# CSE 434 Computer Networks (Fall 2019) Assignment 3

Duo Lu <duolu@asu.edu>

The due date of this assignment is 11:59 pm on October 30, 2019 (**firm deadline**). Late submissions will still be accepted. There will be a 0.5 point late penalty for every day later than the deadline. For example, if you submit it before midnight on Friday, 1 point will be deducted. This is a programming assignment, which is designed to allow you to gain hands-on experience socket programming on Linux. This can easily take more than 10 hours if you are not familiar with C or C++ programming on Linux. **Note that there will not be due date extension any more.** As you may have learned from the previous two assignments, the assignment requires some effort. Hence, please start early and search online from time to time to make sure that you understand how to write C or C++ code using the socket API. The grading is effort-based. Note that in this assignment you are allowed to reuse and augment the code posted on Canvas in assignment 1 and those code in this document. No need to write everything from scratch. You will continue working on it to make it an online Twitter-like application in the next assignment. The next assignment is built upon this assignment, and starting from this assignment, our TA, Mr. Sabur, will be in charge of the assignments and answer questions.

**I have added some hints in the template code for the TCP part.**

Here is a video to help you understand socket programming on Linux:

https://www.youtube.com/watch?v=LtXEMwSG5-8&t=0s&list=PLy9bJILQPYxaaNXRcwD4HyY-tm3JZtA5c&index=2

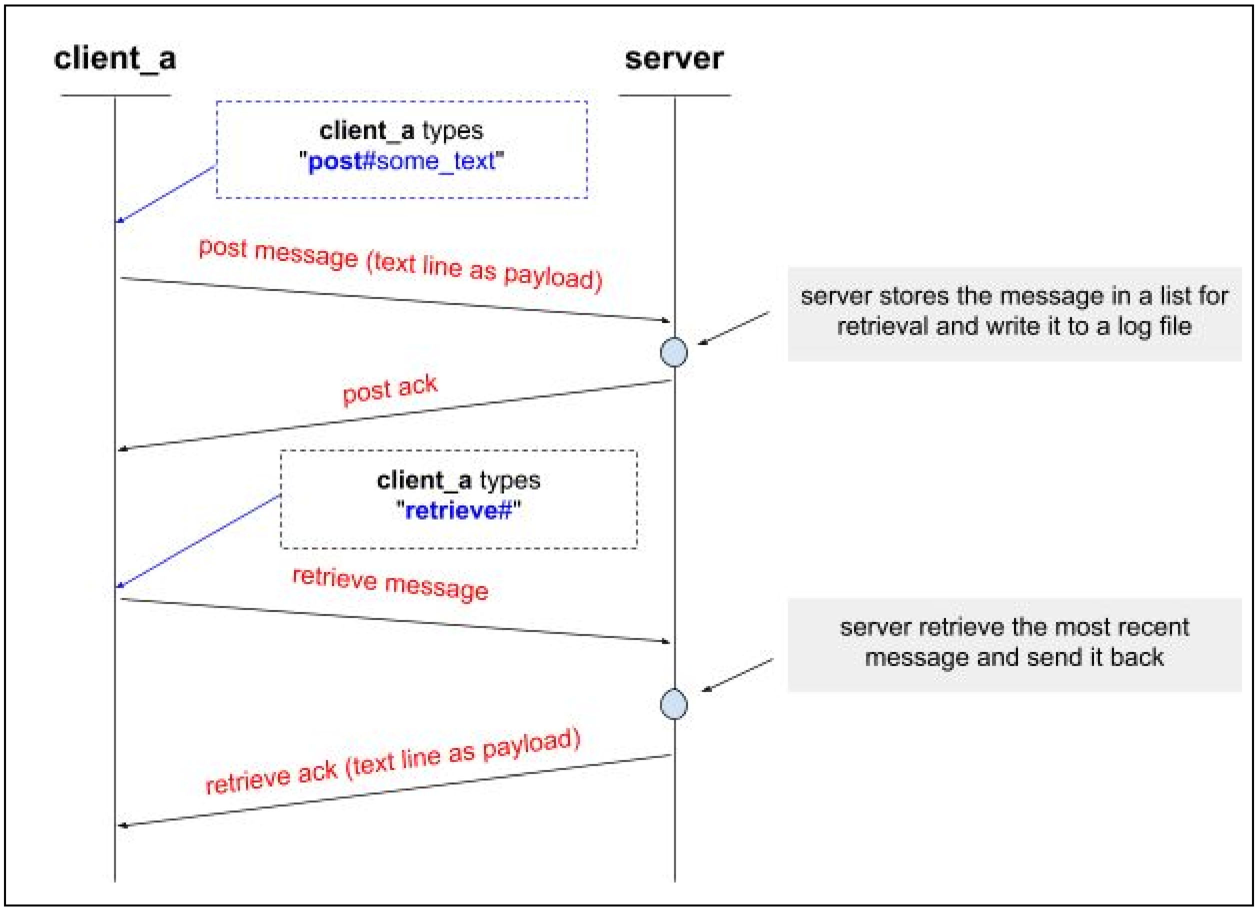
This video is mainly about TCP socket. UDP socket is similar but simpler.

**Task 1: UDP based message sending and receiving.**

Remember the "UDP send" and "UDP receive" code you tried to run in the first assignment? Now you are going to write a program based on these four pieces of code with the following requirements. In part one we just lay out the basic structures.

1) You will write a client-server application in C or C++ running on Linux.
2) The client and the server can run on the same machine or on different machines (Linux).

3) You can interact with the client machine over the command line, i.e., the program reads the line you typed on the command line, like that in "UDP send". You can also write a GUI, but I will not recommend you to spend time on that right now unless you are very familiar with GUI related coding in C or C++. GUI will be an extra credit extension in the next assignment.
4) On the client side, you can type a line "post#some_text" to post a message on the server. The "some_text" part can be any text message, like whatever you text in iMessage, SnapChat, Twitter, etc. The input line is checked and sent by the client after you hit the enter (see more detail in item 6). Note that whatever the user has typed, it must follow that format, i.e., it always starts with "post", then a "#", then some text, and the text contains only one line which can not exceed 200 characters (ASCII characters only). If it does not follow this format, the client will just print out "Error: Unrecognized command format" and the client will not send out anything through the network. The "#" is an indicator of the beginning of the actual text that needs to be sent.
5) On the client side, you can type a line "retrieve#" to ask the client to retrieve the most recently posted text (just one line) from the server. Currently, let's only retrieve one line, i.e., the most recent line, and in the next assignment we will expand this retrieving function. Similarly, the typed text must follow this format, i.e., exactly the string "retrieve" with a following "#", and nothing more. If it does not follow this format, the client will just print out "Error: Unrecognized command format" and the client will not send out anything through the network.
6) Once a line like "post#some_text" is typed, the client will send a UDP datagram to the server in a custom designed protocol. We call this datagram between the client and the server a message. Once the server receives a "post" message, it sends back another message as an acknowledgement of the post to the client. The server works like the "UDP receive". On the client side, if the "acknowledgement of post" message is received, the client program prints a line of "post_ack#successful".
7) Similarly, Once the server receives a "retrieve" message, it sends back another message as an acknowledgement of retrieve to the client which contains the retrieved line of text. Similarly, on the client side, if the "acknowledgement of retrieve" message is received, the client program prints "retrieve_ack#retrieved_text". Here "retrieved_text" means the actual text retrieved.
8) The server maintains a file as a log of all messages, including both the received message and the sent messages. For example, after a message is received or before a message is sent, you can just append a line to the log file. This is for debugging and the server does not need to print anything on the screen since usually a server machine does not have a monitor. For example, if a client sends a post message with "hello!" in the payload text, the server can append the following line (substitute the actual IP address and port number) to the file: "<timestamp> [client_ip:client_port] post#hello!". You can design your own log file format but it should contain necessary information about the time, the client, the message content.
9) A message sequence diagram is provided as follows.

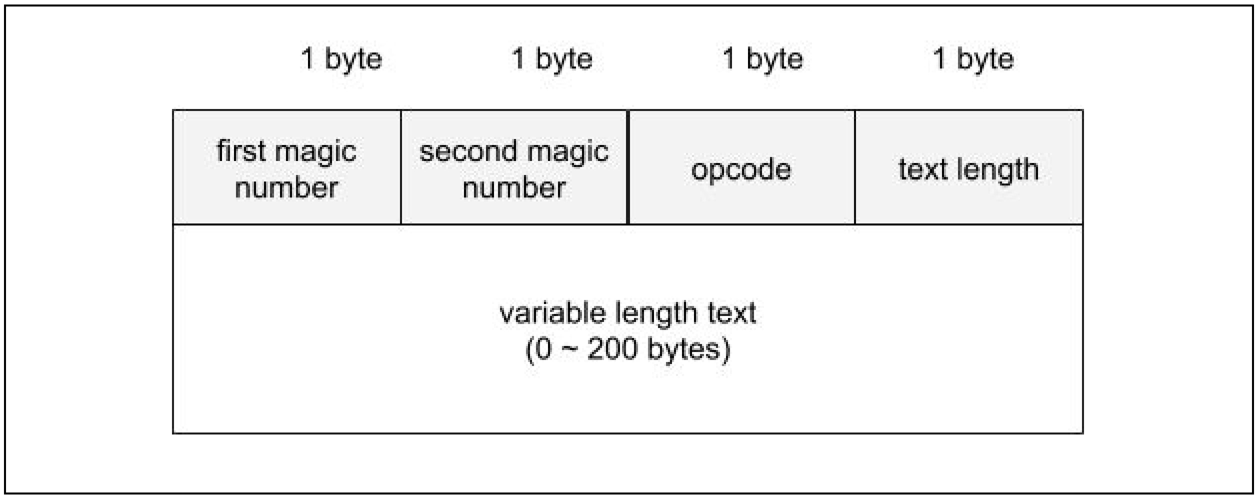

client_a
server
client_a types "post#some_text"
post message (text line as payload)
server stores the message in a list for retrieval and write it to a log file
post ack
client_a types "retrieve#"
retrieve message
server retrieve the most recent message and send it back
retrieve ack (text line as payload)

10) You can design the message format as you like. I recommend the following binary format if you don't have a specific idea.

- Fixed 4-byte header.
- The first byte is always the ASCII character of your first name initial, and the second byte is always the ASCII character of your last name initial. This kind of message field is typically called "magic number" to make the communication party to easily distinguish valid messages from spurious messages from another program or message with error. Also, this makes your protocol unique to some extent.
- The third byte is called the "opcode", i.e., operation code, which indicates what the message receiver should handle this message. We have two different types of operations, i.e., "post" and "retrieve". Let's specify that if the message means "posting", this opcode byte is 1 (i.e., 0x01); if the message means an acknowledgement of "posting", this opcode byte is 2 (i.e., 0x02); if the message means "retrieving", this opcode byte is 3 (i.e., 0x03); if the message means an acknowledgement of "retrieving", this opcode byte is 4 (i.e., 0x04),

- The fourth byte is the length of the actual text that needs to be transmitted, in byte. Since the text can be short or long, we need the length to delimit the message boundary.
- After the header, there is the actual variable length text that the user typed. We assume the user can only type ASCII, and it is 200 characters (i.e., 200 bytes) at most.

The format of the "post" message and the acknowledgement of retrieving is shown as follows. The acknowledgement of posting and the "retrieve" message have a similar format as the "post" message, but the opcodes are different, and they do not have any payload text (text length field is also zero). This type of message is also called "header only message".



The client constructs this message, and sends it through a UDP socket to the server. For example, If I type "post#hello!", the following byte stream will be sent

- The first byte is 0x44, i.e., ASCII character "D".
- The second byte is 0x4C, i.e., ASCII character "L".
- The third byte is 0x01, i.e., a "post" message.
- The fourth byte is 0x07, i.e., the line of text posted has seven bytes.
- The next seven bytes are ASCII string "hello!" with a new line character "\n" at the end. Note that "hello!" contains six bytes, and the new line character is one more byte, in total seven bytes. The text payload always ends with a new line character.

11) The server's IP address and port number can be hardcoded in the code, like that in "UDP send".

12) Please put all your code for the client side in a single file, name it "udp_client.c" or "udp_client.cpp". Similarly, please put all your code for the server side in a single file, name it "udp_server.c" or "udp_server.cpp".

There are a few important things to be noticed beforehand to avoid wasting time in coding and debugging.

1. The client sends out a message and receives the acknowledgement. When it sends out the message, the destination IP and port number are the IP of the server a port number that the server keeps receiving from (i.e., the "servaddr" structure in the code provided below, which is manually set to IP address "127.0.0.1" and port number "32000" in the example code in assignment 1 if you are running the server on the same machine). Note that this address hard coded on the client side must be the same address that the server program binds to and listens to. Usually, you don't need to care about the source IP and port on the client side. The operating system kernel does the job of choosing the source IP and port for you.
2. However, when the server receives the message, it can know the source IP and port from an argument of the recvfrom() function (i.e., the "cliaddr" structure in the code provided below). The source IP and port are on the client side. If both the client and the server are on the same machine, the source IP and the destination IP are the same, i.e., something like 127.0.0.1. Only the port numbers are different. Once the server receives the message and obtains the source IP and port number, it sends back the acknowledgement to this IP and this port. This means the server will call the "sendto()" function with a destination of the "cliaddr" structure.
3. You need to search online and understand what those functions do and how to use them, such as socket(), bind(), sendto(), recvfrom(), etc. Note that the fgets() function merely reads a line from the keyboard. If you don't press the enter button, fgets() will not return.
4. Please do remember to call fflush() after writing a line to the file. Otherwise, if the file not properly closed by fclose(), unflushed lines will not appear in the file. Note that those unflushed lines may contain a lot and even some cases the entire log file can be left blank if it is not properly closed.
5. Note that you DO NOT need to use multiple threads to handle multiple clients because it is connectionless. You can do it with a single loop.

Once you finish the programming, test it on Linux using the same environment that you used for assignment 1. Optionally, you can set up multiple instances of virtual machines and put the server and clients in different virtual machines to test it. You need to search online and figure out how to set up the virtual network to allow different virtual machines to communicate with each other. Multiple virtual machines will also require more memory resources on your physical machine. Also, you need to change the hardcoded server IP address in the program accordingly. Hence, it is recommended to test everything inside one virtual machine first.

Deliverables:

1. Please record a short video (less than three minutes, usually one minute should be enough and you can edit the video to make it succinct) demonstrating the functions of both your client program and your server program, including posting, posting ack, retrieving, retrieving ack, and logging messages on the server. While demo your program, please also open Wireshark in the background and capture the traffic generated by your program. You can apply a filter to only show those traffic about your program by specifying the IP address or port number that hardcoded in your program. Please show the message header and payload using the bottom window section of wireshark and make sure the message follows the format of your design. You also need to demonstrate the log file at the end (just open it and show the content, make sure it agrees with the messages transmitted). It is recommended to open two terminals and run the client on one side and run the server on the other side. You can record the desktop screen (just install an application called "Kazam" on Ubuntu or something else with a similar function), or you can just record a video using your smartphone. Please narrate while recording by either adding text on the video (you can just open the textpad and type) or speak (make sure the speaker is connected to the virtual machine if you run Linux inside a virtual machine, and you may need to configure it on VirtualBox accordingly).
2. Please upload the video to your Google Drive, and create a shareable link so that anyone with the link can see the video. You just need to submit this link on Canvas, not the whole video file. **Note that you need to grant permission to access the video file such that everyone with the link can see it. You can also upload it onto Youtube or some other online video sharing site and submit a link as an alternative to Google Drive shareable link.**
3. Please also submit your code on Canvas.

Grading rubrics:

There is 1 point for the demonstration of each message type including post, post_ack, retrieve, retrieve_ack. There are 2 points for the demonstration of server log file. In total this task has 6 points.

Here is the "**UDP client**" code. Be careful when you copy and paste it! Some of code snappit is not pure ASCII, like the double quotation mark, etc. You may need to copy it to the Ubuntu Linux text editor first and then copy it to your coding IDE next. I have added more detailed example code highlighted in red. **Remember that this is just an exercise of socket programming. Please do not over-engineer this project.**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {

    int ret;
    int sockfd;
    struct sockaddr_in servaddr;
    char send_buffer[1024];
    char user_input[1024];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
      printf("socket() error: %s.\n", strerror(errno));
        return -1;
    }

    // The "servaddr" is the server's address and port number,
    // i.e, the destination address if the client needs to send something.
    // Note that this "servaddr" must match with the address in the
    // UDP server code.
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(32000);

    // TODO: You may declare a local address here.
    // You may also need to bind the socket to a local address and a port
    // number so that you can receive the acks from the socket.
    // You may also skip the binding process. In this case, every time you
    // call sendto(), the source port may be different.

    // Optionally, you can call connect() to bind the socket to a
    // destination address and port number. Since UDP is connection less,
    // the connect() only set up parameters of the socket, no actual
    // datagram is sent. After that, you can call send() and recv() instead
    // of sendto() and recvfrom(). However, people usually do not do this
    // for a UDP based application layer protocol.
```

```
while (1) {

    // The fgets() function read a line from the keyboard (i.e, stdin)
    // to the "send_buffer".
    fgets(user_input,
          sizeof(user_input),
          stdin);

    // m is a variable that temporarily holds the length of the text
    // line typed by the user (not counting the "post#" or "retrieve#".
    int m = 0;

    // Compare the first five characters, check input format.
    // Note that strncmp() is case sensitive.
    if (strncmp(user_input, "post#", 5) == 0) {

        // Now we know it is a post message that should be sent.
        // Extract the input text line length, and copy the line to
        // the payload part of the message in the send_buffer. Note
        // that the first four bytes are the header, so when we
        // copy the input line of text to the destination memory
        // buffer, i.e., the send_buffer + 4, there is an offset of
        // four bytes after the memory buffer that holds the whole
        // message.

        // Note that in C and C++, array and pointer are interchangable.

        // TODO: Check the user input format and make sure it is not
        // empty or longer than 200 characters.

        m = strlen(user_input) - 5;
        memcpy(send_buffer + 4, user_input + 5, m);

        send_buffer[0] = MAGIC_1; // These are constants you defined.
        send_buffer[1] = MAGIC_2;
        send_buffer[2] = OPCODE_POST;
        send_buffer[3] = m;

    } else if (......) {

        // TODO: Check whether it matches to "retrieve#".
        // Note that a retrieve message has no payload, i.e., m = 0.

    } else {

        // If it does not match any known command, just skip this
        // iteration and print out an error message.

        continue;

    }
```

```
        // The sendto() function send the designated number of bytes in the
        // "send_buffer" to the destination address.
        ret = sendto(sockfd,                    // the socket file descriptor
               send_buffer,                     // the sending buffer
               m + 4, // the number of bytes you want to send
               0,
               (struct sockaddr *) &servaddr, // destination address
               sizeof(servaddr));             // size of the address

        if (ret <= 0) {
            printf("sendto() error: %s.\n", strerror(errno));
            return -1;
        }

        // TODO: You are supposed to call the recvfrom() function here.
        // The client will receive the acknowledgement from the server.

    }

    return 0;
}
```

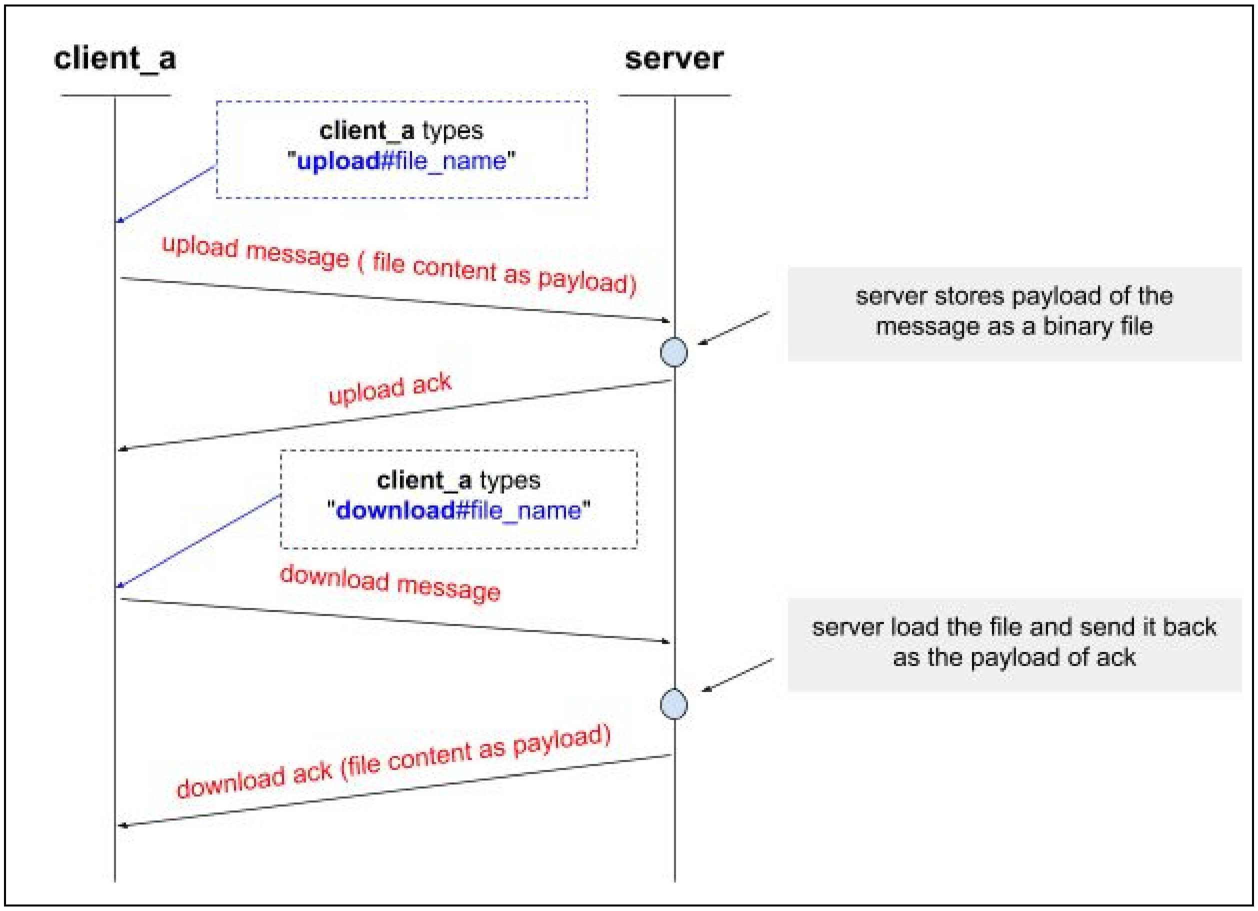
Here is the "**UDP server**" code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {

    int ret;
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;
    char recv_buffer[1024];
    int recv_len;
    socklen_t len;

    // This is a memory buffer to hold the message for retrieval.
    // I just declare a static character array for simplicity. You can
    // use a linked list or std::vector<...>. My array is only big enough
    // to hold one message, which is the most recent one.
    // Note that one more character is needed to hold the null-terminator
    // of a C string with a length of 200, i.e., strlen(msg) == 200).
    char recent_msg[201];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        printf("socket() error: %s.\n", strerror(errno));
        return -1;
    }

    // The servaddr is the address and port number that the server will
    // keep receiving from.
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(32000);

    bind(sockfd,
          (struct sockaddr *) &servaddr,
          sizeof(servaddr));

    while (1) {
        len = sizeof(cliaddr);
        recv_len = recvfrom(sockfd, // socket file descriptor
                  recv_buffer,       // receive buffer
                  sizeof(recv_buffer),  // max number of bytes to be received
                  0,
                  (struct sockaddr *) &cliaddr,  // client address
                  &len);             // length of client address structure

        if (recv_len <= 0) {
            printf("recvfrom() error: %s.\n", strerror(errno));
            return -1;
        }

        // TODO: check whether the recv_buffer contains a proper header
        // at the beginning. If it does not have a proper header, just
        // ignore the datagram. Note that a datagram can be up to 64 KB,
        // and hence, the input parameter of recvfrom() on the max number
        // of bytes to be received should be the maximum message size in
        // the protocol we designed, i.e., size of the header (4 bytes) +
        // the size of the maximum payload (200 bytes). For the same reason,
        // the receiver buffer should be at least this big.

        // One more thing about datagram is that one datagram is a one
        // unit of transport in UDP, and hence, you either receive the whole
        // datagram or nothing. If you provide a size parameter less than
        // the actual size of the received datagram to recvfrom(), the OS
        // kernel will just truncates the datagram, fills your buffer with
        // the beginning part of the datagram and silently throws away the
        // remaining part. As a result, you can not call recvfrom() twice
        // to expect to receive the header first and then the message
        // payload next in UDP. That can only work with TCP.

        // If the received message has the correct format, send back an ack
        // of the proper type.

        if (recv_buffer[0] != MAGIC_1 || recv_buffer[1] != MAGIC_2) {
            // Bad message!!! Skip this iteration.
            continue;

        } else {

            if (recv_buffer[2] == OPCODE_POST) {

                // Note that you need to erase the memory to store the most
                // recent message first. C string is always terminated by
                // a '\0', but when we send the line, we did not send
                // this null-terminator.
                memset(recent_msg, 0, sizeof(recent_msg));

                // Note that recv_buffer[3] contains the length of the text
                // line, see the protocol description.
                // Be careful, you may need to do some sanity checks on
                // recv_buffer[3], i.e., whether it is non-zero, etc.
                memcpy(recent_msg, recv_buffer + 4, recv_buffer[3]);

                // TODO: Now you need to assemble an ack, like what you did
                // in the client code.

            } if (......) {

                // TODO: Handle the retrieve message.

            } else {

                // Wrong message format. Skip this iteration.
                continue;
            }

        }

        // You are supposed to call the sendto() function here to send back
        // the echoed message, using "cliaddr" as the destination address.

    }

    return 0;
}
```

**Task 2: TCP based file sending and receiving.**

The previous programming task is about UDP. Now let's start to have a look at TCP, based on the "TCP send" and "TCP receive" code you tried to run in the first assignment.

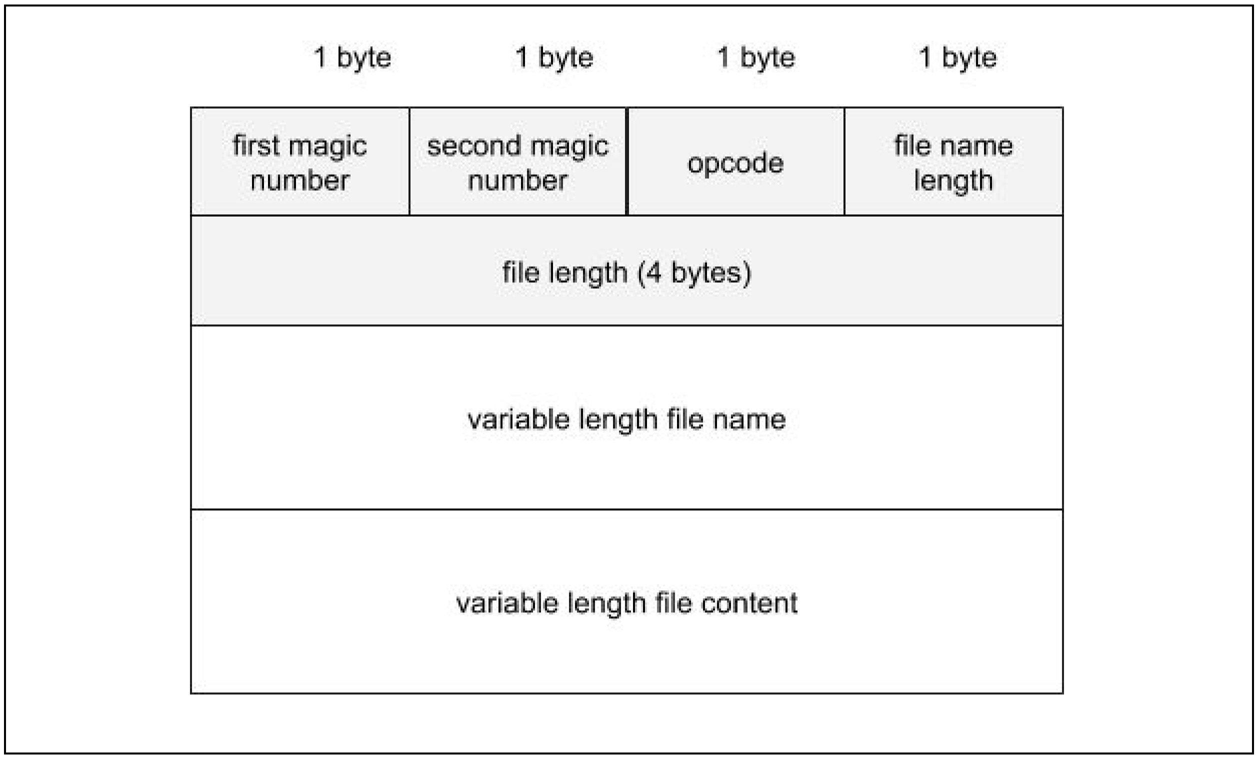
1) Write a client-server application in C or C++ running on Linux. This is a separate application, and please do not merge it with the code for the previous task.
2) The client and the server can run on the same machine or on different machines (Linux).
3) You can interact with the client machine over the command line, i.e., the program reads the line you typed on the command line, like that in "TCP send".
4) You can type a line "upload$file_name", and this line is read into the memory of the client like that in "TCP send". The "file_name" part is the name of a file on the local computer. Note that we use "$" instead of "#" here. After you hit the enter button, the client opens the file, reads each byte in the file, and sends it to the server.
5) Once the server receives the file upload message, it extracts the file name and file content, and then writes the file content into a binary file.
6) After the server finishes receiving the file, it sends back an acknowledgment message. Once the acknowledgment message is received on the client side, the client prints "upload_ack$file_upload_successfully!" on the commandline.
7) You can type a line "download$file_name" to retrieve a file from the server, like that in retrieving a post, but using TCP. The client needs to send a message to the server indicating a download operating is needed. The retrieved file will be stored in a local directory with the specified "file_name" as its file name.
8) Once the server receives a file download message, it reads the previously uploaded file and sends it back to the client. This is similar to the upload procedure, just the work done by the client and the server are switched. After the client finishes saving the received file, it prints "download_ack$file_download_successfully!" on the commandline. Note that there is no further "ack of ack" needed. We are using TCP and it provides reliable transport of byte streams.
9) A message sequence diagram is provided as follows.


client_a
server
client_a types "upload#file_name"
upload message ( file content as payload)
server stores payload of the message as a binary file
upload ack
client_a types "download#file_name"
download message
server load the file and send it back as the payload of ack
download ack (file content as payload)

10) You can design your own protocol to transmit a file. An easy way is just modifying the protocol we have designed for the UDP message application a little bit as follows.

- The same fixed sized header, but this time the header is 8 bytes.
- The first byte and the second byte are the same magic numbers as those in the previous task.
- The third byte is the opcode. For file uploading, the opcode is 0x80; for file uploading acknowledgement, it is 0x81; for file downloading, it is 0x82, and for file downloading acknowledgement, it is 0x83.
- The fourth byte is the length of the file name.
- The next four bytes are a 32-bit integer for the length of the file content. Note that we need to transmit two blocks of variable length data, one is the file name, and the other is the file content. A 32-bit integer for file length is enough for transmitting a file up to 4 GB. As a result, the whole message can be quite long.

The format of the upload message and the download acknowledgement message is shown as follows. Note that they have actual file content as the payload. The format of the download message and the upload acknowledgement does not have the file content, but they should still have the file name.



With this design, one file is transmitted in just one message. Since TCP does the connection oriented reliable transport of continuous byte stream for us, one message should work for this task.

11) The server's IP address and listening port can be hardcoded on both sides, like that in "TCP send" and "TCP receive".

**There are a few important things to be noticed beforehand to avoid wasting time in coding and debugging.**

1. Be careful, it is a binary file that may have any format, not necessarily a text file. Hence, you need to operate a binary file instead of a text file that you did for the server log in the previous task. This means opening the file with fopen("file_name", "rb") and reading it by fread() on the client side. Similarly, on the server side you need to open a binary file with fopen("file_name", "wb"), and write the received bytes to a file using fwrite(). Also, a general binary file may be pretty big, and you do not need to create a huge buffer in the memory to finish receiving every byte before writing it to the file. You can just use a fixed size buffer and call fread() and send() multiple times reusing the buffer. Every time fread() returns with the buffer filled with part of the file data, you can just send it out. Please do remember to check the return value of the fread() because the file size is usually not a multiple of your buffer size. Also, remember to check the return value of the send(). It returns the amount of bytes that is actually sent. In rare cases, the send()

function can be interrupted and return a value less than the expected number of bytes to be sent indicated by its input parameters.

2. For the same reason as that mentioned in the previous item, you can use the fixed buffer for the receiving and file writing on the server side. Note that TCP maintains order in a byte stream for you. Hence, the first call of recv() with a specified buffer size 64 KB will give you the first 64 KB of the byte stream, and the second call of recv() with the same buffer size will give you the second 64 KB. Please do remember to check the return value of recv() and see whether it is 64 KB. Another important thing is that by default, TCP socket blocks, i.e., it waits for future data until it fully fills the buffer you provided. This means the last call of recv() should usually need a buffer less than 64 KB because the file size is usually not a multiple of 64 KB. If you provide a buffer larger than the remaining size of the last segment, recv() will just wait forever for future data, because there will be no more data. Hence, it is important for the receiver side to know the size of the uploaded file in advance before starting receiving the first byte of the file. Note that this is quite different from fread(). One more subtlety is that in some cases, recv() may be interrupted and return a value less than the buffer you specified, for example, if a POSIX signal is delivered to the thread that blocks on recv(). In this case, errno will be set to a value rather than 0. Although this does not happen frequently, your code needs to handle it. In most cases, you just need to call recv() again. This means you need to use the return value of recv() as an indication of the actual progress of the receiving of the byte stream.
3. Note that the file length field in the header is a 32-bit number. Whenever you need to serialize a number into multiple bytes, or deserialize multiple bytes into a number, you need to keep the same "endianness" on both sides. Also, you can see I carefully designed the header to make it aligned to a four byte boundary. It is good practice to always make the header aligned to make it easier and faster to process. Note that if you represent the header as a struct in C or C++, be careful that the compiler may pad bytes to make it aligned to the largest field in the struct, i.e., if one of the fields in the struct contains an int, it is aligned to a four byte boundary, and if one of the fields in the struct contains a double, it is aligned to an eight byte boundary.
4. Note that multiple clients may set up multiple connections and send files at the same time. On the server, you can set up multiple threads for multiple clients, where each thread runs an infinite loop like that in "TCP  receive". The number of clients is usually unbounded and the server usually dynamically creates a new thread each time a new connection is accepted. The newly created thread, i.e., a worker thread, will run an infinite loop that does the actual work (that is the reason why it is called a "worker thread"). In this loop, the server keeps receiving from the connection socket and receives the file. Please search for "Linux + pthread programming" to learn some basics on multithreading on Linux. The provided code snippet already has everything you need for handling multiple connections in multiple threads. However, it is important to understand these codes so that you can diagnose problems when something unexpected happens.

5. You can test this program using different files. You can try to send a picture (e.g., a .jpg file), a PDF file, a text file, or some other files. Make sure the file received by the server is the same as the file sent by the client. TCP provides reliable transport, and most of the time there will be no error. However, as we discussed in class, checksum does not guarantee that the data stream is error free. In very rare cases, the file you transmitted can be corrupted even using TCP. But more often it is corrupted by buggy code. If you find it does not work as you expected, put some print statements in the middle of the program for diagnostics or use the debugger of an IDE.

Deliverables:

1. Please record another short video demonstrating file upload and download with a similar setting as the previous task, and submit a shareable link of the video. Please do not combine the two videos on this task and the previous task together.
2. Please also submit your code on Canvas.

Grading rubrics:

There are 2 points for demonstrating the file upload function, and 2 points for demonstrating the file download function. In total this task has 4 points.

Here is the "**TCP client**" code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[]) {
    int ret;
    int sockfd = 0;
    char send_buffer[1024];
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        printf("socket() error: %s.\n", strerror(errno));
        return -1;
    }

    // Note that this is the server address that the client will connect to.
    // We do not care about the source IP address and port number.

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = htons(31000);

    ret = connect(sockfd,
                  (struct sockaddr *) &serv_addr,
                  sizeof(serv_addr));
    if (ret < 0) {
        printf("connect() error: %s.\n", strerror(errno));
        return -1;
    }

    while (1) {
        fgets(send_buffer,
              sizeof(send_buffer),
              stdin);

        // These two lines allow the client to "gracefully exit" if the
        // user type "exit".

        if (strncmp(send_buffer, "exit", strlen("exit")) == 0)
            break;

        // TODO: You need to parse the string you read from the keyboard,
        // check the format, extract the file name, open the file,
        // read each chunk into the buffer and send the chunk.
        // You need to write an inner loop to read and send each chunk.
        //
        // Note that you need to send the header before sending the actual
        // file data

        while (...) {

            // This the inner loop.

        }
    }

    close(sockfd);

    return 0;
}
```

Here is the "**TCP Server**" code. **Do remember to compile your code with the pthread library.**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// This line must be included if you want to use multithreading.
// Besides, use "gcc ./tcp_receive.c -lpthread -o tcp_receive" to compile
// your code. "-lpthread" means link against the pthread library.
#include <pthread.h>

// This the "main" function of each worker thread. All worker thread runs
// the same function. This function must take only one argument of type
// "void *" and return a value of type "void *".
void *worker_thread(void *arg) {

    int ret;
    int connfd = (int) (long)arg;
    char recv_buffer[1024];

    printf("[%d] worker thread started.\n", connfd);

    while (1) {
        ret = recv(connfd,
                   recv_buffer,
                   sizeof(recv_buffer),
                   0);
        if (ret < 0) {
            // Input / output error.
            printf("[%d] recv() error: %s.\n", connfd, strerror(errno));
            return NULL;
        } else if (ret == 0) {
            // The connection is terminated by the other end.
            printf("[%d] connection lost\n", connfd);
            break;
        }

        // TODO: Process your message, receive chunks of the byte stream,
        // write the chunks to a file. You also need an inner loop to
        // receive and write each chunk.


    }

    printf("[%d] worker thread terminated.\n", connfd);
}


// The main thread, which only accepts new connections. Connection socket
// is handled by the worker thread.
int main(int argc, char *argv[]) {

    int ret;
    socklen_t len;
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
    struct sockaddr_in client_addr;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd < 0) {
        printf("socket() error: %s.\n", strerror(errno));
        return -1;
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(31000);

    ret = bind(listenfd, (struct sockaddr*)
               &serv_addr, sizeof(serv_addr));
    if (ret < 0) {
        printf("bind() error: %s.\n", strerror(errno));
        return -1;
    }


    if (listen(listenfd, 10) < 0) {
        printf("listen() error: %s.\n", strerror(errno));
        return -1;
    }

    while (1) {
        printf("waiting for connection...\n");
        connfd = accept(listenfd,
                 (struct sockaddr*) &client_addr,
                 &len);

        if(connfd < 0) {
            printf("accept() error: %s.\n", strerror(errno));
            return -1;
        }
        printf("conn accept - %s.\n", inet_ntoa(client_addr.sin_addr));

        pthread_t tid;
        pthread_create(&tid, NULL, worker_thread, (void *)(long)connfd);

    }
    return 0;
}
```

**Some more detailed guide on the file transmission part using TCP**

On the client side.

```
// Connection set up code ...

// This the outer loop of the client.
while (1) {

      // The client read a line using fgets()

      // Check this line, whether it contains "upload$" or "download$"
      // at the start. Assume we are dealing with an upload message.


      // Open the file using fopen(). If it is not successful,
      // print a line about the error and break.

      // If the file open is successful, obtain the file name and file size.
      // You can seek to the end and get the offset using fseek()
      // and ftell(), which tells the file size.
      // Note that after this you need to call fseek() again to go back to
      // the beginning of the file. Otherwise your fread() only returns an
      // end-of-file error.
      // Fill the header.
      // You might wonder how to write the file length as the four
      // bytes in the header. Here is a solution.
      send_buffer[0] = MAGIC_1;
      send_buffer[1] = MAGIC_2;
      send_buffer[2] = OPCODE_UPLOAD;
      send_buffer[3] = m; // m is the length of the file name

      int *ptr = (int *)(send_buffer + 4);
      *ptr = file_size;

      // Now you call send() to send the fixed sized header.
      send(sockfd, send_buffer, 8, 0);

      // Then just call send() again to send the variable length file
      // name, assume it has m bytes.
      send(sockfd, file_name, m, 0);


      // Next, use a nested inner loop to send the chunks
      bytes_send = 0;
      while (bytes_send < file_size) {

            // Read a chunk to the send_buffer, e.g., read 1024 bytes.
            // You can declare a static buffer of 1024 bytes,
            // like the "send_buffer" variable. Or you can allocate the
            // buffer dynamically using malloc().
            fread(...)

            // Send this chunk
            ret = send(...)

            // Check the return value. if nothing goes wrong, keep counting.
            bytes_send += ret
      }

      // Once this loop finishes, every byte of the file is sent.
      // Now the client waits for an acknowledgment.

      recv(...)

      // Check whether the received message is OK, and print a line.
      // Note that this simple code does not handle all kinds of errors.
      // It should work if nothing goes wrong.
      // We don't expect you to design a protocol that is robust against
      // all kinds of errors. As long as it works in this simple case,
      // it is OK.
```

```
}
```

On the server side.

```
// This is the main loop of the worker thread
while (1) {

      // Receive the fixed size header, i.e., just read out the first 8
      // bytes.
      ret = recv(connfd, recv_buffer, 8, 0);

      // Parse the "recv_buffer", check the header, and extract the
      // length of the file name. Similarly, extract the file size.
      int m = recv_buffer[3] // size of the file name


      // This is a pointer pointing to the file length field of the header.
      int *ptr = (int *)(recv_buffer + 4);
      // extract the file length as an integer
      int file_size = *ptr;

      // Call recv() again, receive exactly "file_name_length" number
      // of bytes. Do remember to check the return value and verify that
      // it really receives the desired amount of bytes.
      // Note that TCP is different from UDP. A connection is a stream of
      // bytes. Hence, you can call recv() multiple times to only read out
      // part of the bytes in the stream in sequence. However, in UDP, you
      // can only call recvfrom() once to get a single datagram. A datagram
      // is read out as a whole piece or truncated if the buffer you provide
      // is not big enough. But in TCP, you can call recv() many times and
      // each time you read out as many bytes as specified in the second
      // argument of recv().

      // TODO: Now you need to check whether it is an upload message or a
      // download message by inspecting the received header.

      if (recv_buffer[2] == OPCODE_UPLOAD) {

          // Now we know it is an upload message.

          // Assume the file name can not be too long.
          char file_name[128];
          // Now we receive exactly m bytes as the file name.
          ret = recv(sockfd, file_name, m, 0);

          // Open a new file to writing, using the file name you received.
```

```
            // Now we know the name of the file, create it by fopen().
            fopen(file_name, "w");

            // This is the inner loop, like that on the client side.
            // Now we know we need to receive exactly this "file_size"
            // amount of bytes.
            bytes_received = 0;
            while (bytes_received < file_size) {

                // receive a chunk to the recv_buffer, e.g., 1024 bytes.
                // Similarly, you can declare a static buffer of 1024 bytes,
                // or you can malloc() dynamically.
                bytes_received += recv(...)

                // write this chunk to the file
                fwrite(...)
            }

            // Once this loop finishes, every byte of the file is received.
            // Now you send an acknowledgment.
            send(...)

        } else if (...) {





        } else {

            // Unknown opcode!! Probably your client code is buggy!!
            // Just print out a line and abort.
            exit(-1);
        }


}
```

**Preview of the next assignment**

The programming assignments are purposely broken into multiple tasks to decrease the difficulty, and in the next assignment, we will keep working on it and add more features. The UDP part is the core of the messaging protocol, and the TCP part is a supplement for transmitting files such as pictures In this assignment, many important features are omitted so that you can focus on learning to use the socket API. For example, in an online messaging application like Facebook Messenger or Twitter, the client should be able to connect to the server and disconnect from the server dynamically. The server should be able to accommodate huge amount of clients efficiently (as you might expect, handling one connection with an independent thread is not efficient). Also, each client should have a client ID and password to "log in" in the first few messages. One client should be able to post under a topic and follow a topic to receive feeds. We will expand the requirements and write the core of an almost full-fledged online messaging program in the next assignment based on what we have learned in this assignment.

If you have time and you want to explore Linux network programming as well as Linux system programming, you are welcome to read some books or online tutorials on this topic. Some of the books that I recommend are shown as follows. They are all available online. Just search the book name with "PDF" and you will find the link to download a copy of it.

- The Definitive Guide to Linux Network Programming, by Keir Davis, John W. Turner, Nathan Yocom
- Linux System Programming, by Robert Love
- Unix Network Programming, by Richard Stevens
- Advanced Programming in the Unix Environment, by Richard Stevens

Among them, the last two books are considered the "bible" of this field. Don't be scared with the word "Unix". Essentially the system APIs are the same as those on Linux. However, these two books are pretty thick.