Csd 2161 Networking

Assignment 3: *File Download over UDP*

Technical Design Document

Group Members:

Joel Jie Lee, 2301419 joeljie.lee@digipen.edu

Rachel Lim Jia En, 2301526 racheljiaen.l@digipen.edu

Tan Yee Tong, 2301414 yeetong.t@digipen.edu

Design Introduction

TCP:

enum CMDID {

UNKNOWN = (unsigned char)0x0,

REQ\_QUIT = (unsigned char)0x1,

REQ\_DOWNLOAD = (unsigned char)0x2,

RSP\_DOWNLOAD = (unsigned char)0x3,

REQ\_LISTFILES = (unsigned char)0x4,

RSP\_LISTFILES = (unsigned char)0x5,

DOWNLOAD\_ERROR = (unsigned char)0x30

FILE\_TRANSFER\_DONE = (unsigned char)0x50

};

**UNKNOWN**: Client inputs in the wrong Error

**REQ\_QUIT** = Client requesting to quit the program

**REQ\_DOWNLOAD** = Client request to download program

**RSP\_DOWNLOAD** = Client receives this response from the Server

**REQ\_LISTFILES** = Client to request a list of files for them to choose what to download

**RSP\_LISTFILES** = Server Response with the list of files that can be downloaded

**DOWNLOAD\_ERROR** = Client-Server to indicate that the download cannot be done

**FILE\_TRANSFER\_DONE** = Server to signal the client that the file transfer is done

TCP Protocol

**REQ\_LISTFILES & RSP\_LISTFILES (TCP Connection):**



Client sends REQ\_LISTFILES to Server to request for a list of files to download.

Server receives REQ\_LISTFILES, it will send RSP\_LISTFILES in the format below.



| Command ID | Number of files | Length of file list, N | Filename length L1 | Filename  of file 1 | ….. | Filename length Ln | Filename  of file N |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 byte | 2 byte | 4 bytes | 4 bytes | L1 bytes | …… | 4 bytes | Ln bytes |

**REQ\_DOWNLOAD & RSP\_DOWNLOAD (TCP Conection):**



**TCP CONNECTION:**

Client sends REQ\_DOWNLOAD to Server to request to download a file and Server receives the message in the same format below.

| Command ID | IP Address | UDP Port ID | Filename length | “F” | “I” | “L” | “E” | “.” | “c” | “p” | “p” |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 byte | 4 byte | 2 bytes | 4 bytes | 1 byte | 1 byte | 1 byte | 1 byte | 1 byte | 1 byte | 1 byte | 1 byte |

After the Server Receives REQ\_DOWNLOAD, it will send an RSP\_DOWNLOAD to the Client in the format below:

| Command ID | IP Address | Port Number | Session ID | File Length |
| --- | --- | --- | --- | --- |
| 1 byte | 4 bytes | 2 bytes | 4 bytes | 4 Bytes |

Network File Downloading

In this assignment, our team is using Stop n Wait, via UDP.

Server Sends to Client: [Checksum, 2][Sequence number, 4][Data Length, 4][File data…]

Client Sends to Server: [Checksum, 2][Session Number, 4][ACK, 4]

Timeout is also used to prevent block on the thread. This means that if it runs out it indicates failure, then the packet is being resend/discarded depending on the situation.

Sequential ACK starts from 1 to infinity, the number will keep increasing upon successfully receiving the packets.

ACK represents last successful packet received.

Sequence Number represents current packet being/to send.

Checksum:

uint16\_t CalculateChecksum(size\_t length\_of\_data, void\* data)

* Read data as uint16\_t, and add to checksum value
* If not enough bits then pad with 0s.

bool ValidateChecksum(size\_t length\_of\_data, void\* data, uint16\_t checksum\_val)

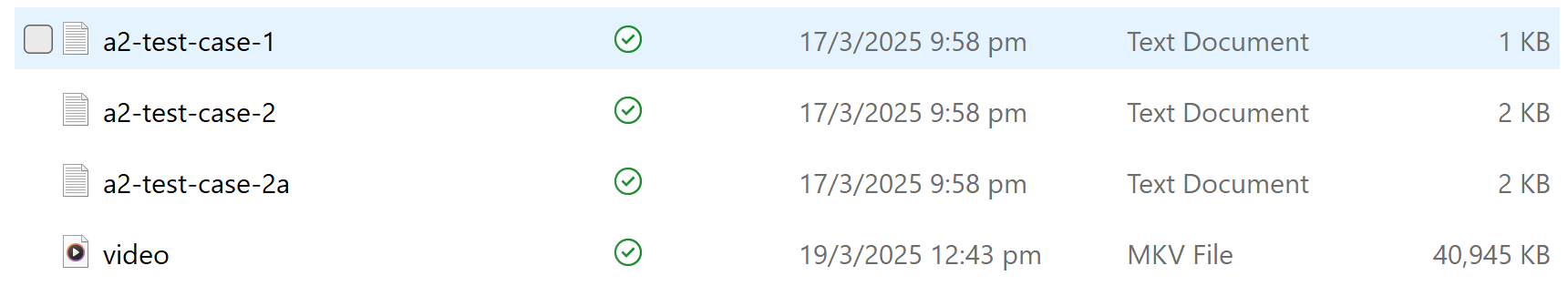
* Calculate checksum on data received
* Add together with checksum received, if they both add up to 0xFFFF then pass.

Timeout:

* 0.5s by default, can be modified in Config.txt.
* Reset when packet is sent/resent.
* Individual timeout for each session.

# Server

File to be downloaded stored in Download folder:



File Download Protocol

* Server sends the packet to client port, storing the packet in a buffer first.
* It waits for ACK from client, with the expected received ACK being the sequence number of the packet.
  + **Test Case 1:** Received the correct ACK: send next packet and increment sequence number.
  + **Test Case 2:** Receive a Wrong ACK: Do nothing, wait for timeout or correct ack.
  + **Test Case 3:** Did not receive the ACK before timeout, resend packet.
  + **Test Case 4:** Receive Duplicated ACK (of a lower number), Do nothing, wait for timeout or correct ack.

For server.exe to work, Config.txt must be present.

# Client

File downloaded are stored in Download folder

Under Specifications:

* File size should be less than or equal to 200 MB
* The number of downloading files must be not more than 256
* Multiple files can be simultaneously transmitted to different clients
* Make sure the clients have enough hard disks to store the received files

File Download Protocol

* The client receives the packet from the server and sends an ACK.
* Packet is checked for corruption: If checksum fails, then wrong sequence number is assumed.
* Otherwise, check for sequence number > ACK variable.
  + If successful, then increment ACK to reflect that latest successful packet is received. Write packet to file, and increment bytes\_read
  + If failed, then leave ACK as is.
* ACK is send regardless of success/failure, as long as packet was read.
* Message format: [Checksum, 2][Session, 4][ACK, 4]

Communication Protocol:

* /l: request list of available files
* /d [IP:Port] [filename]: send request file download (REQ\_DOWNLOAD) to Server with Client’s IP address and port number
* If file exists, Server responds to Client with RSP\_DOWNLOAD for downloading details, otherwise send DOWNLOAD\_ERROR
* Client1 acknowledges the receiving of datagram for file download
* Client2 can request file list and request downloading simultaneously
* /q: quit

Note:

* No blocking of stdin while downloading, input can be entered.
  + However, the result of the command is only shown after downloading finished.
  + This meets rubrics, as stdin is not being blocked.

**Client receives from INPUT:**

**/l** REQ LISTFILES = (unsigned char)0x4

* Send server command ID: REQ\_LISTFILES

**/d** REQ DOWNLOAD = (unsigned char)0x2

* Send server command ID: REQ\_DOWNLOAD

**Client Receives from Server**

Server send command ID: RSP\_LISTFILES 0x5

* Client receive command ID: RSP\_LISTFILES 0x5

Server send command ID: RSP\_DOWNLOAD 0x3

* Client receive command ID: RSP\_DOWNLOAD 0x3

# UDP Protocol

When the server receives the REQ\_DOWNLOAD from the Client. The server will respond with RSP\_DOWNLOAD and at the same time also send a datagram to the Client.

Server Sends to Client: [Checksum, 2][Sequence number, 4][Data Length, 4][File data…]

Sending of data is done over UDP, where a separate single thread runs recv/sendto constantly in a loop for the udp download socket.

**Datagram and ACK (UDP Connection):**



To keep track of all the file sessions, we are using mapping and our own defined struct File\_Sessions via **std::map<int, File\_Session> file\_sessions{};**

**struct File\_Session**

{

//Represents the current packet to send.

int current\_sequence\_number{ -1 };

std::string dest\_ip\_addr{};

std::string dest\_port\_num{};

std::filesystem::path file\_path{};

double time\_last\_packet\_sent{};

bool isSend{ false };

sockaddr\_storage addrDest{};

};

Each struct keeps track of one client’s file download session.

**Important parameters read from the file include:**

* Config.txt
  + Udp\_packet\_size 512 by default.
  + Timeout: 0.5 by default.

**When does a session consider COMPLETE?**

A file transfer is considered complete when: file\_length == file data sent/read

When this happens:

* Both client and server automatically stop file download.
* The server will stop the file transfer via file\_sessions.erase(session);
* The client just quits the while loop and continues interactions with server.

# Server-Client Relationship (UDP)

**Whenever the Server sends a datagram**:

* It will calculate the checksum value, sequence number (session ID), data and data length.

**Whenever the Server receives a datagram**, it checks whether the checksum is valid:

* If ACK is expected, then increment the sequence number and send the next packet.
  + File\_sessions[session].current\_sequence\_number++ (ack value for next packet must match this value)
  + file\_sessions[Session\_number].isSend = true (send next packet)
* If the ACK is wrong, then it just wait for timeout and resends the expected packet.

**Whenever the Client sends a datagram**:

* It will calculate the checksum value, session number (session ID), and ack value.

**Whenever the Client receives a datagram**, it checks whether the checksum is valid:

* If it is valid, then check sequence number > ACK
  + If true, then increment ACK and write packet to file.
  + If not, then do nothing
* Will always send back checksum + session number + ack after receiving packet, whether ack was successful or not.

Test Cases Between Client and Server

ACK is correct if it matches that of the current packet being sent.

**Test Case 1: Received the correct ACK: send next packet**



In this case, the Server increments sequence number and sends new packet immediately.

* Server sends packet 1,and received ACK 1 (normal case), then sends packet 2.

**Test Case 2: Receive a Wrong ACK: Wait for timeout**



In this case, the Server sends a packet 2 and the Client responds with a wrong ACK value. The server will discard the ack and resend packet after timeout.

**Test Case 3: Did not receive the ACK before timeout.**



In this case, The server will resend the Packet again.

**Test Case 4: Receive Duplicated ACK (of a lower number), Do nothing**



* ACK value is wrong, so wait for timeout and resend then.

# File Downloading

The file-downloading process between the client and server is implemented using a Stop-and-Wait protocol over UDP. This ensures reliable file transfer by using checksums, sequence numbers, and acknowledgments (ACKs) to handle packet loss, corruption, and retransmission.

Key Functions and Data Structures

1. **int GetDataFromFile(std::string file, int offset, int bytes\_to\_read, char\* buffer)**

Used by server to read data from file to be sent to the client. It is used to extract chunks of data from the file to be sent to the client in UDP packets.

For example, if the file is 1000 bytes and the server sends 512-byte chunks, GetDataFromFile is called with offset = 0 and bytes\_to\_read = 512 for the first chunk, offset = 512 for the second chunk, and so on.

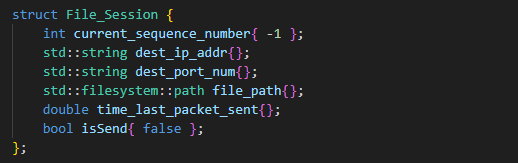
Parameters:

* std::string file: The path to the file being downloaded.
* int offset: Start position to read from file.
* int bytes\_to\_read: Number of bytes to read from file.

Return: The number of bytes successful read from file.

1. **File\_Session Struct**

Used by the server to manage the state of each file download session.



Fields:

* current\_sequence\_number: The sequence number of the next packet to send.
* dest\_ip\_addr: The IP address of the client.
* dest\_port\_num: The UDP port of the client.
* file\_path: The path to the file being downloaded.
* time\_last\_packet\_sent: The timestamp of the last packet sent.
* isSend: A flag indicating whether the next packet should be sent.

1. **uint16\_t CalculateChecksum(size\_t length\_of\_data, void\* data)**

Calculates a 16-bit checksum for each UDP packet before sending it to the client. The checksum is used by the client to validate integrity of the received packet.

Parameters:

* size\_t length\_of\_data: The length of the data block in bytes.
* void\* data: A pointer to the data block.

Return: A 16-bit checksum value.

1. **bool ValidateChecksum(size\_t length\_of\_data, void\* data, uint16\_t checksum\_val)**

This function validates the checksum of a received data block to ensure that the data has not been corrupted during transmission.

Client uses this function to validate the checksum of each received UDP packet.

If the checksum is invalid, the client discards the packet and resends the ACK for the previous successfully received packet.

Parameters:

* size\_t length\_of\_data: The length of the data block in bytes.
* void\* data: A pointer to the data block.
* uint16\_t checksum\_val: The checksum value received from the sender.

Return: true if checksum is valid, otherwise return false

1. **int WriteToDownloadSocket(std::string dest\_ip\_address, std::string dest\_port, char\* data, size\_t num\_bytes)**

This function writes data to the UDP socket for file transfer. It ensures that all data is sent, even if multiple sendto calls are required.

The server uses this function to send UDP packets containing file data to the client.

The client uses this function to send ACKs back to the server.

Parameters:

* std::string dest\_ip\_address: The IP address of the destination (client).
* std::string dest\_port: The UDP port of the destination (client).
* char\* data: A pointer to the data to be sent.
* size\_t num\_bytes: The number of bytes to send.

Return:

* The number of bytes successfully sent.
* -1 on SOCKET\_ERROR.
* 0 if the socket has been gracefully closed.
* -2 if the IP/port cannot be resolved.

1. **void File\_Download\_Interaction()**

This function runs on a separate thread on the server and handles the file download process for all active sessions. It manages packet transmission, ACK reception, and timeout handling.

This function runs on a separate thread on the server and manages all active file download sessions.

It ensures that packets are sent, ACKs are received, and retransmissions are handled as needed.

Function responsibilities:

* Sends file data in chunks to the client.
* Waits for ACKs from the client.
* Handles retransmission in case of timeout or incorrect ACKs.
* Updates the session state (e.g., sequence number, timeout duration).

**File Download Workflow**

1. Client Requests File Download

* The client sends a REQ\_DOWNLOAD command to the server, specifying the file to download.
* Command format: /d [IP:Port] [filename]

1. Server Responds with File Details

* Server checks of file exists
* If file exists, server sends RSP\_DOWNLOAD message containing:
* Command ID (0x3)
* Server IP address (4 bytes)
* Server UDP port (2 bytes)
* Session ID (4 bytes)
* File length (4 bytes)

1. UDP File Transfer Begins

* The server reads the file in chunks using GetDataFromFile and sends each chunk as a UDP packet.
* Each packet contains:
  + Checksum (2 bytes)
  + Sequence number (4 bytes)
  + Data length (4 bytes)
  + File data payload (variable length)

1. Client Validates and Acknowledges

* The client receives each packet, validates the checksum, and sends an ACK back to the server.
* If the checksum is invalid, the client discards the packet and resends the ACK for the previous packet.

1. File Download Completion

* The server continues sending packets until the entire file is transmitted.
* The client saves the file to its local download directory.
* Both the server and client terminate the session.

Timeout Handling

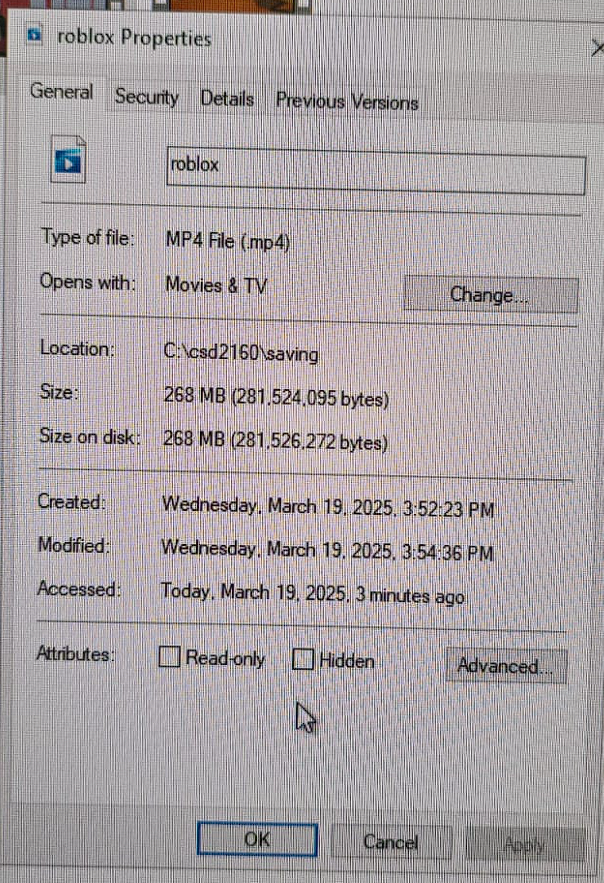
* Timeout Mechanism:
* The server waits for an ACK from the client after sending each packet.
* If the server does not receive an ACK within a timeout period, it retransmits the packet.
* Timeout is 0.5s by default, adjustable in config.txt
* Handling Duplicate ACKs:
  + If the server receives a duplicate ACK (i.e., an ACK for a previously received packet), it ignores the duplicate and continues waiting for the correct ACK.

Example workflow

1. **Client Requests File List:**
   * Client sends: /l
   * Server responds with a list of available files.
2. **Client Requests File Download:**
   * Client sends: /d [IP:Port] [filename]
   * Server responds with file details (RSP\_DOWNLOAD).
3. **UDP File Transfer:**
   * Server sends file data in chunks.
   * Client validates each chunk and sends ACKs.
4. **File Download Completion:**
   * Client saves the file locally.
   * Server and client terminate the download session, and continues interactions.

Testing Client-Server

We are using the roblox.mpv file as the download, which is 286 MB, satisfies the requirement for the MAX file size of 200MB.



**We have done 3 different Client-Server tests**.

1. Downloading via **Local Machine**: Local Server and Multiple Local Clients.

OUTPUT:

* Output on both Server and Clients is expected
* The download is successful for all clients.
* Download time: <10 seconds

1. Downloading over **Wifi**: 1 Machine runs as a Server and supports multiple clients

OUTPUT:

* Output on both Server and Clients is expected
* The download is successful for all clients.
* Download time: < 8 minutes with 3 clients

1. Downloading over **Ethernet**: 1 Machine runs as the Server, the other machine supports multiple clients

OUTPUT:

* Output on both Server and Clients is expected
* The download is successful for all clients.
* Download time: <4 mins for 3 clients

Packet number received will not be indicated in order to reduce our transmission time (by avoiding unnecessary std::cout). Only file completion is indicated.

Errors are also indicated.

When starting the file transfer session, do note that “TIMEOUT” error is outputted onto server console. This is not actually an error, but rather caused by initializing file session with 0 as timeout.

As it is not required in rubrics (only recommended for debug), we have decided it’s ok to leave as is.

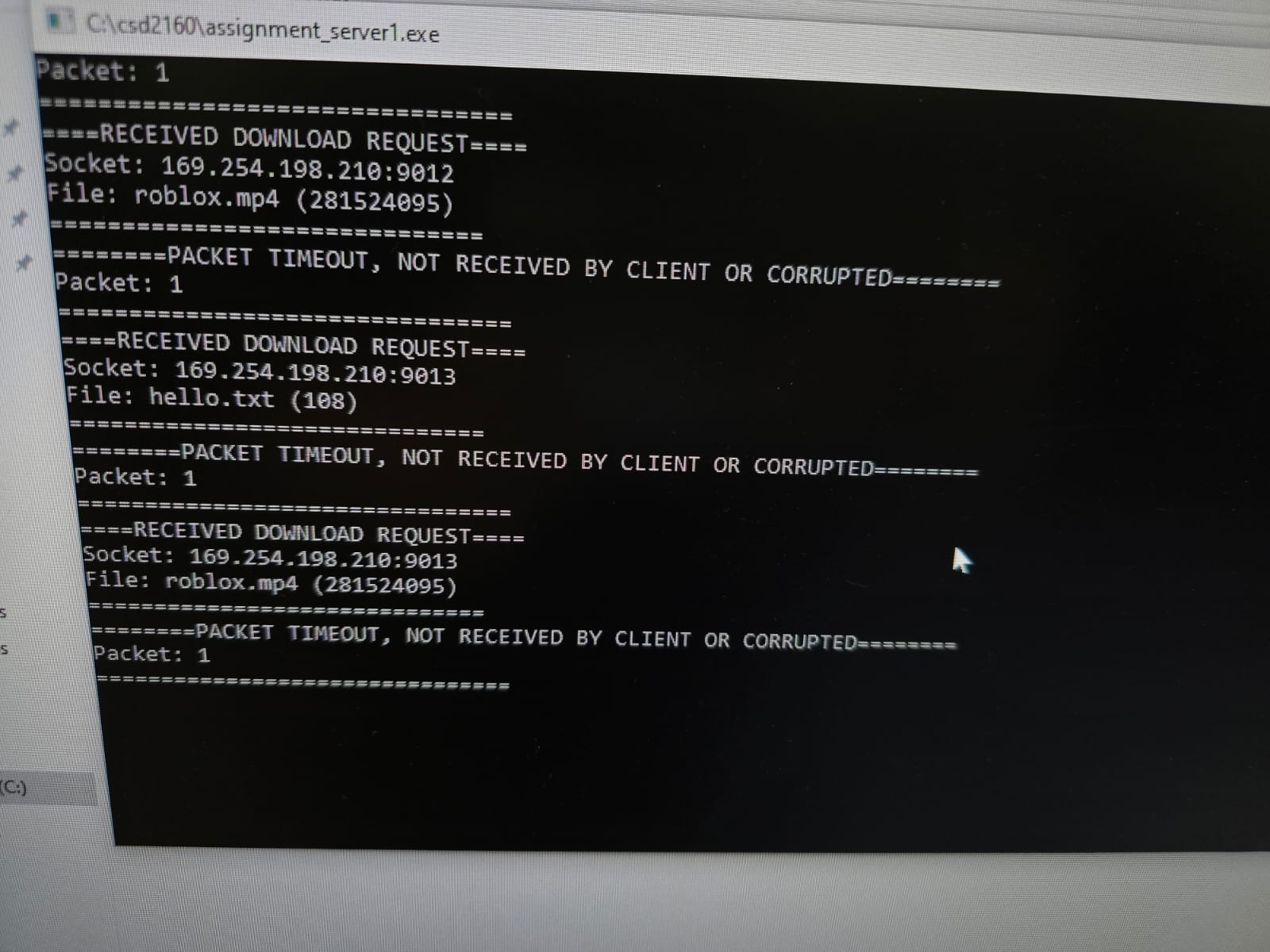
These 3 different tests prove that our code can support multi-threading download over UDP over Wifi, Ethernet, and Local Machine.

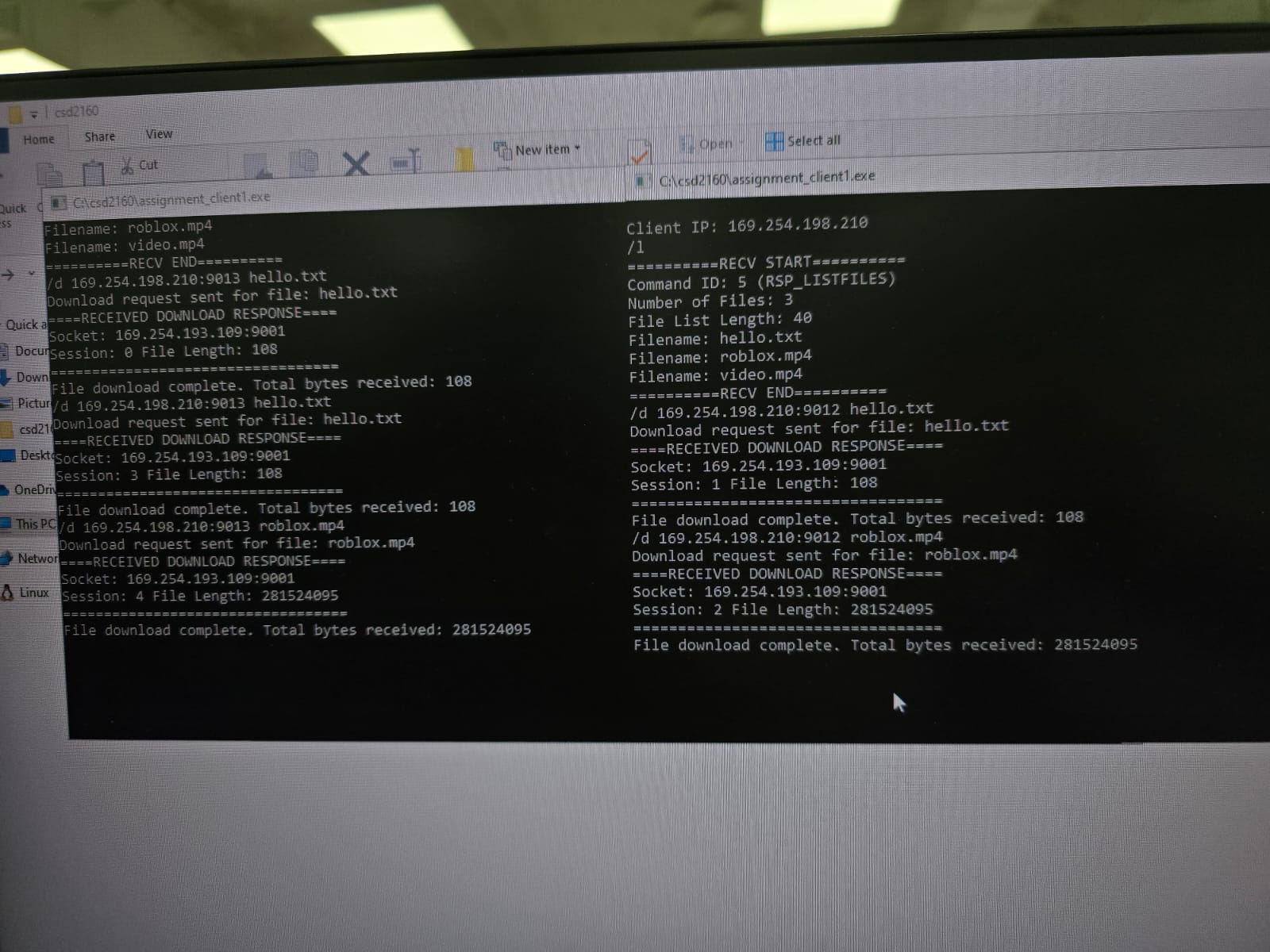
For more in-depth explanations and run-throughs of the tests, supplementary videos of testing the program via the ETHERNET are available [here](https://drive.google.com/drive/folders/1aQ-6zfOQZPfOqYQtNXUHi8roza25IW3u?usp=drive_link). (\*please reach out to us if the google drive folder cannot be accessed. )

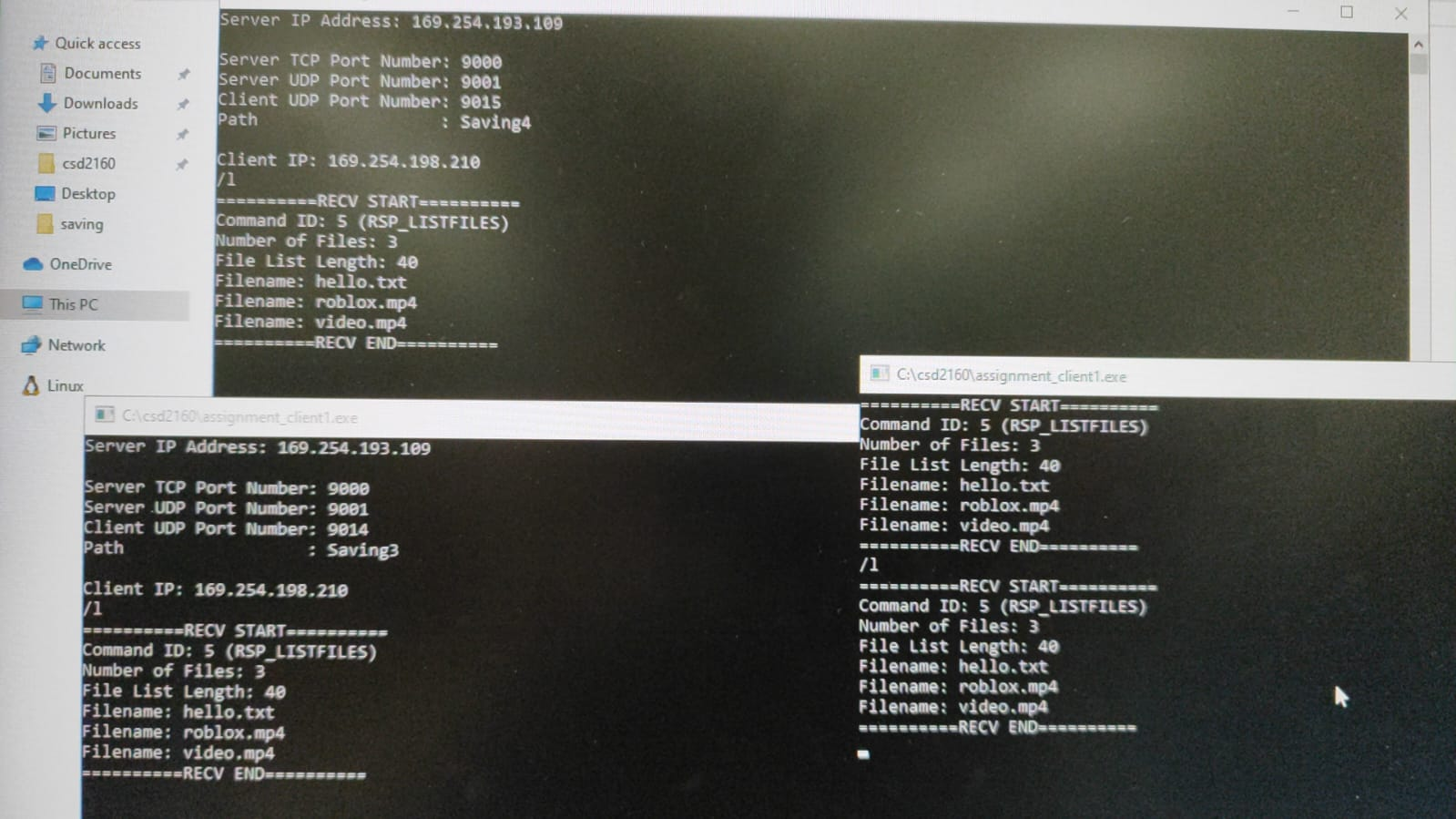
**Test Results: Downloading over Ethernet**

Testing scenario: DIT0843SG (Client) AND DIT0735SG (Server) via ethernet cable (ports 9000, 9001, 9012 and 9013)

Some lab computers may or may not be configured correctly as we had trouble with others.

Server1:  


Two Clients (UDP ports 9013 and 9012):  


Three Clients (UDP ports 9015, 9014, 9013):  


Manual disconnections tested + timeout + 3 clients.

Data integrity tested through a video playthough. In google drive link.

**Findings**

Compiling the test results from the three tests, we have found that the code:

* supports multi-threading (multiple clients) file downloading over Ethernet, Wifi, and Local Device.
* /l is consistent across all clients
* Able to download simultaneously for all 3 clients
* Able to complete download at the same time, stdin still able to do while downloading --> automatically sends req\_listfiles after download completed
* Able to download files > 200mb simultaneously per client (3 clients), suggests evidence of reliable data transfer, manual disconnection still ok

**Remarks**

1. Error message (Timeout occurred) pops up when download starts, this is not true as timeout is originally set to 0 when starting download. As it is not listed as a requirement (only as debug recommendation), we have decided it's ok to leave it as is.

Individual Contribution

Joel Jie Lee:

* Creation and Binding of UDP port for server and client
* Setting up of separate thread for UDP.
* RSP\_DOWNLOAD handling for Server side
* REQ\_DOWNLOAD handling for Server side
* DOWNLOAD\_ERROR sending for Server side.
* File Download process for server side: Map of file sessions, reading and writing to udp port, managing reliable data transfer etc.
* Optimization of program
* Testing on Lab machines
* Adding of comments and cleanup of project.

Tan Yee Tong:

* Did Checksum.hpp/.cpp for calculating and validating checksum.
* Did GetFileData in Utility.cpp
* Did the TCP protocol of the RSP\_DOWNLOAD (Client side)
* Did the GetDataFromFile function in the Ulitity.cpp
* Did TCP /d directive handling to send REQ\_DOWNLOAD (client)
* Did the receiving part of the UDP connection.
* Technical Document

Rachel Lim Jia En:

* Did /l directive handling (client and server)
* Did UDP /d directive handling (client)
* Did file download handling on client side
* Technical Document
* Testing on Lab machines