National University of Computer and Emerging Sciences
Islamabad Campus

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

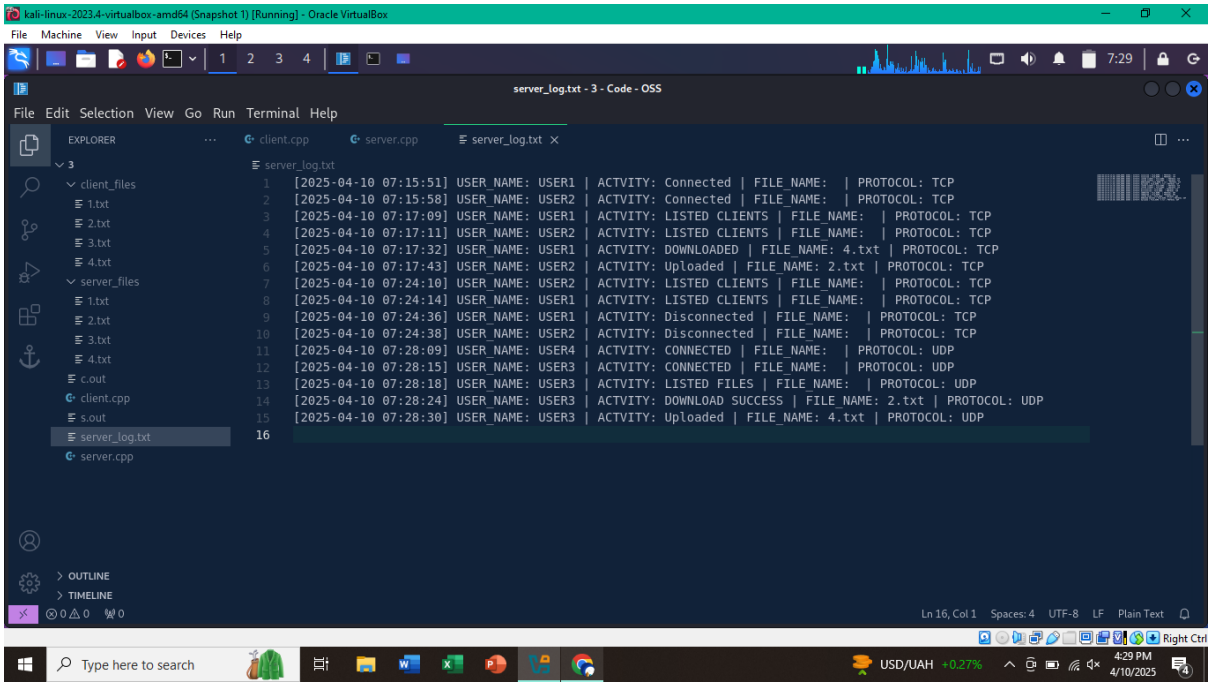
TCP File Transfer:

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).



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 APPROAH ON THE CLIENT SIDE AS WELL.
void MAKE_SERVER_DIR(){
    if(!filesystem::exists("server_files")){
        filesystem::create_directory("server_files");
        cout<<"\n MADE TEH 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., RECIEVING :: 75%).

6. HANDLE_TCP_SOCK()

What it does:

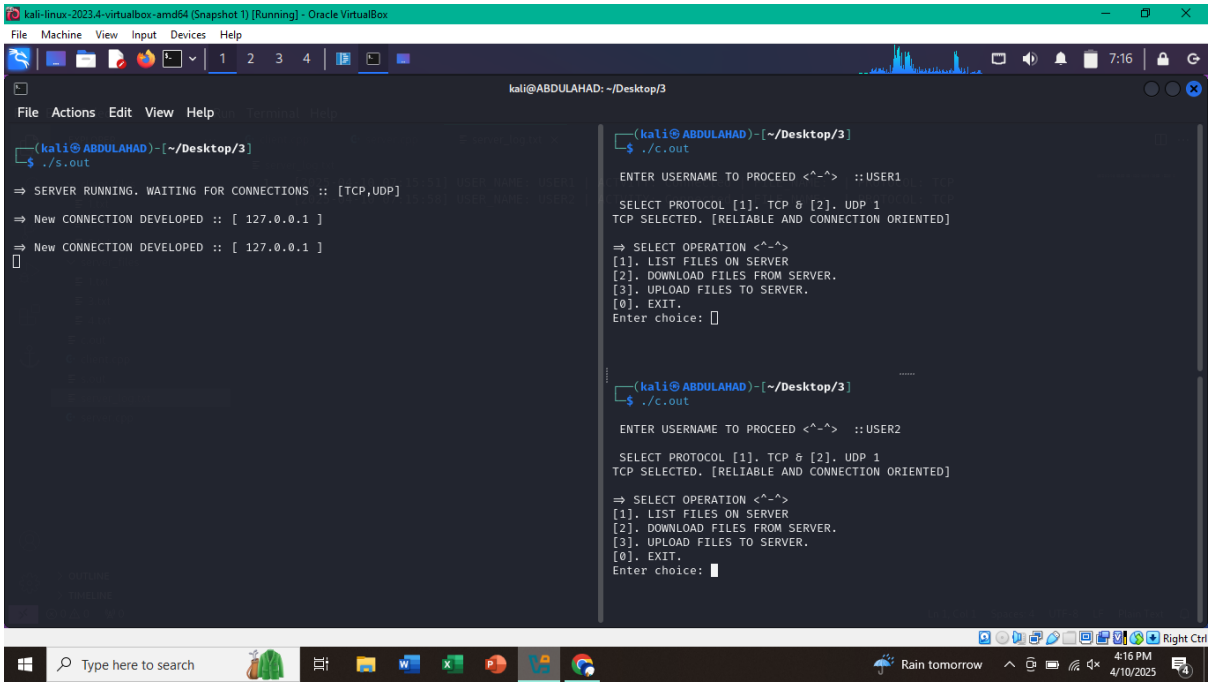
Main TCP handler for client commands:

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.



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.

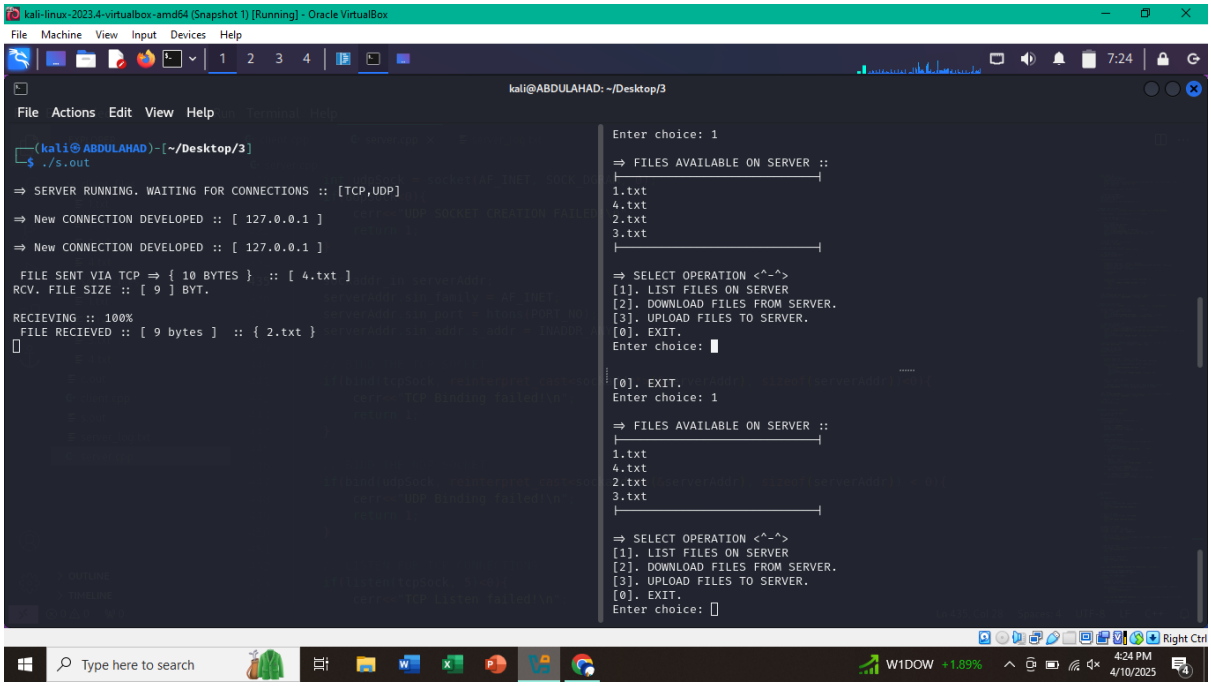
4. DOWNLOAD_TCP()

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).



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:

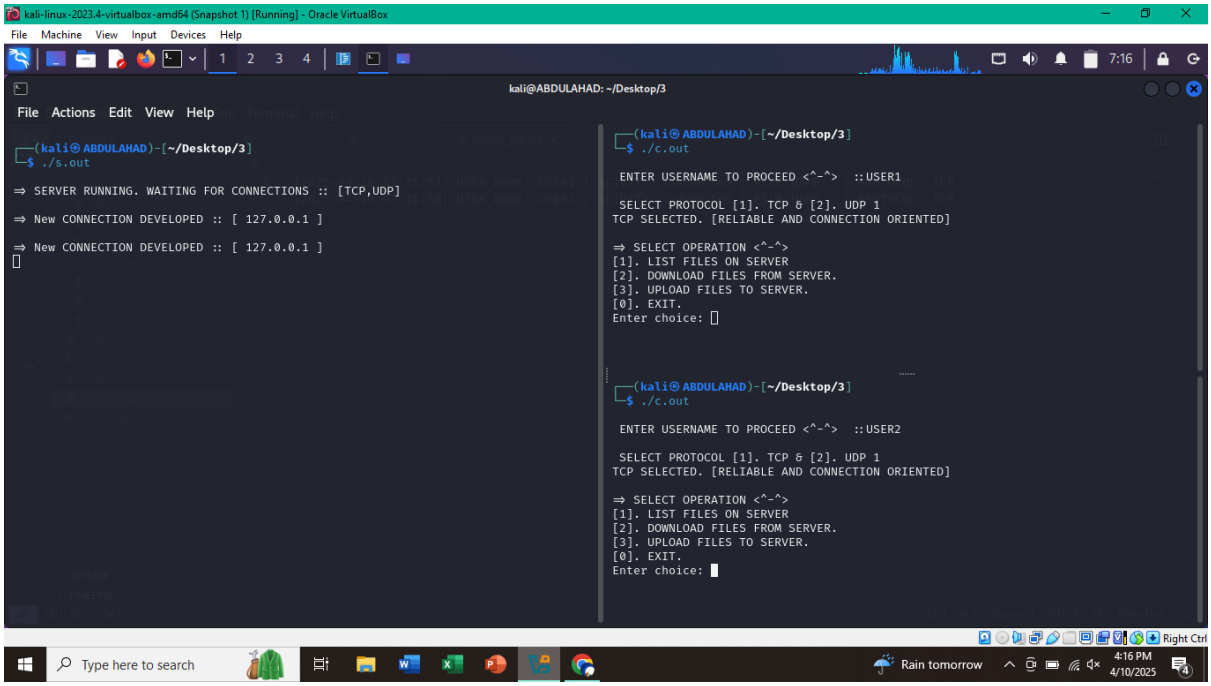
Files are split into 1024-byte chunks to avoid overwhelming the network.

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:

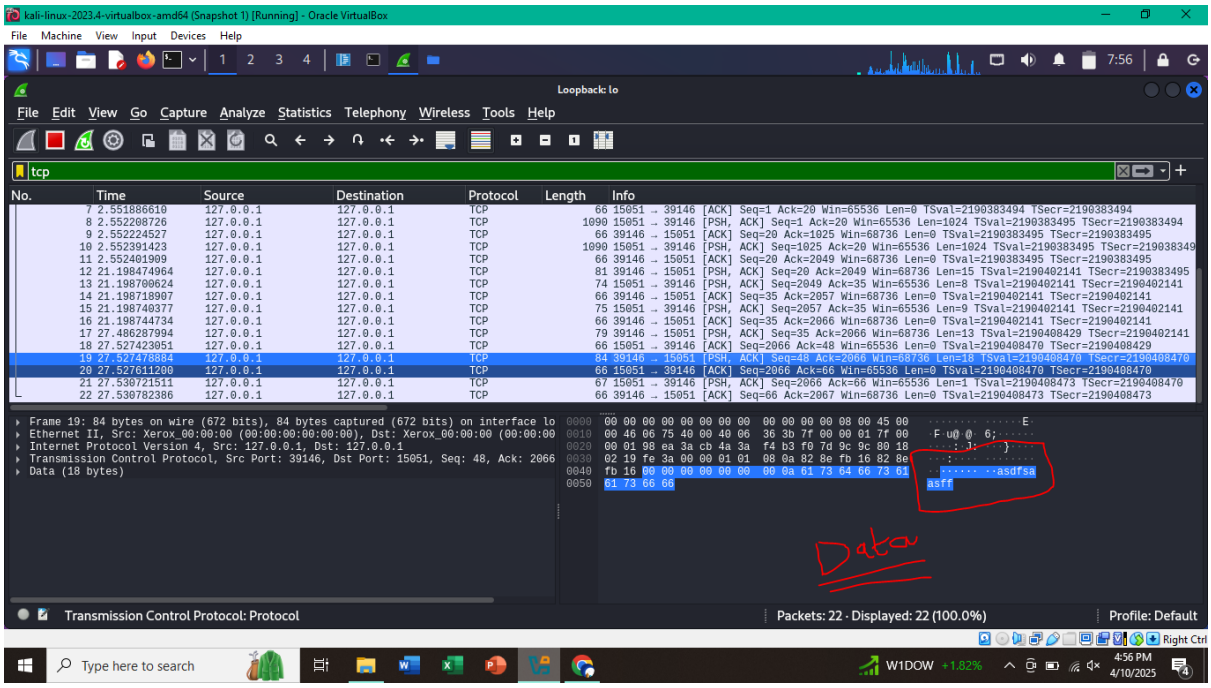
File sizes are converted using **htobe64()/be64toh()** for cross-platform compatibility.

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.



<u>I used almost similar approach for the UDP as well, just we must change the SOCK_STREAM to SOCK_DGRAM.</u>

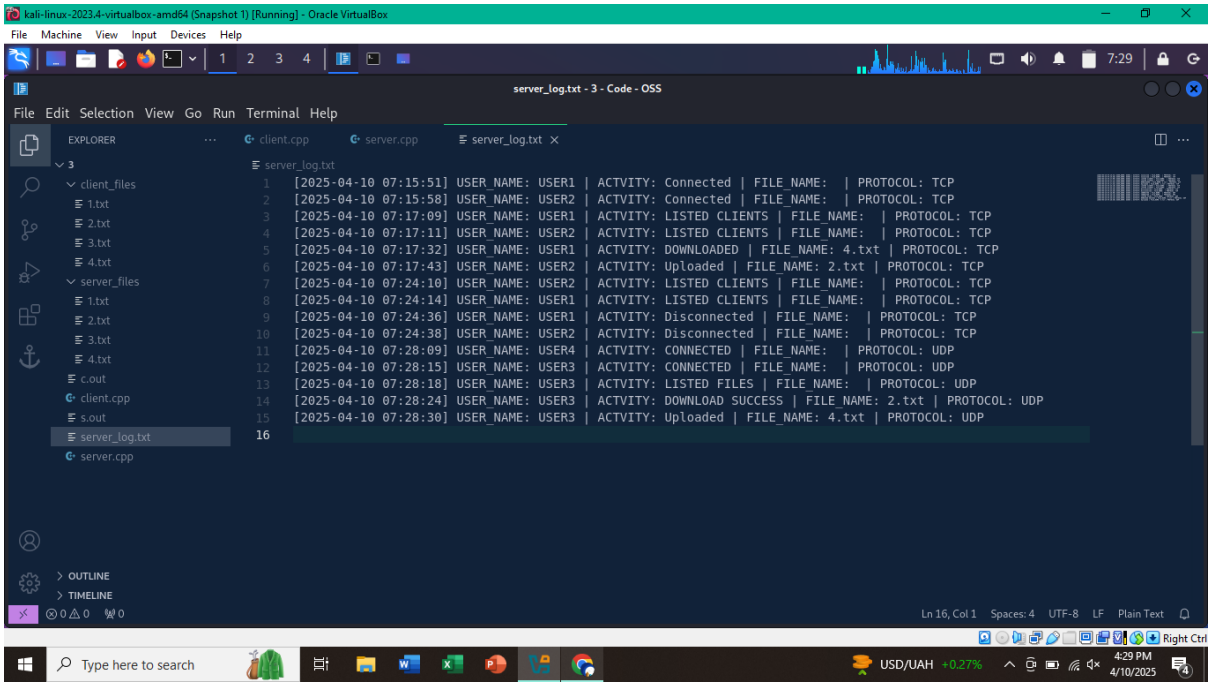
UDP File Transfer:

<u>Server-Side UDP Code Breakdown:</u>

<u>1. LOG_ACTIVITY ()</u>

What it does:

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



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 APPROAH ON THE CLIENT SIDE AS WELL.
void MAKE_SERVER_DIR(){
    if(!filesystem::exists("server_files")){
        filesystem::create_directory("server_files");
        cout<<"\n MADE TEH DIRECTORY :: "<<"server_files"<<endl;
    }
}
```

Why it matters:

Ensures a dedicated folder for storing uploaded files.

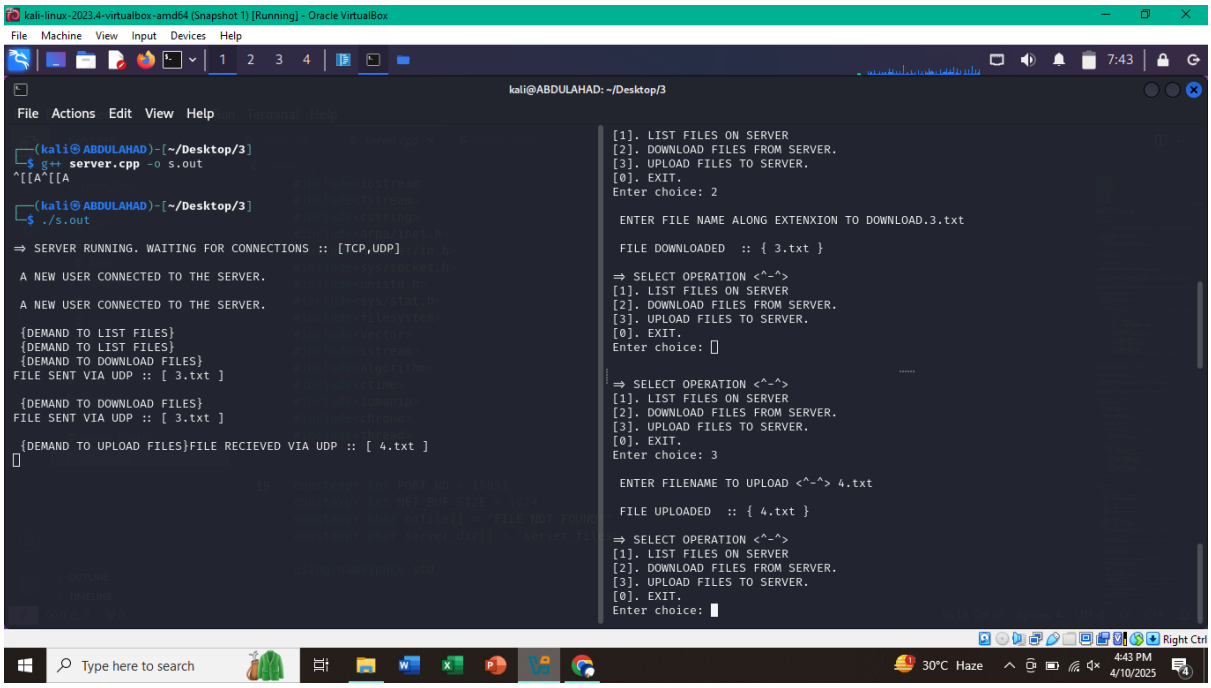
3. SEND_FL_UDP ()

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).



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:

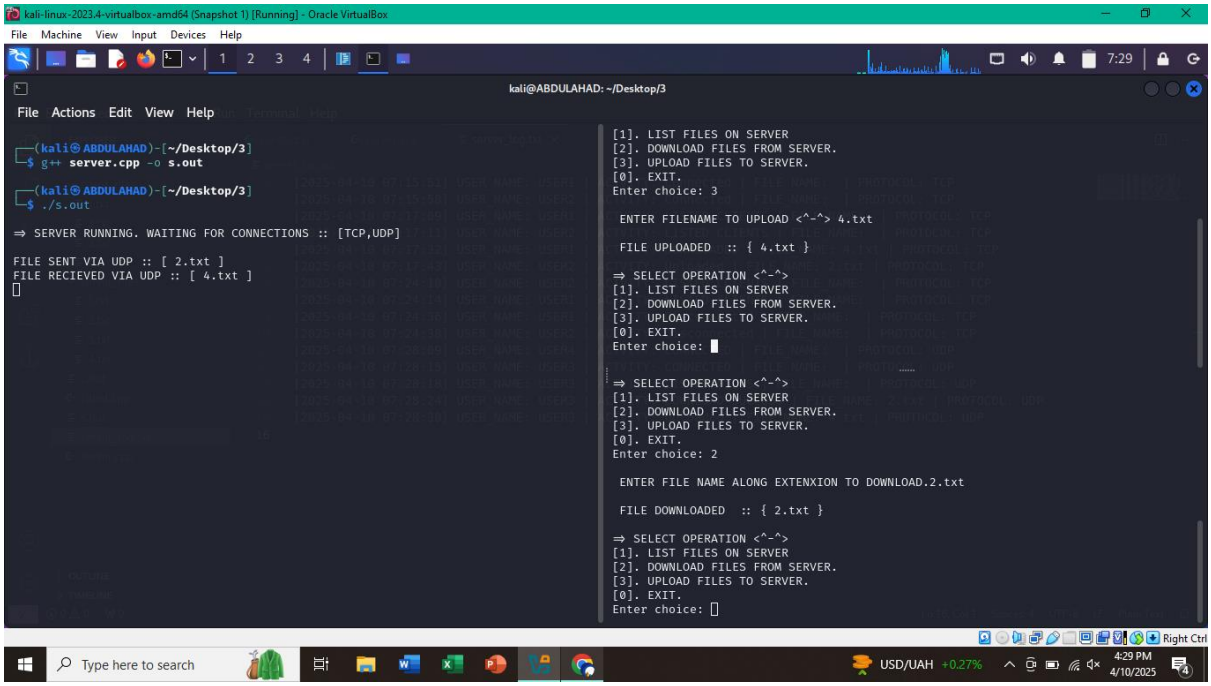
Creates a **client_files** directory for downloaded files.

```
// 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).



No Handshake:

UDP doesn't establish a connection—just fires and forgets.

3. LIST_FILE_UDP ()

What it does:

Requests and prints the server's file list.

EOF Handling:

Stops when the server sends an EOF marker.

4. DOWNLOAD_UDP ()

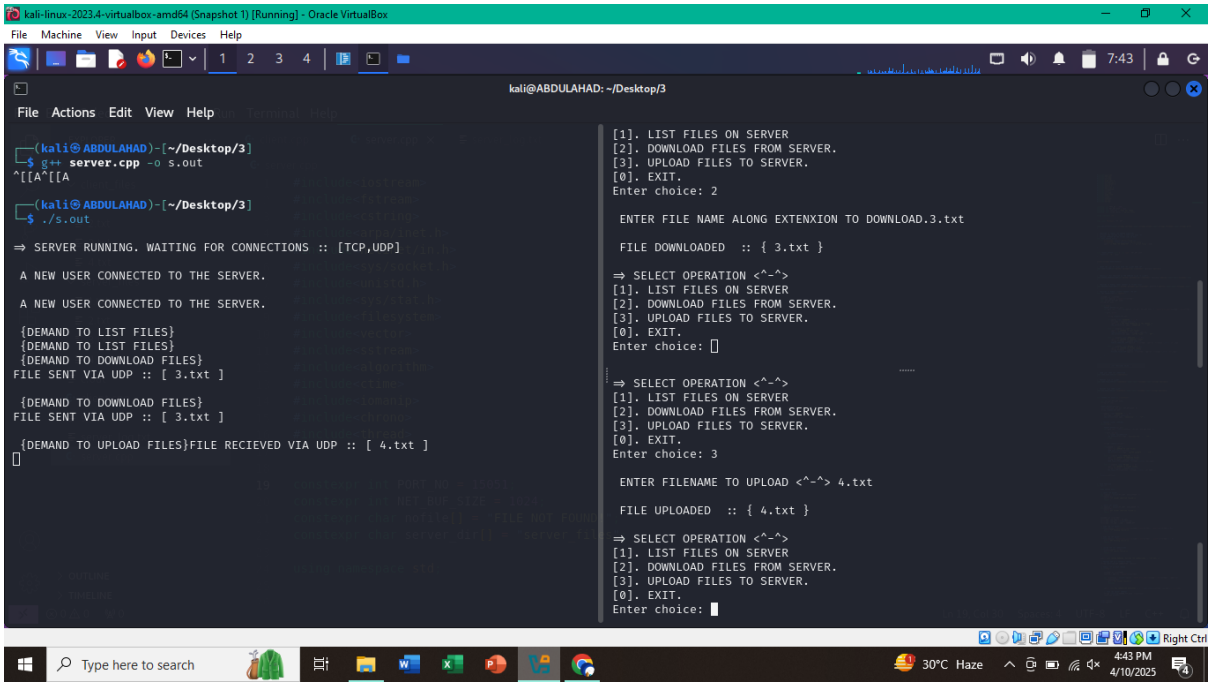
What it does:

1. Sends DOWNLOAD <filename>.
2. Receives chunks and saves them to **client_files/filename**.
3. Unlike TCP, the client doesn't know the file size upfront.

5. UPLOAD_UDP ()

What it does:

1. Sends UPLOAD <filename>.
2. Reads the local file and streams chunks to the server.
3. Ends with an EOF marker.



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:

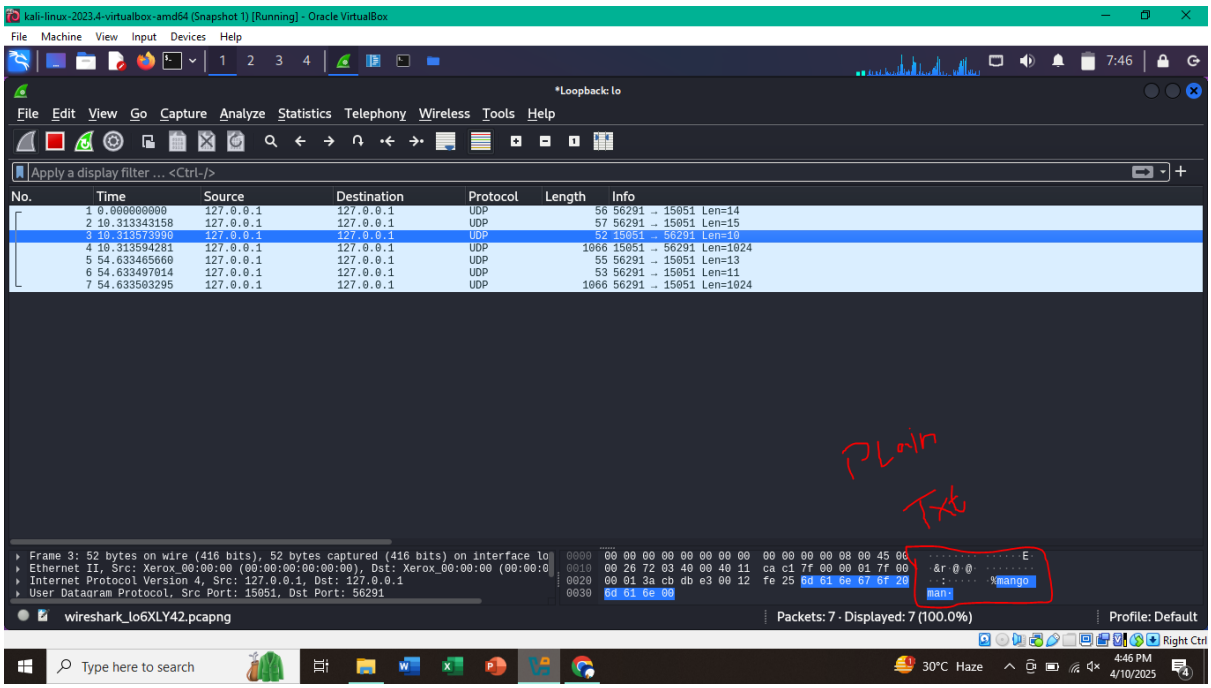
Files are split into 1024-byte chunks to fit UDP datagrams.

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 10.313573990 127.0.0.1 127.0.0.1 UDP 52 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 54.633503295 127.0.0.1 127.0.0.1 UDP 1066 56291 → 15051 Len=1024
Plain Text
Frame 3: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface lo
Ethernet II, Src: Xerox_00:00:00 (00:00:00:00:00:00), Dst: Xerox_00:00:00 (00:00:0
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 15051, Dst Port: 56291
0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 06
0010 00 26 72 03 40 00 40 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
man·
wireshark_lo6XLY42.pcapng
Packets: 7 · Displayed: 7 (100.0%)
Profile: Default
Right Ctrl
Type here to search
30°C Haze
4:46 PM
4/10/2025