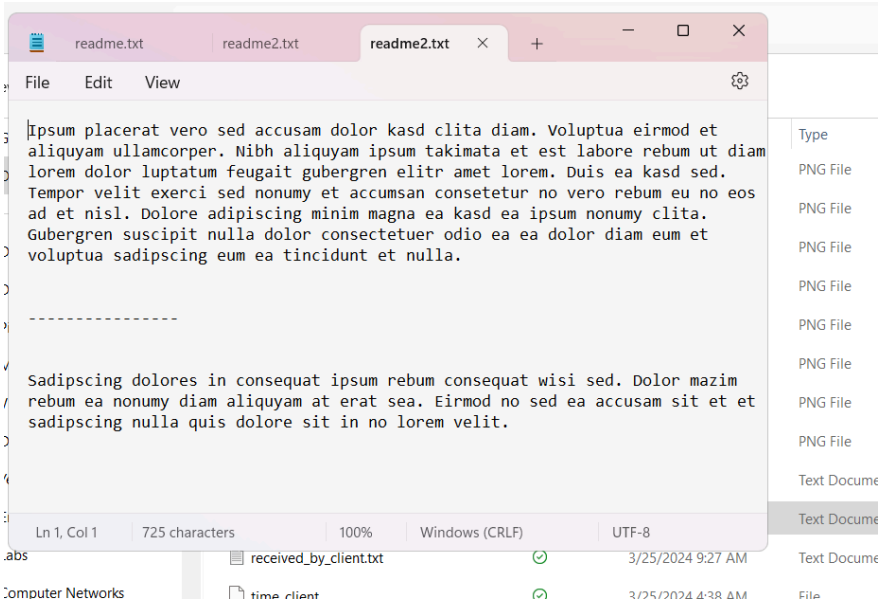




Downloading multiple separate files





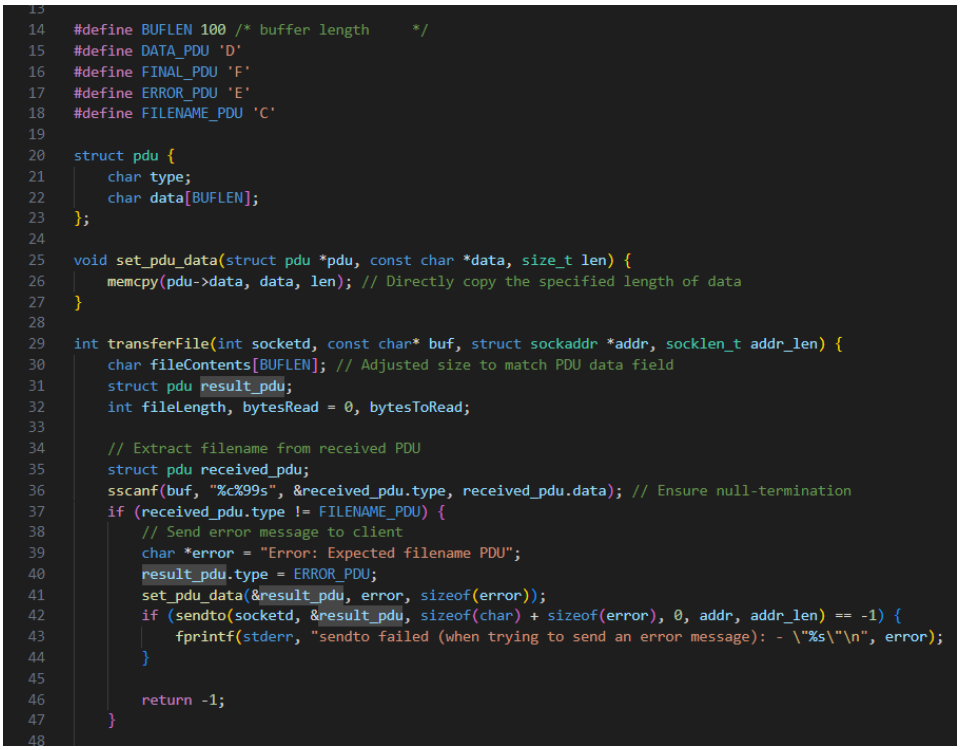


^ The second readme file to download

6. Program Explanation

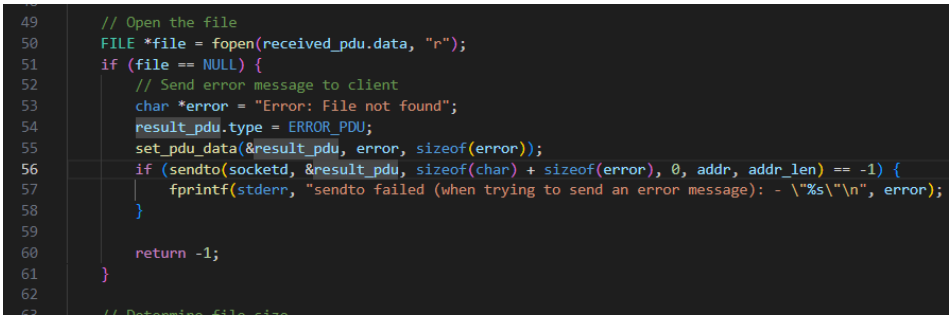
- How the server transfers the complete file to the client.

- 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.



```
#define BUFLEN 100 /* buffer length     */
#define DATA_PDU 'D'
#define FINAL_PDU 'F'
#define ERROR_PDU 'E'
#define FILENAME_PDU 'C'

struct pdu {
    char type;
    char data[BUFLEN];
};

void set_pdu_data(struct pdu *pdu, const char *data, size_t len) {
    memcpy(pdu->data, data, len); // Directly copy the specified length of data
}

int transferFile(int socketd, const char* buf, struct sockaddr *addr, socklen_t addr_len) {
    char fileContents[BUFLEN]; // Adjusted size to match PDU data field
    struct pdu result_pdu;
    int fileLength, bytesRead = 0, bytesToRead;

    // Extract filename from received PDU
    struct pdu received_pdu;
    sscanf(buf, "%c%99s", &received_pdu.type, received_pdu.data); // Ensure null-termination
    if (received_pdu.type != FILENAME_PDU) {
        // 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;
    }
```

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



```
    // Open the file
    FILE *file = fopen(received_pdu.data, "r");
    if (file == NULL) {
        // Send error message to client
        char *error = "Error: File not found";
        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;
    }
```

- 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)

```
    // Determine file size
    struct stat st;
    if (stat(received_pdu.data, &st) == 0) {
        fileLength = st.st_size;
    } else {
        fclose(file);

        // 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;
    }
```

- Read the fileContents of the file and based on whether the 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)

```
    // Read and send file contents in chunks
    while ((bytesToRead = fread(fileContents, sizeof(char), BUFLEN, file)) > 0) {
        result_pdu.type = (fileLength <= bytesRead + bytesToRead) ? FINAL_PDU : DATA_PDU;
        set_pdu_data(&result_pdu, fileContents, bytesToRead);

        if (sendto(socketd, &result_pdu, sizeof(char) + bytesToRead, 0, addr, addr_len) == -1) {
            fprintf(stderr, "sendto failed\n");
            break;
        }
        bytesRead += bytesToRead;
    }
```

socklen_t addr_len

- 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;
```

```
        // Send error message to client
        char *error = "Error: File not found";
        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;
```

```
// 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;
```