Due Date:	As shown on the Moodle
Topics covered:	Reliable data transfer using UDP
Deliverables:	Submit a compilable Visual Studio project (Release mode, Visual Studio 2022) in a zipped file (CSD2161_yourid_3.zip, CSD2160_yourid_3.zip or CS260_yourid_3.zip), satisfying the conditions stipulated and outlined in the syllabus. In particular, the README should have a brief description of how the compiled executable is to be run i.e., what kind of arguments etc. The project should be submitted through the submission link on the Moodle.
Objectives:	To demonstrate an ability to design networking application for a simple server application that supports multiple connections at the same time. In addition, the student is expected to be able to perform reliable data transfer using UDP sockets.

1 Programming statement: UDP File Downloading.

For this assignment, you will implement file downloading over UDP.

1.1 Specifications

1. You will have a client program and a server program. Clients will be able to connect to the server via TCP. When connected, clients should be able to know from Server what are files for downloading. As a client, I can type something like "/l" and Server will show me every file that I can potentially download from the server.

2. Any client may request to download a file (e.g. filelist.cpp) by typing in the command e.g. "/d 192.168.0.98:9010 filelist.cpp". This means Client expects to receive the file via UDP port number 9010 and Client's ip address is 192.168.0.98. When this occurs, Server should notify Client the information with regarding to the file downloading (e.g. file length). Server should then indicate to Client that is trying to download the file the IP address and UDP port number of Server that offers the file downloading. Both Server and Client will know which port to send their data. The messaging other than the file transfer is via TCP.

3. For this assignment, on Client, you can prompt the following to obtain Server IP Address, Server TCP Port Number, Server UDP Port Number and Path to store files, before you enter a command to request Server to send the list of files for downloading, request to download or quit:

```
Server IP Address
Server TCP Port Number
Server UDP Port Number
Client UDP Port Number
Path to store files
```

Server UDP Port Number is the port that a UDP socket that transfers file is sending the datagram to **all the clients**, Client UDP Port Number is the port that a UDP socket that can accept file transfer is listening on, and the other settings should be self-explanatory.

4. All files must transfer correctly to receive full credit. Assume that I will not attempt to send a file larger than 100MB and that the computer receiving the file will always have enough hard drive space to store it.

5. The program does not need to transfer files at full speed over Ethernet (if you are testing on Digipen Lab PCs); it should be able to send files at a reasonable speed. Sending a 10MB file in about a minute is reasonable. Sending a 10MB file in 5 minutes is not reasonable. Also, assume that at any given time, I will attempt to transfer multiple files to different clients.

6. To ensure a reliable data transfer over UDP, you may implement stop-and-wait or a pipeline scheme (either Go-back-N or Selective Repeat (with windows size fixed to e.g. 10) to control the sliding window).

7. You should also simulate a random packet loss (e.g. packet loss rate is 1%) on both Server and Client to see the effect of packet loss. It is helpful when you turn on the logging mode, the retransmission information (e.g. Packet sequence number, the size of the packet) can be kept track of.

1.2 Design requirement

You will build a multi-threading server and a multi-threading client that can connect to it. Requirements are:

1. Server will prompt the user for a TCP port and will then start listening on its IP address and port, which it will display. It also prompt the user to key in a UDP port number that is used to transfer file to the clients. It prompts the user to key in the path/directory where the download files are. Server will wait for an incoming connection.

2. Client will start and prompt user for an IP address and TCP port number of Server. Client prompts the user to key in the Server UDP port on which Server sends the file data and also prompts the user for Client UDP port that receives the transferred file. Client prompts the user to key in the path/directory to store the received download file. Client will then have a means to connect to the server.

3. When the user/Client is connected to Server, if the user types a command and presses enter, it will send to the server a message with the defined format as described below (similar to Assignment 2).

4. The message format for TCP segment includes a few fields and is defined as follows[1]:

   (a) command id: 1 byte, used to indicate the command id, which could be defined as

[1]Assume all the number (either 2 or 4 bytes) are all in network byte order

```
enum CMDID {
UNKNOWN = (unsigned char)0x0,//not used
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,
CMD_TEST = (unsigned char)0x20,//not used
DOWNLOAD_ERROR = (unsigned char)0x30
};
```

(b) For command id: *REQ_DOWNLOAD*

i. IP address: 4 bytes. This is the IP address of Client requesting UDP file download.

ii. port number: 2 bytes. This is the port number of Client for UDP file downloading.

iii. file name length: 4 bytes. This is used to indicate the total length of the download file name in bytes.

iv. file name: the name of the file with the length specified by the above `file name length` field.

(c) For command id: *RSP_DOWNLOAD*

i. IP address: 4 bytes. This is the IP address of Server offering UDP file download.

ii. port number: 2 bytes. This is the port number of Server for UDP file downloading.

iii. Session ID: 4 bytes. This is used to indicate the ID for the file download session (each file downloading is considered a new session).

iv. file length: the length of the file for download.

(d) For command id: *RSP_LISTFILES*

i. number of files: 2 bytes. This is used to indicate the number of files listed in this message;

ii. length of file list: 4 bytes. This indicates the length of remaining bytes of the following fields of **file name length** and **file name** in the message. If the number of files in the list is N, then there are N pair of the following fields:

iii. file name length: 4 bytes. This is used to indicate the total length of the download file name in bytes.

iv. file name: the name of the file with the length specified by the above `file name length` field.

The message sent should follow the given format. If it is a command message type of *REQ_QUIT*, *REQ_LISTFILES* or *ECHO_ERROR*, no other fields but command id is required. In order to set and get the value in network byte order, you should 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.

5. **Optional in the case you want to use your own method to implement reliable data transfer over UDP.** The message format for UDP file data may include a few fields and be defined as follows:

   (a) Session ID: 4 bytes. This is used to indicate the downloading session.

   (b) flags: 1 byte. LSB of flags is used to indicate whether this datagram is an ACK or file data and LSB should be 0 for file data.

   (c) sequence number: 4 bytes. This is used to indicate the sequence number of the datagram carrying file data.

   (d) file length: 4 bytes. This is used to indicate the download file length.

   (e) file offset: 4 bytes. This is used to indicate the position at the download file for the first byte of file data in this message.

   (f) file data length: 4 bytes. This is used to indicate the length of file data in this message.

   (g) file data: total number of bytes for file data is specified in the above `file data length` field.

6. **Optional in the case you want to use your own method to implement reliable data transfer over UDP.** The message format for UDP ACK (indivisual ACK) datagram may include a few fields and be defined as follows:

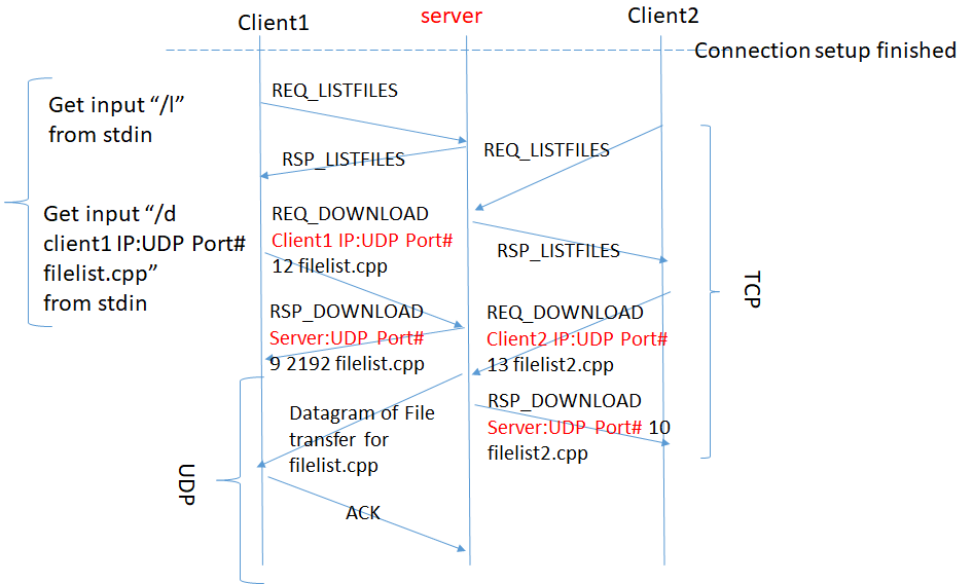
   (a) Session ID: 4 bytes. This is used to indicate the downloading session.

   (b) flags: 1 byte. LSB of flags is used to indicate whether this datagram is an ACK or file data. LSB should be 1 for ACK datagram.

   (c) ACK number: 4 bytes. This is used to indicate the acknowledgement for receiving datagram with the sequence number equal to this ACK number.

7. When Client types in "/q", Client quits. NOTE: It is necessary for Server to be able to accept multiple clients at the same time.

8. The sequence diagram below shows the example of messaging between the server and two clients (Client1 and Client2).



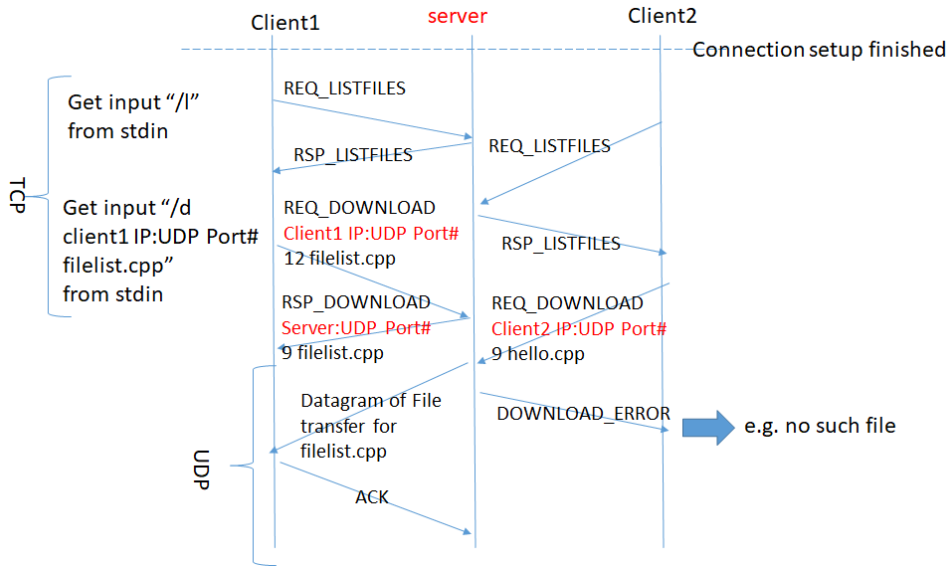
(a) If Client1 types in e.g. "/d 192.168.0.98:9010 filelist.cpp", the client program will identify the command "/d" and send to Server, intend to receive the file from IP address 192.168.0.98 and UDP port number 9010. Both Client1 and Client2 send the message to Server with the TCP connection.

(b) Server searches the file to be download. If the file exists, Server will respond with *RSP_DOWNLOAD*. Otherwise, it sends *DOWNLOAD_ERROR* to the client.

(c) Server starts the file transfer via the UDP port that is specified in *RSP_DOWNLOAD* command message to Client1.

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

9. The sequence diagram below shows the example of messaging between Server and two clients (Client1 and Client2). But Server fails to search the file for Client2.



(a) Please note that the receiver should process a few times to receive a complete TCP packet when it uses a fixed size buffer to receive a TCP packet with a unknown size.

(b) We assume you use `taskqueue.h` and `taskqueue.hpp` to provide multi-threading for server. You should use std::thread for your client.

i. The pre-threading model given by `taskqueue.h` and `taskqueue.hpp` can be used by the code for the server. It creates a pool of threads and queues the socket of the accepted TCP connection. The thread once obtains the socket from the queue and handles accordingly, which will take charge of the data communication with a client. In this thread function, the server will send and receive the message from the connecting client. If the server starts N threads, then there are maximum N clients that can connect to the server.

ii. The client creates a thread with the function to handle the receiving of the message from the server. If the client needs to respond, it will send the message to the server in this thread function. The main thread accepts the input from the stdin. If the input is a command to send a message, the client

will construct a message and send to the server. All the command message receiving will be handled by the thread function. **Keying in the input from the stdin should not block the receiving and sending to the server and vice verse**.

1.3 Rubrics

The whole assignment will be scored upon a total of 100 points.

(a) Your project/directories have been submitted according to the proper submission conventions as stipulated in the syllabus.

(b) A proper README file that indicates the format of either the configuration file or the arguments that the program takes in. If you are working in pairs/trios, please indicate your partner clearly in the README file too. Also, write out the division of labour between the two or three of you in the README file. .

(c) 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.

(d) Your program should be able to list files consistently. So say that now Client A and B are currently logged on. On Client A, I should be able to view the list of files for downloading, the same as on Client B and vice versa.

(e) Your program should be able to support the transfer of files to more than 2 clients. Say if we have three clients A, B and C logged on. We should be able to transfer a file to A, a file to B, a file to C concurrently. For testing of this stage, we would transfer files of a relative smaller size, say 1MB. The client programs and server program should not crash in anyway during the transfers. Any program crashes or incorrect transfer of files for this will result in deduction of points.

(f) Show the total number of packets sent and received during file transfer for the simulation of packet loss. Packet loss rate may be given in the configuration file.

(g) Show that you have implemented a stop-and-wait or pipeline scheme (Go-back-N or Selective Repeat) to ensure reliable data transfer. Use the parameter defined in the configuration file.

(h) Stress test of transferring large files. We shall test the downloading of binary files of size above 10MB. All transfers should be correct. On the LAN in school, the transfer should happen in a reasonably fast speed too. For example, the 100MB file transfer should be completed within TBD minutes time.

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.