CSD2161/CSD2160/CS260/MET3302 2025 Spring *Assignment 3*

Program RTIS/METS Name _____

DigiPen Institute of Technology

Due Date:	16th March 2025
Topics covered:	Reliable data transfer over UDP
Grouping :	Group project: group size: 1 - 3
Deliverable:	Moodle submission: A compilable Visual Studio solution (Release mode), a README and a design document in a zipped file. In case there are multiple submissions per group, the lowest grades from these submission will be given.

This assignment requires an implement of ***file downloading over UDP*** .

1 Specifications

1. For a pair of client and a server program, (multiple instantiations of) clients will be able to connect to the server via *TCP* . Once connected, clients should be able to know from the server the list of files for downloading by sending the /l directive.

2. A client may request to download a file via *UDP* by the /d directive with a client's IP address and port number.

   (a) For example, "/d 192.168.0.98:9010 filelist.cpp" requests to download a file named `filelist.cpp` via UDP port number 9010 and Client's IP address is 192.168.0.98.

   (b) When this occurs, the server should notify the client the information with regarding to the file downloading (e.g. file length).

   (c) The server should then share the its IP address and UDP port number for the file downloading to the client (that is trying to download the file). Both the server and the client should know which port to be used for file downloading.

3. Apart from the file transfer, all messages are exchanged via TCP.

4. All files should be received correctly under the bad communication channel. Thus, certain reliable communication protocol should be implemented, such as stop-and-wait or Go-back-N or Selective Repeat.

5. Assumptions on the downloading files:

   (a) File size will be less than 200 MB.

   (b) The number of downloading files: ≤ 256.

   (c) Multiple files can be simultaneously transmitting to different clients.

   (d) Clients have enough hard disk to store the received files.

2 Detailed Requirements

A multi-threading server and a multi-threading client are to be designed.
Requirements are:

1. Server will prompt the info in Listing 1 at beginning.

```
        Server TCP port Number:
        Server UDP Port Number:
        Path                  :

```

Listing 1: Server Initial Prompt definition

   (a) Server TCP port Number : the server will listen on its IP address and this port, and display the received message on the terminal.

   (b) Server UDP Port Number : for file downloading to clients.

   (c) Path : the location of the download files. Only one path is needed.

2. A client should prompt the info in Listing 2 at beginning, then the client should connect to the server.

```
        Server IP Address     :
        Server TCP Port Number:
        Server UDP Port Number:
        Client UDP Port Number:
        Path                  :

```

Listing 2: Client Initial Prompt definition

   (a) Server IP Address and Server TCP Port Number : specified in the above step 1a;

   (b) Server UDP Port Number : the port that the downloading file to be sent to all the clients, specified in the above step 1b;

   (c) Client UDP Port Number : to receive the downloading files;

   (d) Path : path to store the received download

3. Once a client is connected to the server, a TCP message will be sent to the server when a client user types a directive and presses Enter (similar to Assignment 2).

4. TCP message format
   The TCP message format includes a mandatory Command ID field and several optional fields according to the associated Command ID.

   The Command ID [1] is 1 byte long, and indicates different commands detailed in Listing 3.

```
        enum CMDID {
          UNKNOWN       = (unsigned char)0x0,//not in use
          REQ_QUIT      = (unsigned char)0x1,
          REQ_DOWNLOAD  = (unsigned char)0x2,
          RSP_DOWNLOAD  = (unsigned char)0x3,
```

```
6           REQ_LISTFILES  = (unsigned char)0x4,
7           RSP_LISTFILES  = (unsigned char)0x5,
8           CMD_TEST       = (unsigned char)0x20,//not in use
9           DOWNLOAD_ERROR = (unsigned char)0x30
10        };
11
```

Listing 3: Command ID definition

The remaining fields can be in one of the following three variations according to the Command ID in the message.

(a) `REQ_DOWNLOAD` Command ID: There are four fields after the Command ID field, illustrated in Figure 1.

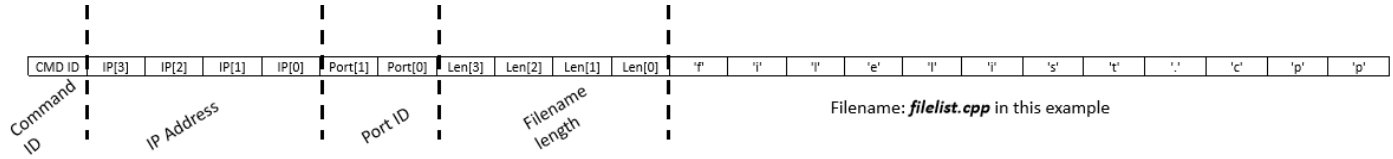

Figure 1: Message Format: REQ_DOWNLOAD

i. IP Address [4 bytes]: IP address of the client requesting file downloading.

ii. Port Number [2 bytes]: port number of the client requesting file downloading.

iii. Filename Length [4 bytes]: total length of the download filename in bytes.

iv. Filename [variable length, defined above]: name of the file, including path.

(b) `RSP_DOWNLOAD` Command ID: There are four fields after the Command ID field, illustrated in Figure 2.

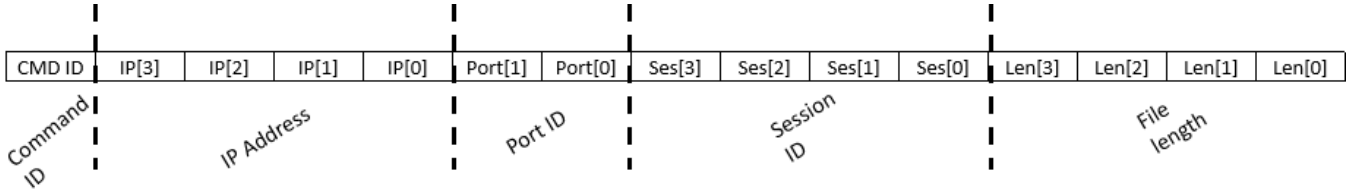

Figure 2: Message Format: RSP_DOWNLOAD

i. IP Address [4 bytes]: Server IP address.

ii. Port Number [2 bytes]: Server's port number for file downloading.

iii. Session ID [4 bytes]: ID for the file download session (each file downloading is considered a new session).

iv. File Length [4 bytes]: length of the file to be downloaded.

(c) `RSP_LISTFILES` Command ID: this command indicates the list of available files to be downloaded on Server. There are four fields after the Command ID field, illustrated in Figure 3, which shows two files are available for downloading: `1.jpg` and `2.mov`, and each byte content is give in Listing 4.

[1]Assume all the number (either 2 or 4 bytes) are all in network byte order: big-endian.

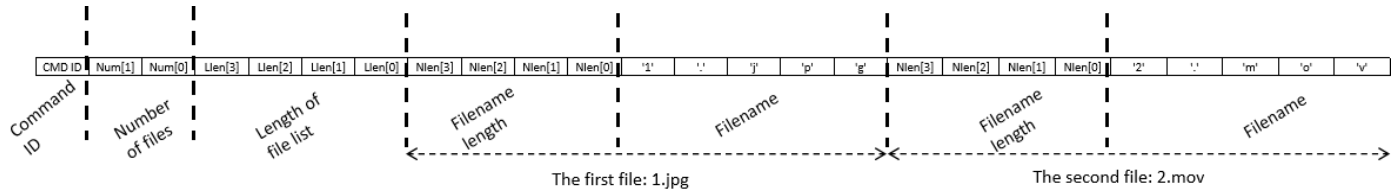

Figure 3: Message Format: RSP_LISTFILES

```
1          CMD ID                          0x05
2          Num[1]                          0x00
3          Num[0]                          0x02
4          Llen[3]                         0x00
5          Llen[2]                         0x00
6          Llen[1]                         0x00
7          Llen[0]                         0x12=decimal 18
8          Nlen[3]                         0x00
9          Nlen[2]                         0x00
10         Nlen[1]                         0x00
11         Nlen[0]                         0x05
12         Filename string byte[0]         '1'
13         Filename string byte[1]         '.'
14         Filename string byte[2]         'j'
15         Filename string byte[3]         'p'
16         Filename string byte[4]         'g'
17         Nlen[3]                         0x00
18         Nlen[2]                         0x00
19         Nlen[1]                         0x00
20         Nlen[0]                         0x05
21         Filename string byte[0]         '2'
22         Filename string byte[1]         '.'
23         Filename string byte[2]         'm'
24         Filename string byte[3]         'o'
25         Filename string byte[4]         'v'
26
```

Listing 4: Example content of RSP_LISTFILES Message

i. Number of Files [2 bytes]: number of files listed in this message;

ii. Length of File List [4 bytes]: length of the following **Filename Length** and **Filename** fields. Unit: bytes. If there are **N** files in the list, then there are **N** pair of the following fields:

   A. Filename Length [4 bytes]: length of the download filename & path. Unit: bytes.

   B. Filename [variable length, defined above]: the path and filename.

(d) `REQ_QUIT`, `REQ_LISTFILES` or `DOWNLOAD_ERROR` Command IDs: No other fields but Command Id.

Similar with the Assignment 2, each Command ID has a corresponding directive so that users can interact with the client. The Command ID and directive mapping is shown in Table 1.

In order to set and get the value in network byte order, you can use **htonl** or **htons** to convert the host byte order to network byte order at the sender side, and use **ntohl** or **ntohs** to convert the network byte order to host byte order at the receiver side.

Table 1: Command ID and Directive Mapping

Command ID	Directive	Comment
REQ_DOWNLOAD	/d	Client: requests downloading.
RSP_DOWNLOAD	N.A.	Reply from server: downloading info.
REQ_LISTFILES	/l	Client: requests the downloading file list.
RSP_LISTFILES	N.A.	Reply from server: list of available files
REQ_QUIT	/q	Client: requests to quit.
DOWNLOAD_ERROR	N.A.	Reply from server: downloading error!

5. UDP message format
   To be proposed, documented and implemented by each group.

6. The server should be able to serve multiple clients at the same time.

7. A possible of message transaction between the server and two clients (Client1 and Client2) is illustrated in Figure 4, and explained below:

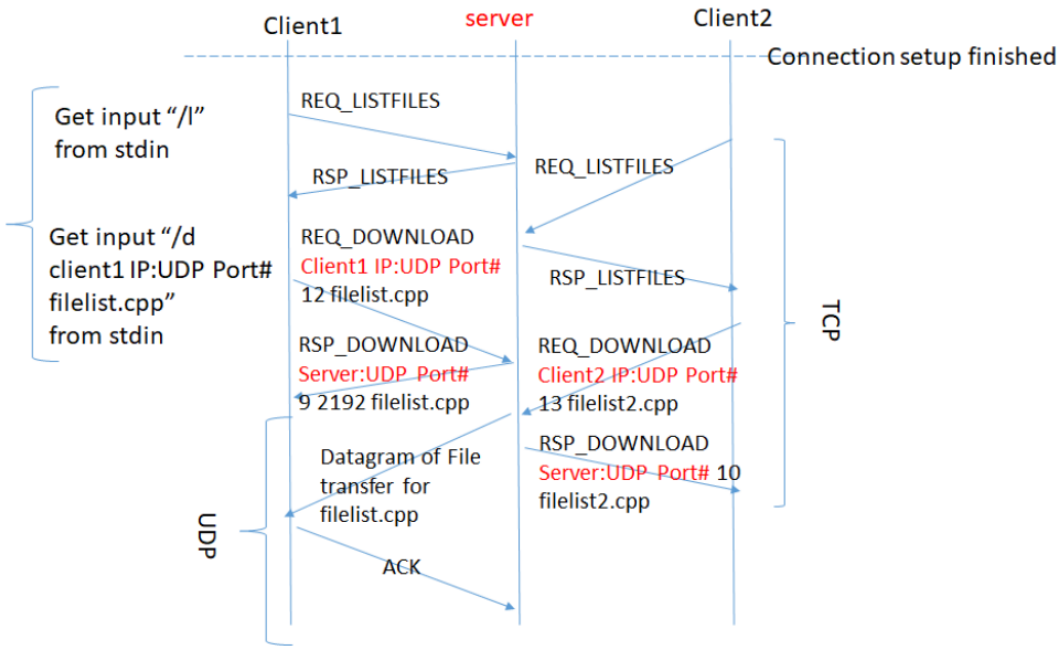

Figure 4: Transaction example: 2 clients

(a) Client1 types "/l" to request the list of available files. The client program should identify the /l directive and send a REQ_LISTFILES message to Server over TCP.

(b) Server replies the RSP_LISTFILES message.

(c) Client1 types "/d 192.168.0.98:9010 filelist.cpp", the client program should identify the /d directive and send REQ_DOWNLOAD message to Server over TCP: request to receive the file from IP address *192.168.0.98* and UDP port number *9010*.

(d) Server searches the file to be download. If the file exists, Server will respond with `RSP_DOWNLOAD` for downloading details. Otherwise, it sends `DOWNLOAD_ERROR` to Client1.

(e) Server starts the file transfer via the UDP port that is specified in `RSP_DOWNLOAD`.

(f) Client1 acknowledges the receiving of datagram for file download.

(g) Client2 can request file list and request downloading simultaneously.

8. Another possible of message transaction between Server and two clients (Client1 and Client2) is shown in Figure 5.
Note Server fails to search the file for Client2 in this example.

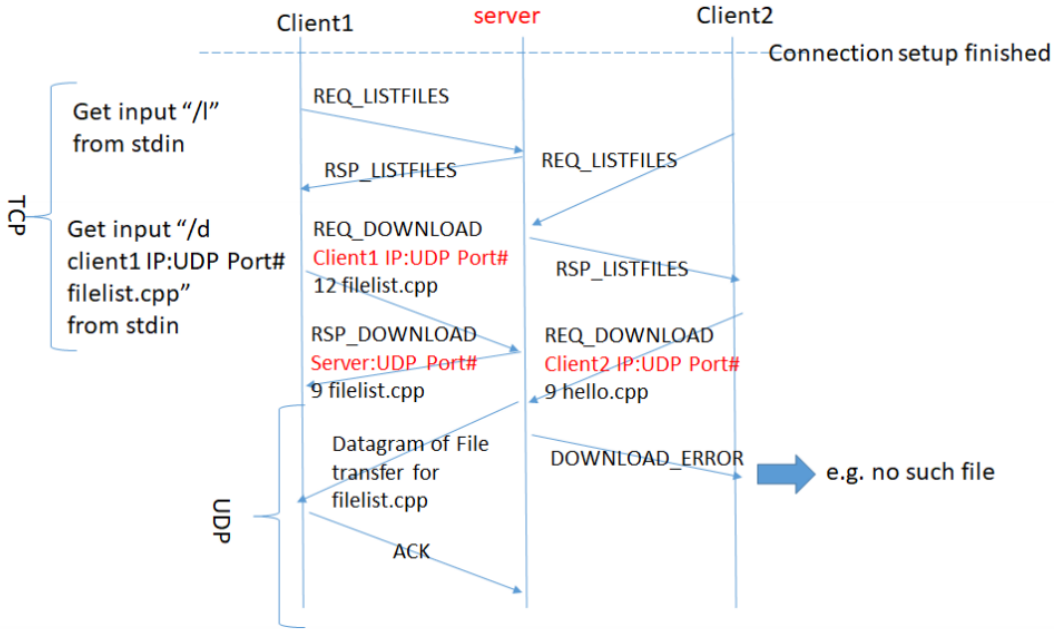

Figure 5: Transaction example: 2 clients with failure

3 Design Implementation

To support a reliable FTP over UDP, the following items are expected to be implemented.

1. Buffer for long packets.
When a receiver uses a fixed size buffer to receive a packet with an unknown size, this receiver may process multiple times to receive a complete packet.

2. Multi-threading.

   (a) Multi-threading @Server:
   The **pre-threading** model given by the provided `taskqueue.h` and `taskqueue.hpp` can be used.
   This model creates a pool of threads and queues sockets of the accepted TCP connections. Once a thread has obtained the socket, this thread will take charge of the data communication with the associated client.

If the server starts `N` threads, then there are maximum `N` clients that can connect to the server.

(b) Multi-threading @Client:
The `std::thread` method can be used for client.

- A client can create a thread (named `T_Commu`) to handle the receiving messages from the server and transmitting messages to the server.
- A main thread (named `T_Main`) accepts the input from stdin.
- Actions over stdin (such as typing) and message communication should not block each other.

3. Reliable data transfer over UDP.

(a) The communication channel is bad and there are packet losses.
To emulate bad communication channel, the wired communication is assumed between the clients and the server. An **Ethernet Break Circuit** shown in Figure 6 will be used in-between the connecting Ethernet cables, and cable disconnection will be achieved by pressing buttons on this circuit board.

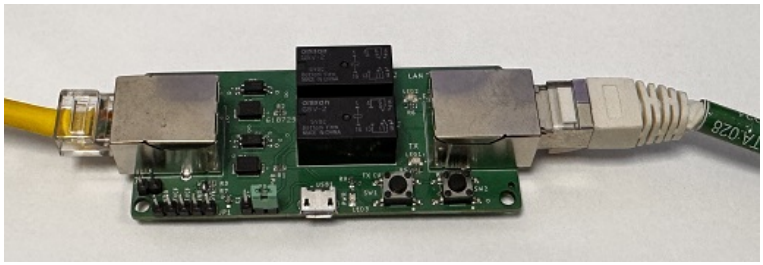

Figure 6: Ethernet Break Circuit

Assumptions:

- Each Ethernet breaking time should be less than 0.5 second.
- It is possible to have multiple Ethernet breaking incidents during a file transmission.
- There are limited set of **Ethernet Break Circuit** to be lent.

Recommendation: log the retransmission information (e.g. packet sequence number, the size of the packet) for debugging.

(b) To achieve reliable communication over UDP on bad communication channel, you **can** add the following fields on the UDP datagram:

i. Session ID [4 bytes]: the downloading session.
ii. File Length [4 bytes]: the download file length.
iii. File Offset [4 bytes]: the position in the download file as the starting byte in this message. This is needed when a file has to be split into multiple messages.
iv. File Data Length [4 bytes]: to indicate the length of file data in this message.
v. File Data [variable length, defined above]: data bytes of the file.

(c) To achieve reliable communication, you also **can** implement an ACK over UDP with the following fields:

i. Flags [1 byte]: LSB of Flags can be used to indicate whether this datagram is an ACK or file data.

ii. ACK Number [4 bytes]: to indicate the acknowledgement for receiving datagram with the sequence number equal to this ACK number.

iii. Sequence Number [4 bytes]: to indicate the sequence number of the datagram carrying file data.

(d) Other possible solutions are welcome.

4 Rubrics

1. The downloaded files should be correct.
2. A README: indicates how to run, including (if applies) the format of either the configuration file or the arguments for the program.
3. A design report (named `Assignment_3_Design_Report_SIT-ID.pdf`) with the following contents:

   (a) Design introduction: the customized reliable communication protocol over UDP,including message format, ACK scheme, and transaction sequence, and communication performance.

   (b) Verification: how did your group verify the design? Screenshots/photos, log files can be provided.

   (c) Individual contribution: including all members' full names, SIT IDs, and DigiPen IDs.

4. Your code should be properly commented with appropriate naming of the variable names. On top of that, you should demonstrate a re-use of code from assignment 2 if possible.
5. Your program should be able to list files consistently. For example both Client A and B are currently logged on. The received the list of files for downloading should be same from the both clients.
6. Your program should be able to support the transfer of files to more than 2 clients concurrently.
7. Show that you have implemented a reliable communication protocol, such as stop-and-wait or pipeline scheme (Go-back-N or Selective Repeat) or others. Define important and configurable parameters in a configuration file.
8. Stress test of transferring large files ($\geq$ 100MB).
9. Faster file downloading speed will get higher scores.
   During grading:

   - Same execution platform will be used

- the communication disruption patterns should be similar

The lecturer reserves the right to add extra test cases for grading. The list above is non-exhaustive. The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.

5 Submission Guideline

Submission procedures for programming assignments are detailed below. Failure to follow any of the following submission procedures and guidelines will result in a 10% deductions from your assignment grade.

1. The submission must be a compile-able C/C++ program under Visual Studio.
2. ANY Source, header, data, or README files submitted must start with the following header:

```
/* Start Header
*************************************************************/

/*! \file (e.g. main.cpp)

\author (e.g. Tang Liang, liang.tang)

\par (e.g. email: liang.tang\@digipen.edu)

\date (e.g. 28 Feb, 2025)

\brief Copyright (C) 20xx DigiPen Institute of Technology.

Reproduction or disclosure of this file or its contents without the prior
written consent of DigiPen Institute of Technology is prohibited. */

/* End Header
*************************************************************/

```

Listing 5: Command ID definition

- To submit your programming assignment, organize a folder consisting of ALL relevant source code (including source files, header files, data files), documentation files. In other words, your submission must be ready for compiling and linking by faculty.
- Naming convention: `<class>_<studentloginname>_<assignment#>`
  For example, if your login is foo and assignment 3 is being submitted, your folder would be named `csd2161_foo_3`.
- Zip this folder and name the resulting file using the following convention: `<class>_<studentloginname>_<assignment#>.zip`. For example, if your login is foo and you are submitting assignment 3, your zipped file would be named as: `csd2161_foo_3.zip`

- Next, upload your zip file to Moodle course account Assignment 3 submission link.
- Finally, perform a sanity check to determine if your programming submission follows the guidelines by downloading the previously uploaded zip file, unzipping it, then compiling, linking, and executing your submission.