Lab Assignment-3
Network Programming / UDP

In this lab assignment, you will implement a client-server data transfer application.
Communication protocols inherently involve concurrency, which can pose implementation challenges. This assignment provides hands-on experience with two different approaches to handling concurrency in protocol design:

- **Thread-based Concurrency:** In this approach, threads are used as the primary mechanism for concurrency. Each thread blocks while waiting on exactly one event source (e.g., keyboard input, or network input). Since you may already have encountered multithreading in previous courses, this method should feel relatively familiar.
- **Event-driven Concurrency (Non-blocking I/O / Asynchronous I/O):** Here, a single thread can listen to multiple event sources simultaneously. For example, the same thread may wait for either keyboard input or an incoming network packet. Depending on the programming language and libraries, this approach may still be combined with multithreading (e.g., Java NIO). At the other end of the spectrum, some environments (e.g., Node.js) are entirely single-threaded, which eliminates race conditions introduced by thread management.

Java has NIO, a non-blocking IO interface. Python supports non-blocking I/O via the asyncio library and frameworks like Twisted and Tornado. node.js is a javascript-based language designed around the event-loop model. Go uses goroutines, which allow for easy concurrency with minimal memory overhead and efficient non-blocking I/O.

To achieve the goals of this assignment, you must work in pairs. That is, there are two different developers involved. This assignment has distinct client and server sides, so there are two distinct pieces of code involved in a full solution. We take advantage of that by dividing the work in this way:

	Client	**Server**
Thread-based	Partner A	Partner B
Not thread-based	Partner B	Partner A

This schedule allows you to (a) experience both implementation approaches, (b) make progress independently of your partner (since your own client and your server should interoperate, even if implemented in different languages), and (c) experience having your code interoperate with

code written by someone else. When you're done, all four combinations of client and server choices should work together.

In this assignment, you will develop a custom ***UDP Application Protocol (UAP)***. UAP defines what messages are sent between the client and server, and how they are encoded.
UAP transfers lines of input from the client's *stdin* to the server, which then prints them on its *stdout*.

UAP message:

Protocol Headers
UAP is much more realistic than the protocols shown in class, in large part because it defines a message as a header plus data (rather than just data). Without a header, you can have only one kind of message, and so can't do simple things you almost certainly need to do, even if you don't realize it yet. (One example is returning an error indication, for instance, even though we don't do that in this assignment.) Protocols you design should always include headers in message encoding

Protocol Sessions
UAP supports the notion of a session. A session is a related sequence of messages coming from a single client. Sessions allow the server to maintain state about each individual client. For instance, the server could, in theory, print out how many messages it has received in each session, for instance, or it could maintain a shopping cart for each session. (We don't actually implement either of those.)
Unlike TCP (which has "connections"), UDP doesn't have any notion related to sessions, so we build them as part of our protocol.

Protocol Messages and Format
UAP defines four message types: **HELLO, DATA, ALIVE, and GOODBYE**. All message encodings include a header. The header is filled with binary values. The header bytes are the initial bytes of the message. They look like this, with fields sent in order from left to right:

magic	version	command	sequence number	Session id	Logical clock	Timestamp
16 bits	8 bits	8 bits	32 bits	32 bits	64 bits	64 bits

- ***magic*** is the value ***0xC461*** (i.e., decimal 50273, if taken as an unsigned 16-bit integer). An arriving packet whose first two bytes do not match the magic number is silently discarded.
- ***version*** is the value 1.
- ***command*** is **0 for HELLO, 1 for DATA, 2 for ALIVE, and 3 for GOODBYE**.

- ***sequence numbers*** in messages sent by the client are 0 for the first packet of the session, 1 for the next packet, 2 for the one after that, etc. Server sequence numbers simply count up each time the server sends something (in any session).
- ***session id*** is an arbitrary 32-bit integer. The client chooses its value at random when it starts. *Both the client and the server repeat the session id bits in all messages that are part of that session.*
- ***Logical clock:*** keep track of the sequence of events.

Multi-byte values are sent **big-endian** (which is often called "network byte order").
In **DATA** messages, the header is followed by arbitrary data; the other messages do not have any data. The receiver can determine the amount of data that was sent by subtracting the known header length from the length of the UDP packet, something the language/package you use will provide some way of obtaining.

Only one UAP message may be sent in a single UDP packet, and all UAP messages must fit in a single UDP packet. UAP itself does not define either maximum or minimum **DATA** payload sizes. It expects that all reasonable implementations will accept data payloads that are considerably larger than a typical single line of typed input.

UAP Message Processing

Server

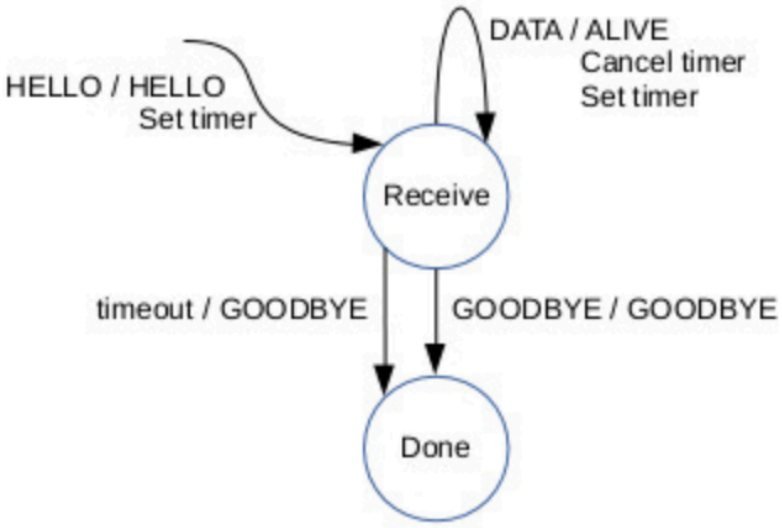
The server sits in a loop waiting for input from the network or from **stdin**. Execution of the server ends when either **end of file** is detected on stdin or the input line is ***"q"***. When the server quits, it sends **GOODBYE** messages to all sessions (described shortly) thought to be currently active.

When a new packet arrives, the **magic number and version** are checked. The packet is silently discarded unless those fields match the expected values.

Next, the server examines the **session id** field. If a session with that id has already been established, it hands the packet to that session. Otherwise, it creates a new session and hands the packet to it.

Server Session

Server sessions operate according to the following FSA diagram:



Here transition labels are of the form ***event / action(s)***, meaning "when an event of the specified type occurs while in the source state, take the actions specified and transition to the destination state." For instance, the transition ***HELLO / HELLO; Set timer*** means that when a **HELLO** message arrives, reply with a **HELLO** message and also set a timer that will raise an event at some later time (unless it is canceled), and then transition to state **Receive**.

Sessions are created (by the server) after receiving a message with a new session id. The newly created session checks that this initial message is a **HELLO**, and terminates if not. Because the client may never send a **GOODBYE**, sessions garbage collect themselves by setting an inactivity timer. If no message is received from the client for too long, a **GOODBYE** is sent and the session terminates.

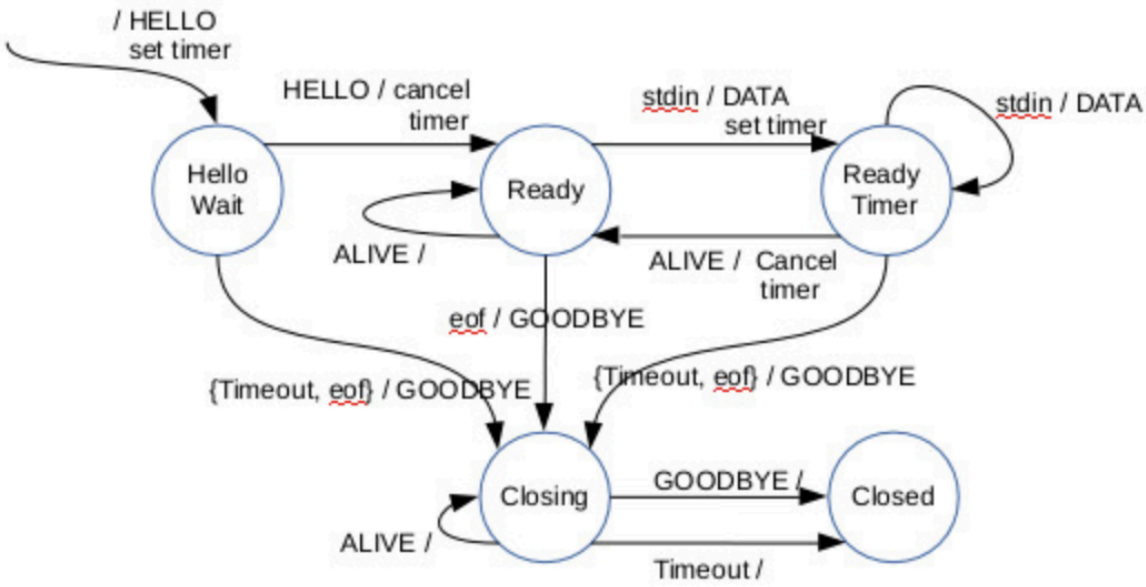
When a **DATA** message is received from the client, its data payload is printed to ***stdout***. To give the client a way to determine that the session is still up, the session sends an **ALIVE** message in response to each **DATA** message it receives.

The server session should keep track of the client sequence number it expects next. That is, if it has just processed a packet with sequence number 10, it should remember that the next sequence number expected is 11. If the next packet received has a sequence number greater than the expected number, a **"lost packet"** message should be printed for each missing sequence number. If the next packet's sequence number repeats the last received packet number, a **"duplicate packet"** message is printed and the packet is discarded. If the next packet's sequence number is less than the last packet's sequence number, we assume it is caused by a protocol error and the session closes: it sends a **GOODBYE** and transitions to **DONE**. (Sequence numbers "from the past" can be caused by the Internet delivering packets "out of order." That can occur, and more realistic protocols would want to deal with it more gracefully.)

If the server session receives a message for which there is no transition in its current state, it is a protocol error. In that case, it closes. For instance, receiving a **HELLO** while in **Receive** is an indication something is seriously wrong, and the only option is to close.

Client Behavior

The client follows the following FSA:



GOODBYE/ in any state transitions to state CLOSED

Basically, the client sends lines of input to the server. If no packets were ever lost, the client would receive a packet back for every one it sends: a **HELLO** in response to a sent **HELLO**, an **ALIVE** in response to a **DATA**, and a **GOODBYE** in response to a **GOODBYE**. If no response is received within a timeout, the client takes that as an indication the server is not running, and so the client terminates. (Note that this is not a very good assumption as the problem could just be a single dropped packet. We'll look at how to do a better job of guessing whether or not the server is really there later in the course.)

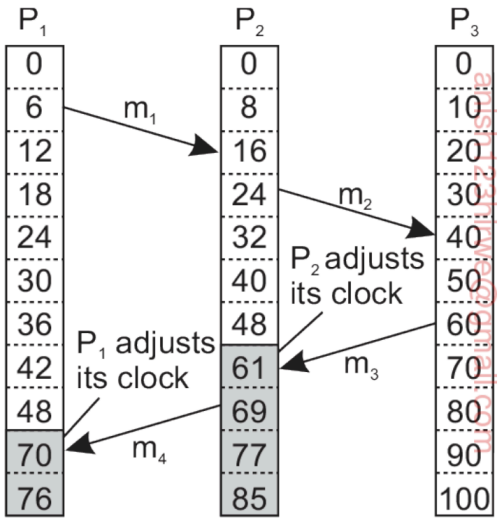
Session termination normally starts with the client, and involves a **GOODBYE** message in each direction. However, if the client receives a **GOODBYE**, it believes that the server is gone, no matter what the client's current state, and so it transitions immediately to the **Closed** state.

The client shuts down when it detects the end of file on **stdin** and the input is coming from a ***tty***, or when the input line is **'q'** and input is from a ***tty***. If **stdin** is connected to a file and ***eof*** is reached, the client should try to shut down after all outgoing messages have been put on the

network. (Whether or not you can do that might depend on the implementation language/package you're using.)

Logical Clock:
Server and client maintain a local counter. When a client or server performs an event, it increments its counter. When a client or server sends a message, it includes its counter value in the logical clock field of the message. When the receiver gets the message, it updates its own counter to be greater than the maximum of its current counter value and the received counter value, and then increments it before performing any new event.

The following figure illustrates the mechanism with three processes: P1, P2, and P3.



The following events update the counter

– All the send/receive events
– Timeout events
– STDIN events
– EOF events

One-Way Latency:

- Client/server puts send **Timestamp** (T_0) in the header.
- Client/Server receives at T_1.
- Latency = $T_1 - T_0$.
- Print one-way latency
- Estimate average-one way latency
- Needs clocks of client and server to be synchronized **(e.g., via NTP).** Otherwise, the result is skewed by clock drift.

Operational Details

To make it easier, or maybe even possible, to run your implementations we have to have a standard way to invoke your code and to process its output. You must conform exactly to the specifications shown here, for both invocation and output.

Invocation

Because how you launch a program can depend on its implementation language, you may need to write a shell script to conform to these instructions. Whatever you do, when we download your files and issue the commands shown below, what is described in the instructions should happen.

- Server: **$ ./server <portnum>**
  <portnum> is the port number the server should bind to.
- Client: **$ ./client <hostname> <portnum>**
  <hostname> and <portnum> give the location of the server. The hostname can be an IPv4 address (e.g., 128.208.1.137).

Output: Basic Typed Input

Suppose you run two client instances, back to back, like this:

$ ./client localhost 1234
one
two
three
eof

$ ./client localhost 1234
foo
bar
eof

Here **eof** is printed by the client program to indicate that end of file has occurred on stdin. The other lines are what the user typed.

The server's output should look like this:

$ ./server 1234
Waiting on port 1234...
0x736f0b1f [0] Session created
0x736f0b1f [1] one
0x736f0b1f [2] two
0x736f0b1f [3] three
0x736f0b1f [4] GOODBYE from client.
0x736f0b1f Session closed
0x545537a9 [0] Session created
0x545537a9 [1] foo
0x545537a9 [2] bar
0x545537a9 [3] GOODBYE from client.
0x545537a9 Session closed

The first value on each interesting line is the session id. The number in square brackets is the sequence number of the packet that caused this output line.

Output: Input Redirected from File

Redirecting stdin to read from a file makes it possible to offer so much input so fast that packets are lost. For example:

$ ./client localhost 1234 <Dostoyevsky.txt
eof

Note: find Dostoyevsky.txt file in the shared drive.

produced this server output:

Waiting on port 1234...
0x149780c3 [0] Session created
0x149780c3 [1] The Project Gutenberg EBook of The Brothers Karamazov by Fyodor
0x149780c3 [2] Dostoyevsky
0x149780c3 [3]
0x149780c3 [4]
0x149780c3 [5]
0x149780c3 [6] This eBook is for the use of anyone anywhere at no cost and with almost no
0x149780c3 [7] restrictions whatsoever. You may copy it, give it away or re-use it under

0x149780c3 [8] the terms of the Project Gutenberg License included with this eBook or
0x149780c3 [9] online at http://www.gutenberg.org/license
0x149780c3 [10]
0x149780c3 [11]
0x149780c3 [12]
0x149780c3 [13] Title: The Brothers Karamazov
0x149780c3 [14]
0x149780c3 [15] Author: Fyodor Dostoyevsky
0x149780c3 [16]
...eliding many lines...
0x149780c3 [456] unhappy young woman, kept in terror from her childhood, fell into that
0x149780c3 [457] kind of nervous disease which is most frequently found in peasant women
0x149780c3 [458] who are said to be "possessed by devils." At times after terrible fits of
0x149780c3 [459] hysterics she even lost her reason. Yet she bore Fyodor Pavlovitch two
0x149780c3 [460] sons, Ivan and Alexey, the eldest in the first year of marriage and the
0x149780c3 [461] Lost packet!
0x149780c3 [462] Lost packet!
0x149780c3 [463] Lost packet!
0x149780c3 [464] Lost packet!
0x149780c3 [465] Lost packet!
0x149780c3 [466] looked after by the same Grigory and lived in his cottage, where they were
0x149780c3 [467] Lost packet!
0x149780c3 [468] Lost packet!
0x149780c3 [469] Lost packet!
...eliding many lines…

Exactly which packets are dropped is non-deterministic. Your output should match the above, except for differences caused by the non-determinism.

Output: Concurrent Clients

You can cause two clients to be concurrently active with a single server using a shell script like this one:

```
#!/bin/bash
./client localhost 1234 <Dostoyevsky.txt >dual-c1.out 2>&1 &
./client localhost 1234 <Dostoyevsky.txt >dual-c2.out 2>&1 &
```

Start the server, redirect its output to some file, and then execute the script.
Doing that should produce server output that shows the two clients are running concurrently, as in this example output file (File uploaded in shared drive). This output is also non-deterministic.

Turn-in

You should turn in a **.tar.gz** file that, when unpacked, creates the directory structure shown here: