



CS2099

# **Computer Network Lab**

## **Assignment 01**

**File Transfer using TCP/UDP.**

**Submitted by:** Abdul Ahad

**Roll number:** 23i-2014

**Date:** 10-04-2025

## Table of Contents

TCP File Transfer: .....	4
Server-Side TCP: .....	4
1. LOG_ACTIVITY ().....	4
2. MAKE_SERVER_DIR().....	4
3. SEND_LIST_TCP().....	5
4. SEND_FILE_TCP().....	5
5. RCV_FILE_TCP().....	5
6. HANDLE_TCP SOCK() .....	5
7. main ().....	6
Client-Side TCP Code Breakdown .....	6
1. MAKE_CLIENT_DIR() .....	6
2. MAKE_TCP_CONNECT().....	6
3. LIST_FILE_TCP().....	7
4. DOWNLOAD_TCP() .....	7
5. UPLOAD_TCP() .....	7
5. main ().....	7
Key Observations.....	8
1. Chunked Transfers:.....	8
2. Error Handling: .....	8
3. Threading: .....	8
4. Network Byte Order:.....	8
5. Testing Notes .....	8
UDP File Transfer:.....	9
Server-Side UDP Code Breakdown:.....	9
1. LOG_ACTIVITY ().....	9
2. MAKE_SERVER_DIR () .....	10
3. SEND_FL_UDP ().....	10
4. HANDLE_DOWNLOAD_UDP () .....	11
5. RECIEVE_FILES_UDP () .....	11
6. UDP_CONNECTION ().....	11
7. main ().....	12
Client-Side UDP Code Breakdown .....	12
1. MAKE_CLIENT_DIR() .....	12

2. MAKE_UDP_CONNECT () .....	12
3. LIST_FILE_UDP () .....	13
4. DOWNLOAD_UDP () .....	13
5. UPLOAD_UDP ().....	13
6. main ().....	14
Key Observations.....	14
1. Connectionless Nature: .....	14
2. Chunked Transfers: .....	14
3. No ACKs or Retries: .....	14
4. EOF Marker: .....	14

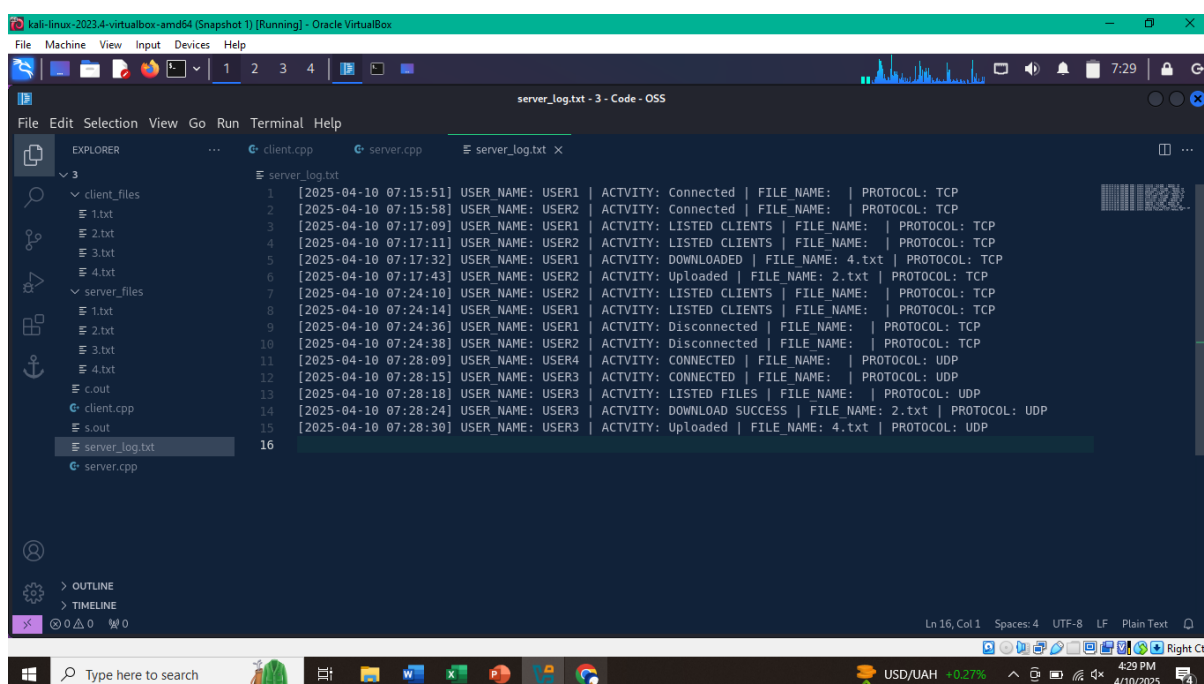
# TCP File Transfer:

## Server-Side TCP:

### 1. LOG\_ACTIVITY()

What it does:

1. Logs user actions (login, upload, download) with timestamps to server\_log.txt.
2. Tracks: Username, Action, Filename, Protocol (TCP/UDP).



The screenshot shows a Code editor window titled 'server\_log.txt - 3 - Code - OSS'. The Explorer sidebar on the left shows a project structure with 'client\_files' and 'server\_files' directories, each containing '1.txt', '2.txt', '3.txt', and '4.txt'. The main editor area displays the contents of 'server\_log.txt', which contains 16 log entries. Each entry is a single line of text with a timestamp, username, activity, filename, and protocol. The log entries are as follows:

Line	Timestamp	USER_NAME	ACTIVITY	FILE_NAME	PROTOCOL
1	[2025-04-10 07:15:51]	USER1	Connected		TCP
2	[2025-04-10 07:15:58]	USER2	Connected		TCP
3	[2025-04-10 07:17:09]	USER1	LISTED CLIENTS		TCP
4	[2025-04-10 07:17:11]	USER2	LISTED CLIENTS		TCP
5	[2025-04-10 07:17:32]	USER1	DOWNLOADED	4.txt	TCP
6	[2025-04-10 07:17:43]	USER2	Uploaded	2.txt	TCP
7	[2025-04-10 07:24:10]	USER2	LISTED CLIENTS		TCP
8	[2025-04-10 07:24:14]	USER1	LISTED CLIENTS		TCP
9	[2025-04-10 07:24:36]	USER1	Disconnected		TCP
10	[2025-04-10 07:24:38]	USER2	Disconnected		TCP
11	[2025-04-10 07:28:09]	USER4	CONNECTED		UDP
12	[2025-04-10 07:28:15]	USER3	CONNECTED		UDP
13	[2025-04-10 07:28:18]	USER3	LISTED FILES		UDP
14	[2025-04-10 07:28:24]	USER3	DOWNLOAD SUCCESS	2.txt	UDP
15	[2025-04-10 07:28:30]	USER3	Uploaded	4.txt	UDP
16					

Why it matters:

Help debug issues (e.g., failed uploads) and monitor server activity.

### 2. MAKE\_SERVER\_DIR()

What it does:

Create a **server\_files** directory if it doesn't exist.

```
// FUNCTION TO CREATE THE SERVER DIRECTORY,
// ACTUALLY FOR TESTING FILES, I HAVE ALSO USED THE
// SIMILAR APPROACH ON THE CLIENT SIDE AS WELL.
void MAKE_SERVER_DIR(){
    if(!filesystem::exists("server_files")){
        filesystem::create_directory("server_files");
        cout<<"\n MADE THE DIRECTORY :: "<<"server_files"<<endl;
    }
}
```

### **Why it matters:**

Ensures a dedicated folder for storing uploaded files.

### **3. SEND\_LIST\_TCP()**

#### **What it does:**

1. Lists all files in server\_files and sends them to the client in chunks.
2. Ends transmission with an EOF marker.

#### **Why chunking?**

Prevents large data bursts (avoids network congestion).

### **4. SEND\_FILE\_TCP()**

#### **What it does:**

1. Checks if the requested file exists.
2. Send the file size first (so the client knows how much data to expect).
3. Stream the file in fixed-size chunks (1024 bytes).

#### **Error Handling:**

Logs failures (e.g., missing file).

### **5. RCV\_FILE\_TCP()**

#### **What it does:**

1. Receives the file size (converts from network byte order).
2. Writes data to a new file in server\_files.
3. Sends an ACK ('1') to confirm success.

#### **Progress Tracking:**

Prints % downloaded (e.g., RECEIVING :: 75%).

### **6. HANDLE\_TCP\_SOCKET()**

#### **What it does:**

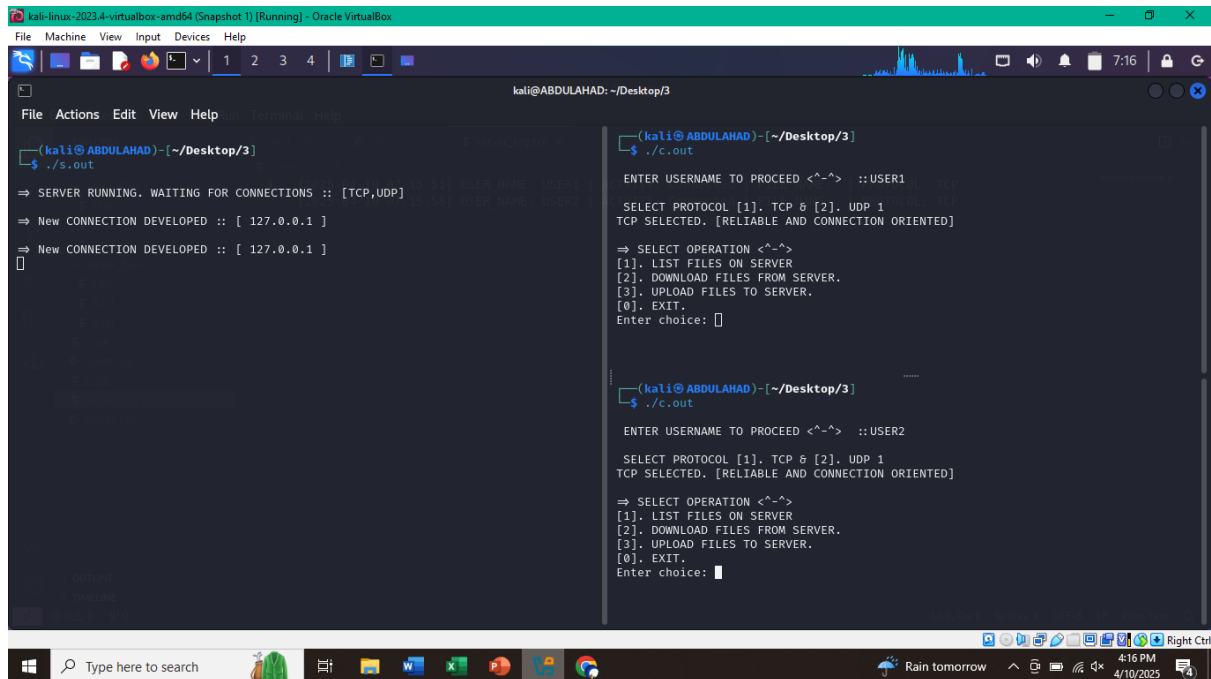
Main TCP handler for client commands:

## LIST:

1. Sends file list.
2. DOWNLOAD <filename>: Triggers SEND\_FILE\_TCP().
3. UPLOAD <filename>: Triggers RCV\_FILE\_TCP().

## Logs:

Connections/disconnections and file transfers.



```
kali@ABDULAHAD: ~/Desktop/3
$ ./s.out
⇒ SERVER RUNNING. WAITING FOR CONNECTIONS :: [TCP,UDP]
⇒ New CONNECTION DEVELOPED :: [ 127.0.0.1 ]
⇒ New CONNECTION DEVELOPED :: [ 127.0.0.1 ]

(kali@ABDULAHAD)-[~/Desktop/3]
$ ./c.out
ENTER USERNAME TO PROCEED <^~^> ::USER1
SELECT PROTOCOL [1]. TCP & [2]. UDP 1
TCP SELECTED. [RELIABLE AND CONNECTION ORIENTED]
⇒ SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice:

(kali@ABDULAHAD)-[~/Desktop/3]
$ ./c.out
ENTER USERNAME TO PROCEED <^~^> ::USER2
SELECT PROTOCOL [1]. TCP & [2]. UDP 1
TCP SELECTED. [RELIABLE AND CONNECTION ORIENTED]
⇒ SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice:
```

## 7. main ()

### Workflow:

1. Creates **server\_files** dir.
2. Binds TCP/UDP sockets to port 15051.
3. Listen to TCP connections.
4. Spawns a thread per client (handles multiple users).

## Client-Side TCP Code Breakdown

### 1. MAKE\_CLIENT\_DIR()

#### What it does:

Creates a client\_files directory for downloaded files.

### 2. MAKE\_TCP\_CONNECT()

#### What it does:

Connects to the server and sends the username (e.g., USERNAME).

### 3. LIST FILE TCP()

**What it does:**

Requests and prints the server's file list.

**EOF Handling:**

Stops when the server sends an EOF marker.

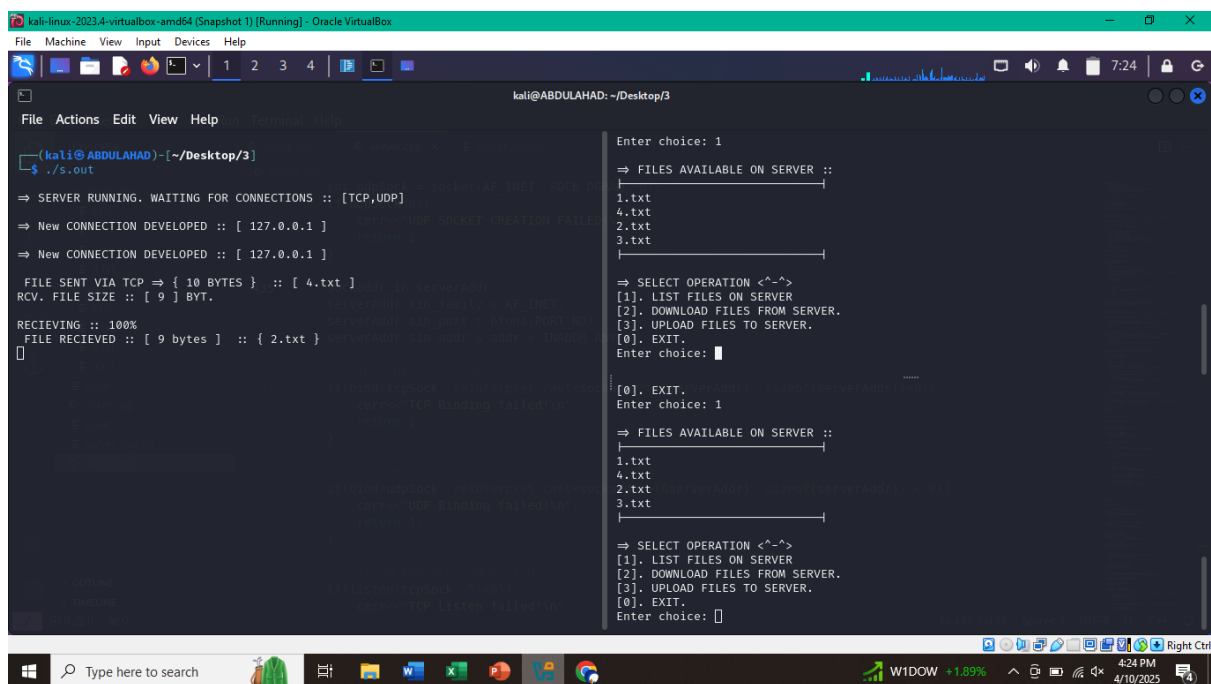
### 4. DOWNLOAD TCP()

**What it does:**

1. Sends DOWNLOAD <filename>.
2. Receives file size first.
3. Saves chunks to client\_files/filename.

**Progress Tracking:**

Shows total bytes received (e.g., 1024/2048 bytes).



```
kali@ABDULAHAD: ~/Desktop/3
$ ./s.out
=> SERVER RUNNING. WAITING FOR CONNECTIONS :: [TCP,UDP]
=> New CONNECTION DEVELOPED :: [ 127.0.0.1 ]
=> New CONNECTION DEVELOPED :: [ 127.0.0.1 ]
FILE SENT VIA TCP => { 10 BYTES } :: [ 4.txt ]
RCV. FILE SIZE :: [ 9 ] BYT.
RECEIVING :: 100%
FILE RECEIVED :: [ 9 bytes ] :: { 2.txt }
[ ]

Enter choice: 1
=> FILES AVAILABLE ON SERVER ::
1.txt
4.txt
2.txt
3.txt
=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: 1
[0]. EXIT.
Enter choice: 1
=> FILES AVAILABLE ON SERVER ::
1.txt
4.txt
2.txt
3.txt
=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: [ ]
```

### 5. UPLOAD TCP()

**What it does:**

1. Sends UPLOAD <filename>.
2. Reads the local file, sends its size (converted to network byte order).
3. Streams chunks to the server.

**Waits for ACK ('1') to confirm success.**

### 5. main ()

**Workflow:**

1. Asks for username and protocol choice (TCP only here).
2. Connects to the server.

### Menu-driven operations:

1. LIST: Calls LIST\_FILE\_TCP ().
2. DOWNLOAD: Calls DOWNLOAD\_TCP ().
3. UPLOAD: Calls UPLOAD\_TCP ().

## Key Observations

### 1. Chunked Transfers:

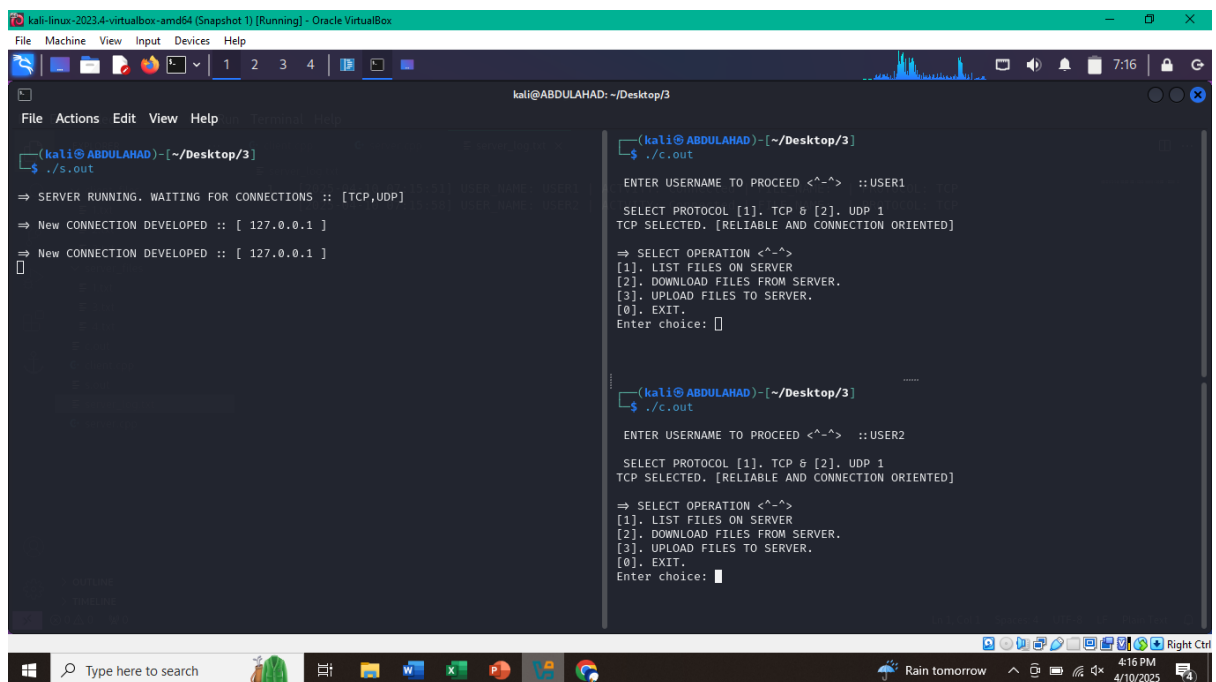
Files are split into 1024-byte chunks to avoid overwhelming the network.

### 2. Error Handling:

Checks file existence (not working properly but still good in terms of detection), logs failures, and validates ACKs.

### 3. Threading:

Server handles multiple clients via threads (one per connection). Although, this is not good for cases when we have many clients, as it will take a lot of resources.



### 4. Network Byte Order:

File sizes are converted using `htobe64()/be64toh()` for cross-platform compatibility.

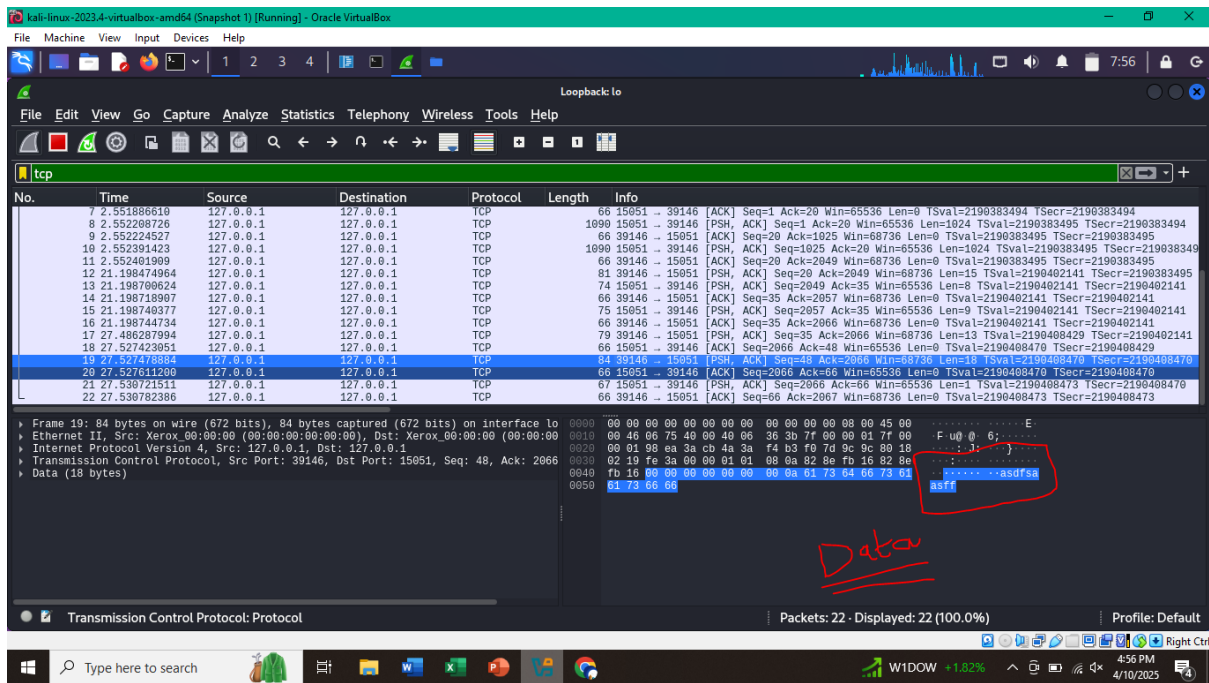
### 5. Testing Notes

Tested with Wireshark to confirm TCP streams. And proper data transfer. First, there was some issues with the cases when the file size was smaller than buffer, it got filled with the garbage. I handled it with flags to detect the length of input data while



reading the file and then made a new string to store the data up till no null character is found.

Since both the sending and receiving sides were at localhost so I used Loop Back analysis for the below image.



**I used almost similar approach for the UDP as well, just we must change the SOCK\_STREAM to SOCK\_DGRAM.**

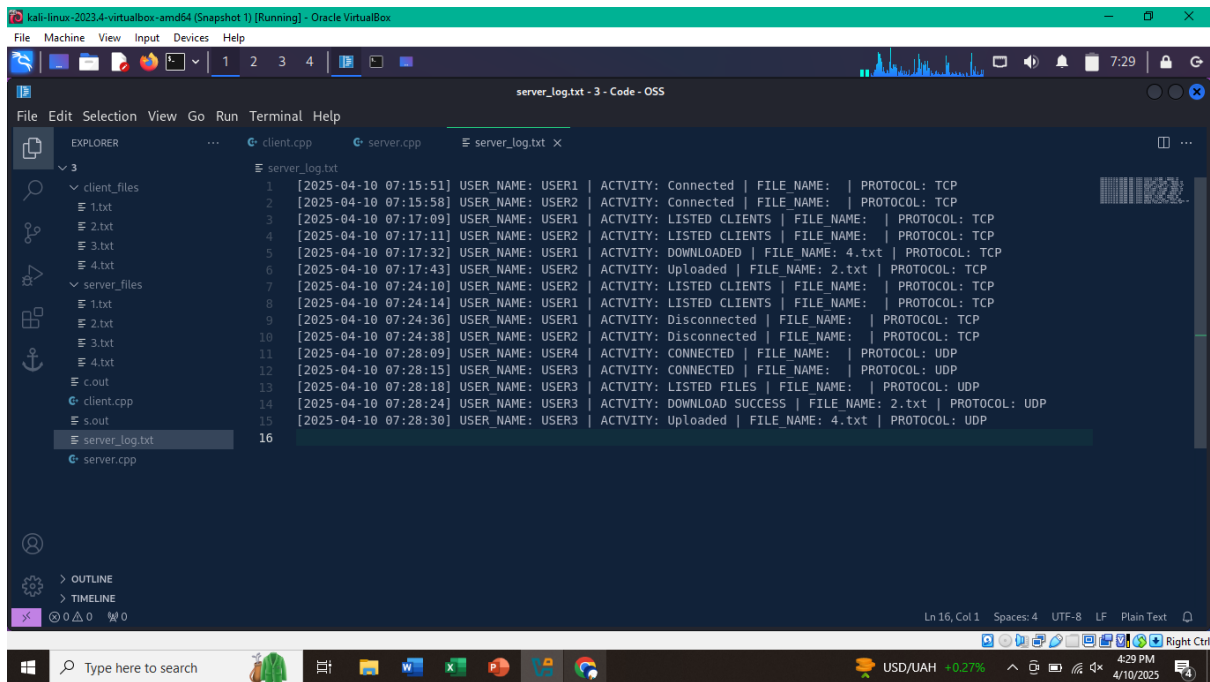
## UDP File Transfer:

### Server-Side UDP Code Breakdown:

#### 1. LOG\_ACTIVITY()

What it does:

1. Logs user actions (login, upload, download) with timestamps to server\_log.txt.
2. Tracks: Username, Action, Filename, Protocol (UDP).



```

1 [2025-04-10 07:15:51] USER_NAME: USER1 | ACTIVITY: Connected | FILE_NAME: | PROTOCOL: TCP
2 [2025-04-10 07:15:58] USER_NAME: USER2 | ACTIVITY: Connected | FILE_NAME: | PROTOCOL: TCP
3 [2025-04-10 07:17:09] USER_NAME: USER1 | ACTIVITY: LISTED CLIENTS | FILE_NAME: | PROTOCOL: TCP
4 [2025-04-10 07:17:11] USER_NAME: USER2 | ACTIVITY: LISTED CLIENTS | FILE_NAME: | PROTOCOL: TCP
5 [2025-04-10 07:17:32] USER_NAME: USER1 | ACTIVITY: DOWNLOADED | FILE_NAME: 4.txt | PROTOCOL: TCP
6 [2025-04-10 07:17:43] USER_NAME: USER2 | ACTIVITY: Uploaded | FILE_NAME: 2.txt | PROTOCOL: TCP
7 [2025-04-10 07:24:10] USER_NAME: USER2 | ACTIVITY: LISTED CLIENTS | FILE_NAME: | PROTOCOL: TCP
8 [2025-04-10 07:24:14] USER_NAME: USER1 | ACTIVITY: LISTED CLIENTS | FILE_NAME: | PROTOCOL: TCP
9 [2025-04-10 07:24:36] USER_NAME: USER1 | ACTIVITY: Disconnected | FILE_NAME: | PROTOCOL: TCP
10 [2025-04-10 07:24:38] USER_NAME: USER2 | ACTIVITY: Disconnected | FILE_NAME: | PROTOCOL: TCP
11 [2025-04-10 07:28:09] USER_NAME: USER4 | ACTIVITY: CONNECTED | FILE_NAME: | PROTOCOL: UDP
12 [2025-04-10 07:28:15] USER_NAME: USER3 | ACTIVITY: CONNECTED | FILE_NAME: | PROTOCOL: UDP
13 [2025-04-10 07:28:18] USER_NAME: USER3 | ACTIVITY: LISTED FILES | FILE_NAME: | PROTOCOL: UDP
14 [2025-04-10 07:28:24] USER_NAME: USER3 | ACTIVITY: DOWNLOAD SUCCESS | FILE_NAME: 2.txt | PROTOCOL: UDP
15 [2025-04-10 07:28:30] USER_NAME: USER3 | ACTIVITY: Uploaded | FILE_NAME: 4.txt | PROTOCOL: UDP
16

```

### Why it matters:

Helps debug issues (e.g., failed uploads) and monitor server activity.

## 2. MAKE\_SERVER\_DIR ()

### What it does:

Create a **server\_files** directory if it doesn't exist.

```

// FUNCTION TO CREATE THE SERVER DIRECTORY,
// ACTUALLY FOR TESTING FILES, I HAVE ALSO USED THE
// SIMILAR APPROACH ON THE CLIENT SIDE AS WELL.
void MAKE_SERVER_DIR(){
    if(!filesystem::exists("server_files")){
        filesystem::create_directory("server_files");
        cout<<"\n MADE THE DIRECTORY :: "<<"server_files"<<endl;
    }
}

```

### Why it matters:

Ensures a dedicated folder for storing uploaded files.

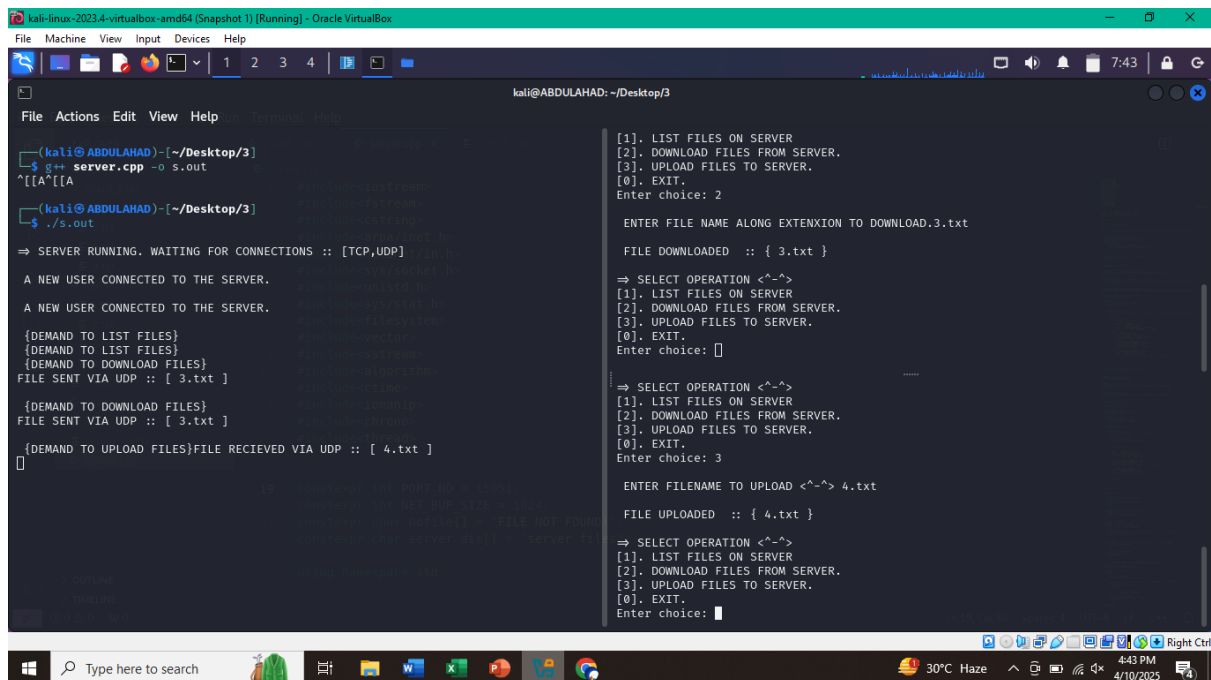
## 3. SEND\_FL\_UDP ()

### What it does:

1. Lists all files in **server\_files** and sends them to the client in chunks via UDP.
2. Ends transmission with an EOF marker.

## Key Challenge:

UDP is connectionless, so each chunk is sent independently (no guaranteed order).



```
kali@ABDULAHAD: ~/Desktop/3
$ g++ server.cpp -o s.out
$ ./s.out

=> SERVER RUNNING. WAITING FOR CONNECTIONS :: [TCP,UDP]

A NEW USER CONNECTED TO THE SERVER.
A NEW USER CONNECTED TO THE SERVER.

{DEMAND TO LIST FILES}
{DEMAND TO LIST FILES}
{DEMAND TO DOWNLOAD FILES}
FILE SENT VIA UDP :: [ 3.txt ]

{DEMAND TO DOWNLOAD FILES}
FILE SENT VIA UDP :: [ 3.txt ]

{DEMAND TO UPLOAD FILES}FILE RECIEVED VIA UDP :: [ 4.txt ]

[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: 2

ENTER FILE NAME ALONG EXTENXION TO DOWNLOAD.3.txt

FILE DOWNLOADED :: { 3.txt }

=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice:

=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: 3

ENTER FILENAME TO UPLOAD <^~^> 4.txt

FILE UPLOADED :: { 4.txt }

=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: ]
```

## 4. HANDLE\_DOWNLOAD\_UDP ()

**What it does:**

1. Checks if the requested file exists.
2. Stream the file in fixed-size chunks (1024 bytes).
3. Ends with an EOF marker.
4. Unlike TCP, UDP doesn't confirm delivery (clients may miss packets).

## 5. RECIEVE\_FILES\_UDP ()

**What it does:**

1. Creates a new file in server\_files.
2. Writes received chunks until EOF is detected.

**Risk:**

Packets may arrive out of order or get lost (no retransmission).

## 6. UDP\_CONNECTION ()

**What it does:**

Main UDP handler for client commands:

1. LIST: Calls SEND\_FL\_UDP ().
2. DOWNLOAD <filename>: Calls HANDLE\_DOWNLOAD\_UDP ().
3. UPLOAD <filename>: Calls RECIEVE\_FILES\_UDP ().

**Key Difference from TCP:**

No persistent connection, each request is standalone.

## 7. main ()

### **Workflow:**

1. Creates **server\_files** dir.
2. Binds UDP socket to port 15051.
3. Spawns a thread to handle UDP requests indefinitely.

## Client-Side UDP Code Breakdown

### 1. MAKE\_CLIENT\_DIR()

#### **What it does:**

Creates a **client\_files** directory for downloaded files.

```
// MAKE CLIENT DIRECTORY [JUST FOR EASE IN TESTING]
void MAKE_CLIENT_DIR(){
    if(!filesystem::exists(client_dir)){
        filesystem::create_directory(client_dir);
        cout<<"\n=> CREATED DIRECTORY :: [ "<<client_dir<<" ]"<<endl;
    }
}
```

### 2. MAKE\_UDP\_CONNECT ()

#### **What it does:**

Send the username to the server (e.g., USERNAME).



```
kali@ABDULAHAD: ~/Desktop/3
$ g++ server.cpp -o s.out
^[[A^[[A
kali@ABDULAHAD: ~/Desktop/3
$ ./s.out
=> SERVER RUNNING. WAITING FOR CONNECTIONS :: [TCP,UDP]

A NEW USER CONNECTED TO THE SERVER.
A NEW USER CONNECTED TO THE SERVER.

{DEMAND TO LIST FILES}
{DEMAND TO LIST FILES}
{DEMAND TO DOWNLOAD FILES}
FILE SENT VIA UDP :: [ 3.txt ]

{DEMAND TO DOWNLOAD FILES}
FILE SENT VIA UDP :: [ 3.txt ]

{DEMAND TO UPLOAD FILES}FILE RECIEVED VIA UDP :: [ 4.txt ]
^

[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: 2

ENTER FILE NAME ALONG EXTENXION TO DOWNLOAD.3.txt

FILE DOWNLOADED :: { 3.txt }

=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: ^

=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: 3

ENTER FILENAME TO UPLOAD <^~^> 4.txt

FILE UPLOADED :: { 4.txt }

=> SELECT OPERATION <^~^>
[1]. LIST FILES ON SERVER
[2]. DOWNLOAD FILES FROM SERVER.
[3]. UPLOAD FILES TO SERVER.
[0]. EXIT.
Enter choice: ^
```

## No Guaranteed Delivery:

The server might miss packets (not retry).

## 6. main ()

### Workflow:

1. Asks for username.
2. Connects to the server via UDP.
3. Menu-driven operations:
4. LIST: Calls LIST\_FILE\_UDP ().
5. DOWNLOAD: Calls DOWNLOAD\_UDP ().
6. UPLOAD: Calls UPLOAD\_UDP ().

## Key Observations

### 1. Connectionless Nature:

UDP doesn't guarantee delivery or order (unlike TCP).

### 2. Chunked Transfers:

Files are split into 1024-byte chunks to fit UDP datagrams.

### 3. No ACKs or Retries:

Missing packets aren't detected or resent (risk of corruption). Since it is not connection oriented it is generally fast. Plus, small header overhead, only 8 bytes unless TCP's 20 bytes.

### 4. EOF Marker:

Used to signal the end of a file/list transmission.

kali-linux-2023.4-virtualbox-amd64 (Snapshot 1) [Running] - Oracle VirtualBox

File Machine View Input Devices Help

\*Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	56	56291 → 15051 Len=14
2	10.313343158	127.0.0.1	127.0.0.1	UDP	57	56291 → 15051 Len=15
3	108.110394900	127.0.0.1	127.0.0.1	UDP	62	15051 → 56291 Len=10
4	10.313594281	127.0.0.1	127.0.0.1	UDP	1066	15051 → 56291 Len=1024
5	54.633465660	127.0.0.1	127.0.0.1	UDP	55	56291 → 15051 Len=13
6	54.633497014	127.0.0.1	127.0.0.1	UDP	53	56291 → 15051 Len=11
7	54.633593295	127.0.0.1	127.0.0.1	UDP	1066	56291 → 15051 Len=1024

Frame 3: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface lo  
Ethernet II, Src: Xerox 00:00:00:00:00:00, Dst: Xerox 00:00:00:00:00:00  
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1  
User Datagram Protocol, Src Port: 15051, Dst Port: 56291

0000 45 06  
0010 00 26 72 03 40 00 48 11 ca c1 7f 00 00 01 7f 00  
0020 00 01 3a cb db e3 00 12 fe 25 6d 61 6e 67 6f 20  
0030 6d 61 6e 00

...&r 0 0 .....E  
...: : : : : %mango  
...: : : : : nati

wireshark\_lo6XLY42.pcapng Packets: 7 - Displayed: 7 (100.0%) Profile: Default

Type here to search 30°C Haze 4:45 PM 4/10/2025