CSE 434 Computer Networks
(Fall 2019) Assignment 4

Duo Lu <duolu@asu.edu>

The due date of this assignment is 11:59 pm on November 27, 2019 (**firm deadline**). This is a programming assignment, which is designed to allow you to gain hands-on experience socket programming. This can easily take more than 10 hours because a lot of features are needed to be implemented. **Note that there will not be due date extension any more.** As you may have learned from the previous assignments, the assignment requires some effort. Hence, please start early and search online from time to time to make sure that you understand the code. **This assignment also has an optional part for extra credit in this class.** The grading is effort-based. Note that you can use other programming languages instead of C/C++ and you can also implement it on other platforms instead of Linux, for example, you can implement the client as an Android App or IOS App and put the server on Amazon AWS. However, all the instructions provided in this document are using pure C on Linux and we only illustrate the details of the protocol.

Note that the due date of the bonus part (i.e., task 2) is the end of this semester, i.e., Dec. 11. However, you still need to submit the basic part (i.e., task 1 and 3) by Nov. 27, end-of-the-day. If you would like to submit the bonus part, please send an email to me and our TA as a notification.

Note that this assignment has been updated on Nov. 4 and now the total points are changed to 14 points. Note that more instructions were added on Nov. 14. Basically I demonstrated how to implement the login message and the post message on the client and the server. You can use them as an example to implement other messages.

(continued in the next page)

Task 1. UDP based real-time message sending and receiving with multiple users.

In the previous assignment, you can communicate with the server using a UDP protocol invented by you. However, in that protocol, clients can not talk to each other. In this assignment, you are going to use the server as a "hub" to allow clients on the "spoke" to see messages from other clients, like Twitter, also with real-time message feeds. See the following instructions.

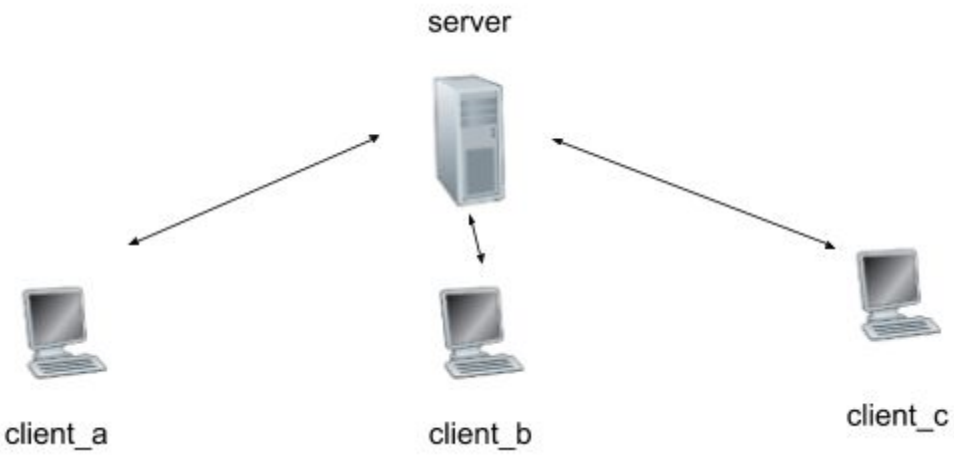


Figure 1: An illustration of communication between a server and multiple clients.

1) Write a client-server application. They may be deployed on the same machine or multiple machines as shown in the figure above.
2) Now consider you are the "**client_a**".
3) On the client side, you can type a line "login#client_a&password". Here "client_a" is the client ID and "password" is the password of this client. Each client has a unique ID, which is just a string consisting of letters, numbers, and underscores. Each client also has a password (which may not be unique). You can make up these IDs and passwords as you like. For example, I may type "login#duolu&123456", which means that my ID is "duolu" and my password is "123456". For privacy reasons, please do not use some ID and password you use in your daily life. The client will read this line and send a login message to the server in a UDP datagram. You may need to design a new message format for this login message. The easiest way is reusing the message format in the previous assignment by adding another opcode value for the login function. This means the client ID and password are sent in text as a message payload, in the exact same format as a line of user input. On the server side, the server needs to parse this line to extract the client ID and the password. See the next few pages for details of the message format.
4) The server keeps a list of all clients with their IDs and passwords, stored in plain text in a file, one line for each client. At the start of the server program, it reads each line into the memory and stores them in an array or some other data structure. You can define your own data structure in memory and the format in the file to hold the user account

information. **Alternatively, you can hardcode the password and ID of every user in your code if it is difficult for you to manipulate files. This can work in this exercise but definitely not feasible in a real software product.**

5) When the server receives a login message, it compares with the client information read from the file and the client information in the message, and authenticates the client. If the password matches, the server sends back a login success acknowledgement message. On the other hand, if the password does not match, the server sends back a login failure acknowledgement. The login success ack message contains a 32-bit number randomly generated by the server as a token. Most importantly, this token must be unique for each client. This token also identifies a "session" that spans the time after the user login and until the user logout. The server needs to remember the client ID, the token, as well as the IP address and the port number for each client, and it keeps a record on which user is online and which user is offline. You can use a structure to represent all user profile information for each user. Note that the IP address and the port number can be obtained when the login message datagram is received, just like that in assignment 3.
6) Once the client receives the ack on a successful login, it prints out a line of "login_ack#successful", and it also needs to save the token. If the login is not successful, it prints out a line of "login_ack#failed".
7) Note that all following actions can only be done after login. If the client sends a message to the server without a successful login beforehand, the server sends back an error message, and the client prints out "error#must_login_first".
8) On the client side, you can type a line "subscribe#client_b" to subscribe to another client who has an ID of "client_b". This is similar to the function of "follow" another user on Twitter. Similarly, you can type a line "unsubscribe#client_b" to cancel the subscription. The client needs to send two types of messages to the server corresponding to these two functions.
9) On the server side, by receiving the subscribe and unsubscribe messages, the server needs to check the existence of the target client, and return messages indicating whether the action is successful or not. When the client receives the acknowledgement, it prints out a line similar to the login message, i.e., something like "subscribe_ack#successful", or "subscribe_ack#failed", or "unsubscribe_ack#successful" or "unsubscribe_ack#failed". The server needs to maintain a table to track which client subscribes which client. (Hint: use the "map" data structure from the C++ standard template library). **Note that for each message from the client to the server or from the server to the client after login, the token must be carried in the message.** This token allows the server to identify a session, i.e., which client generates the message. There are multiple reasons that a subscription action is not successful, e.g., the target client does not exist, etc. Also, there are multiple reasons that an unsubscription action is not successful, e.g., the target client does not exist, or it exists but not in the subscription list of the client, etc.
10) On the client side, you can type a line "post#some_text" to post a message (detailed later). This is the same as that in the previous assignment. However, if this client has subscribers who are also online (i.e., they have signed in), they will receive real-time message feed. This is like asking the server to "push" messages to those clients who are

interested, and hence, the server needs to send a forwarding message to these clients. For the subscribers, their client program will print "<client_a>some_text", where "client_a" is the client ID of the publisher. Also, the server needs to send back an acknowledgement to the client and the client prints a line of "post_ack#successful".

11) On the client side, you can type a line "retrieve#n" to retrieve n most recently posted text lines from those clients that you have subscribed to (detailed laler). Note that in total there are n individual posts retrieved from all subscribed clients, in chronological order from the most recent to the least recent. This is especially useful when a client just signs in and wants to "pull" some message feeds. Note that the server needs to send back n messages. These retrieved text lines will be printed by the client in the format of "<client_a>some_text", which is the same as real-time message feed.
12) You can type a line "logout#" to instruct the client to log off and the client program will send a logout message to the server. After that, the server will send back a logout ack and the client will print out a line "logout_ack#successful". Once the client finishes logout, it will not receive any message from the server.
13) The server regularly checks the last time that a client sends and receives a message. If the client is not active for 1 minutes, the server considers it offline and changes the states of the table that the server keeps for all clients. In this case, if the client sends some message, it will trigger the "must-login-first" error. You can change the "1 minute" timeout to a proper value that best suitable for your demo.

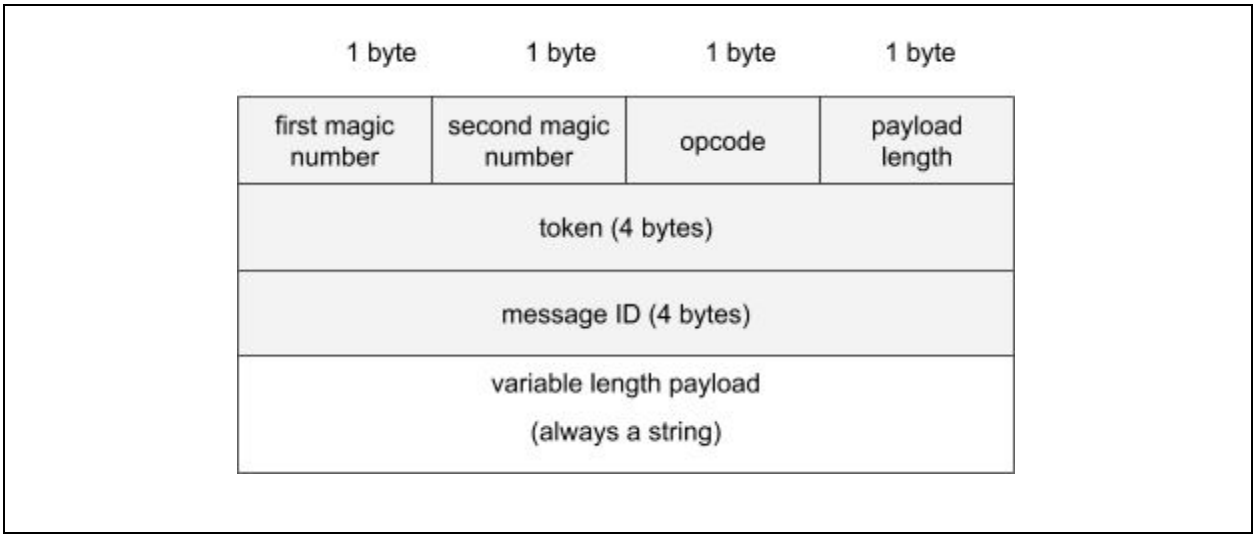
(continued in the next page)

Here is a table listing all types of messages between the client and the server. You can see all messages are in a request-response paradigm. All the messages from the client to the server are highlighted in red and all the messages from the server to the client are highlighted in blue. The only purple message can be in both directions.

Table 1: Summary of messages.

opcode	message name	direction	payload
0x00	session reset	both	none
0xF0	must-login-first error	server ---> client	none
0x10	login	client ---> server	client ID and password
0x80	successful login ack	server ---> client	none
0x81	failed login ack	server ---> client	none
0x20	subscribe	client ---> server	target client ID
0x90	successful subscribe ack	server ---> client	none
0x91	failed subscribe ack	server ---> client	none
0x21	unsubscribe	client ---> server	target client ID
0xA0	successful unsubscribe ack	server ---> client	none
0xA1	failed unsubscribe ack	server ---> client	none
0x30	post	client ---> server	text line
0xB0	post ack	server ---> client	none
0xB1	forward	server ---> client	source client ID, text line
0x31	forward ack	client ---> server	none
0x40	retrieve	client ---> server	none
0xC0	retrieve ack	server ---> client	source client ID, text line
0xC1	end of retrieve ack	server ---> client	none
0x1F	logout	client ---> server	none
0x8F	logout ack	server ---> client	none

Here is an example message format in binary. Note that you can design your own message format and it is not restricted to binary, for example, you can use JSON object as the message format and transmit text messages. In this example message format, each message has a fixed size header of 12 bytes, similar to the message format in the previous assignment.



1. The **first and second magic numbers** can be set as your name initial.
2. The **opcode** values are provided in Table 1. Of course these are just a bunch of arbitrary numbers and you can assign values as you like. Here all messages from the client to the server have an opcode less than 0x80 and all messages from the server to the client have an opcode equal or greater than 0x80, except the session reset message, which can be sent in both directions.
3. The **payload length** indicates how many bytes are in the payload.
4. The token is a 32-bit number generated at login to uniquely identify a session linked to an online user.
    a. For the login message, this field is zero.
    b. For the login ack, this field is the token generated by the server and this token is saved by both the server and the client.
    c. For all other messages, this field is filled with the saved token.
5. The **message ID** is an ID used to identify an individual message. The server generates this ID upon receiving a post message, and it is monotonically increasing. This field is only used in the forward message, forward ack, and in the retrieve ack. It is always set by the server. For other messages, this is always zero.
6. The **variable length payload** is always a string.
    a. For those messages with no payload, this part is empty and the payload length field is zero.
    b. For the login message, this part is a string of "client_ID&password". Note that the "&" character is used to separate the client ID and the password. This means that the client ID cannot have a "&" in it.

c. For the subscribe and unsubscribe message, this is a string of "client_ID" which indicates the target client for subscribe or unsubscribe.
d. For the post message, the payload is a line of text.
e. For the forward message and the retrieve ack message, The payload is a string containing a client ID and a line of text, such as "<client_ID>some_text", where the "client_ID" is the ID of the client that initially posts the line of text. Note that the retrieve ack message contains only one line. For retrieving n lines, n retrieve ack messages are needed (see the next page for more details).

(continued in the next page)

Now we discuss the details of the protocol. A sequence diagram explaining the protocol on login, logout, and real-time message feed is provided in Figure 2, assuming that client_b has subscribed to client_a. The blue parts are user interactions, the red parts are messages and acks, and the shaded parts are server behaviors.

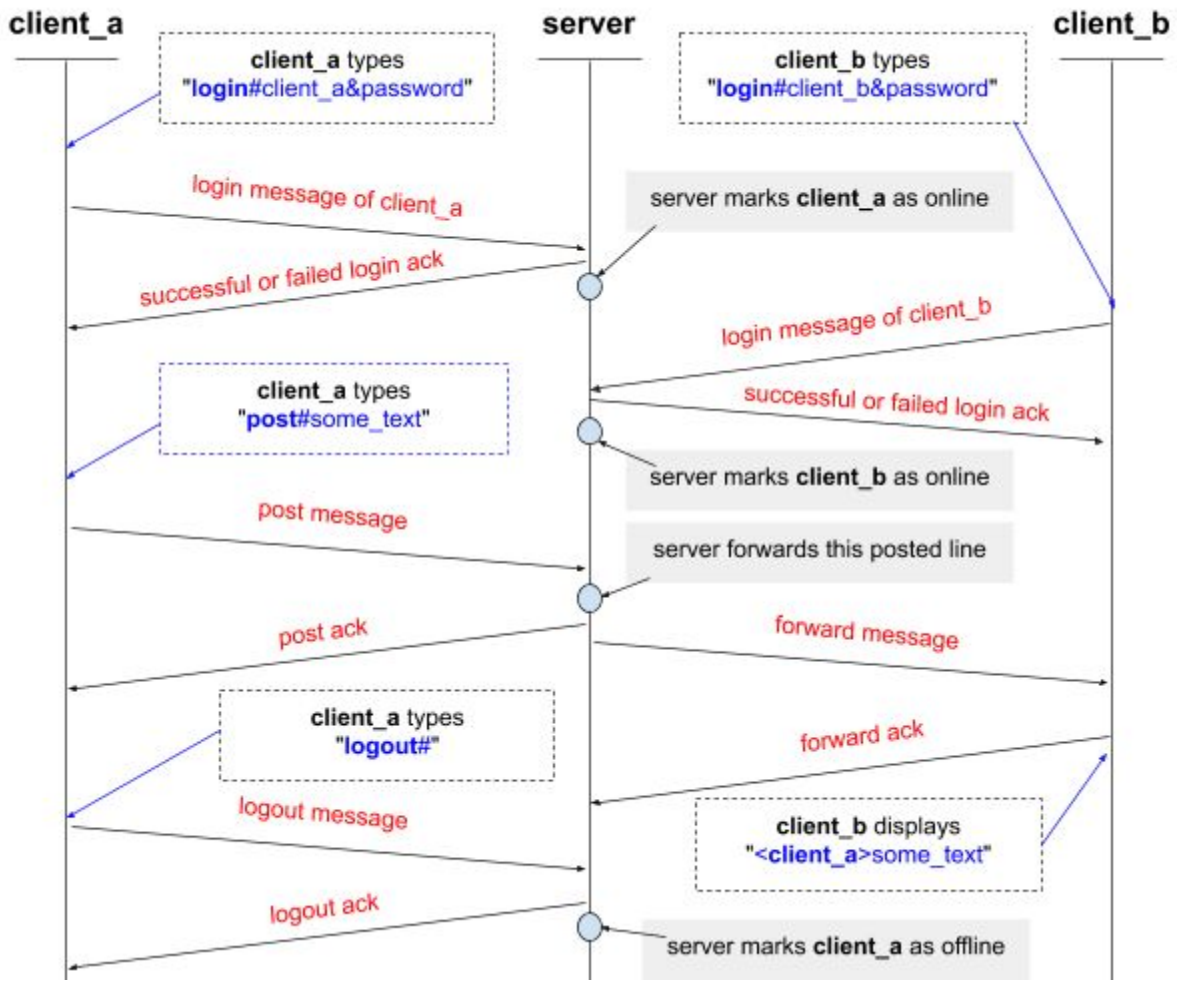


Figure 2: Illustration of real-time message feed.

Note that all messages except for the login message must be sent after the login message. Otherwise, the server sends back an error message indicating the client must login first. This means that the server must be able to remember the states of each client.

The forward message is sent immediately after the server receives the post message. Note that the publisher does not know whether some subscribers are online, and hence, the post ack does not depend on the forward message or the result of forwarding. If there are multiple subscribers, the server will send one forward message to each subscriber. Technically, one

message essentially only involves one client and the server. All clients are decoupled. Only the server knows the existence of multiple clients.

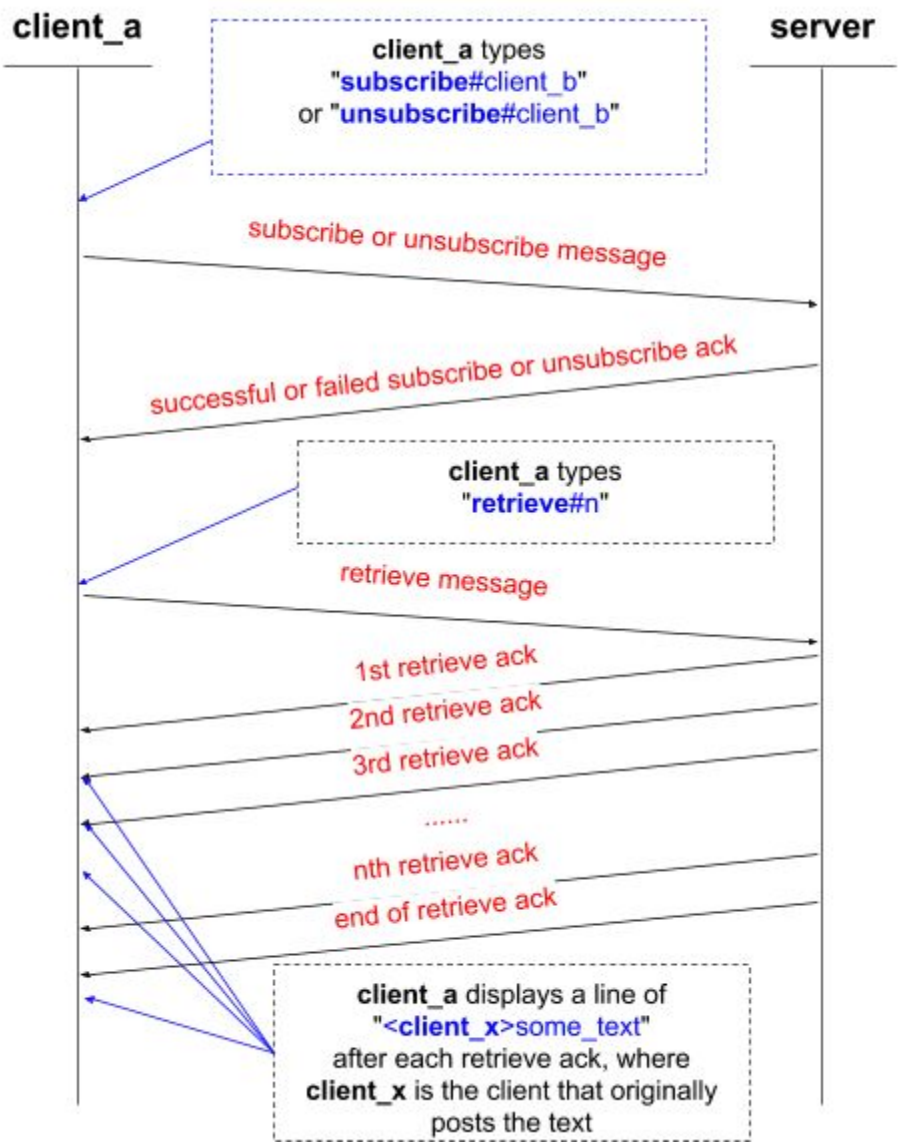


Figure 3: Sequence diagram of subscribe, unsubscribe, and retrieve.

An illustration of other messages are provided in Figure 3. In our design of message format, one retrieve ack can only carry one text line. Hence, the retrieve-n-lines is implemented as one retrieve message from the client to the server and n retrieve acks from the server to the client.

Now we provide more details on the behaviors of the client and the server with an emphasis on how to write code to implement the behaviors. Both the client program and the server program are "**event-driven**", which means they respond to certain events and if there

are no events the program merely waits on events. Moreover, the protocol is "**stateful**", which means that both the client and the server need to remember something that happened before, i.e., they need to declare a few variables to hold the "states" of communication. Typically we call a stateful sequence of messages a "**session**" in the perspective of the server. This session usually starts after the login and stops after logout. One client can be involved in one session to a specific server, and one server needs to hold multiple sessions, one for each online client. Note that a session may span multiple TCP connections or even in UDP like our protocol. The "**token**" mentioned previously is used as a session ID.

See the following **state transfer diagram** for a client in Figure 4. It is a common practice to model the behavior of network applications using a state transfer diagram. Because our message has a request response paradigm, we add a few auxiliary states to handle each message. Note that in the diagram I draw three cascaded states, they are three separate states, i.e., the "post sent" state, the "subscribe sent" state, and the "unsubscribe sent" state. Their behaviors are similar and to save some space, I draw them in a cascaded way. But you should understand that they are essentially three independent states and the transition does not depend on each other.

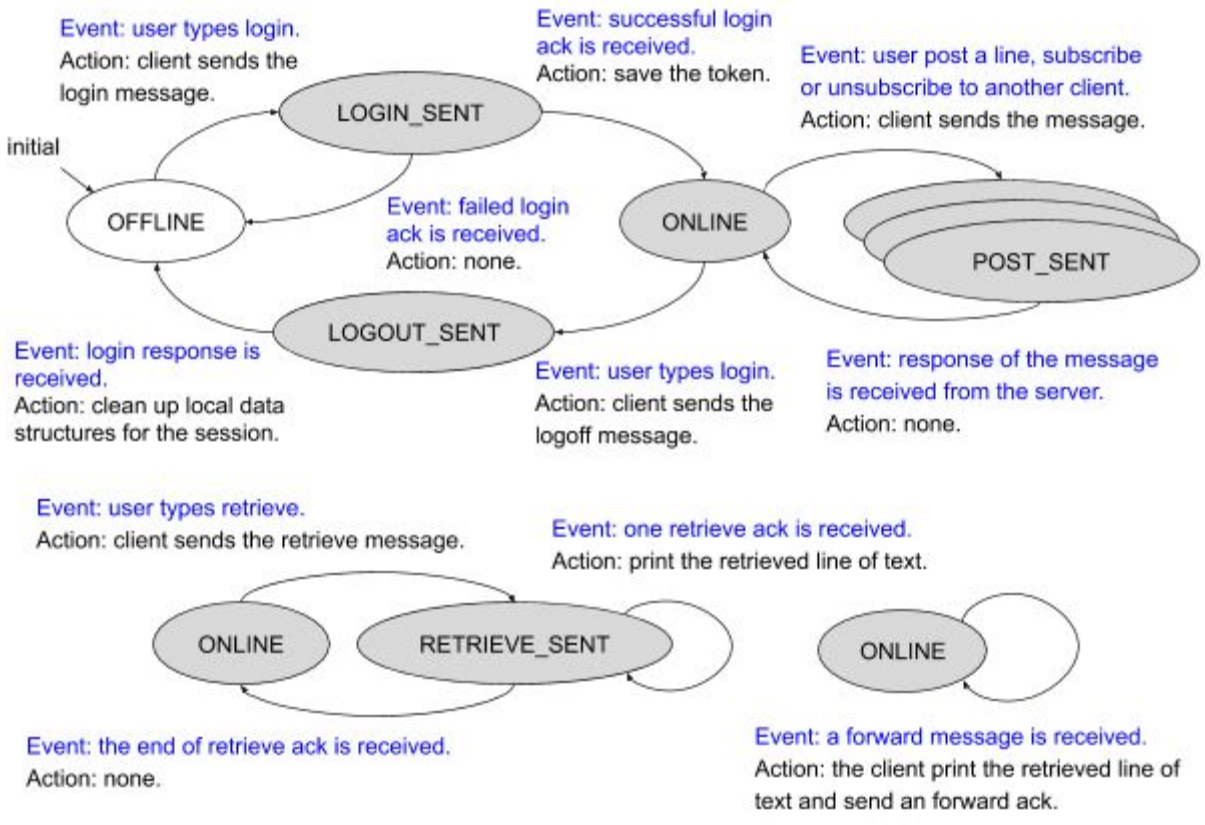


Figure 4: State diagram of a session on the client side

You might wonder what triggers the session state to change. It is an "**event**", i.e., those on the edge of the state diagram. This is similar to the TCP state machine we saw in the lecture. The transitions are detailed as follows. States are highlighted in boldface, user interactions are highlighted in blue, and messages are highlighted in red.

1) Initially, the client software is in the "**OFFLINE**" state.
2) At the "**OFFLINE**" state, the user can type "login#client_id&password", and this event is from the keyboard input. Upon this event, the client software sends the "login message" and transits to the "**LOGIN_SENT**" state. All other events happening in this state will make the client software stay in the "**OFFLINE**" state.
3) At the "**LOGIN_SENT**" state, the client may receive the "login ack" sent back by the server. If the login is successful when the client is at the "**LOGIN_SENT**" state. This is an event from the network side. Upon this successful event, the client software transits to the "**ONLINE**" state. If the login ack indicates a failure or an error, it goes back to the "**OFFLINE**" state. All other events trigger the client to go back to the "**OFFLINE**" state. (Think about what if the server is down and it does not send back any response? Will the client stuck forever? It is not required to handle such cases in this task.)
4) At the "**ONLINE**" sate, the user can type a line "post#some_text", which is an event from the keyboard input. The corresponding action is sending a "post message" to the server. Then, the client software transits to the "**POST_SENT**" state. If some error happens, e.g., the client may receive some spurious message, the client may go back to the "**OFFLINE**" state if the client sees it as an error or stay in the "**ONLINE**" state if the client considers it can handle such an error. This is a design issue and it depends on you. The protocol does not specify every detail.
5) At the "**POST_SENT**" state, if the client receives the "post ack" from the server successfully, it transits to the "**ONLINE**" state. This is an event from the network side. Note that the client can only send one message at a time.
6) The subscribe and unsubscribe functions are similar to the post function, just the names of the states are different. (Note that similar to login, the client can get stuck in these states if the ack is lost. It is not required to handle such cases in this task.)
7) At the "**ONLINE**" state, if the client receives a "forward message", it prints a line of text in the message and remains in the "**ONLINE**" state.
8) At the "**ONLINE**" state, the user can type a line "retrieve#n" to retrieve n most recently posted text lines. The client sends a "retrieve message" to the server and transits to the "**RETRIEVE_SENT**" state.
9) At the "**RETRIEVE_SENT**" state, the client receives the "retrieve ack", print out the retrieved lines of text, and stays in the "**RETRIEVE_SENT**" state. Only if the client receives the "end of retrieve ack", it transits back to the "**ONLINE**" state. (You can see there is the same issue as login, it can get stuck. It is not required to handle such cases in this task.)
10) Similar to log in, at "**ONLINE**" state, the user can type a line "logout#", and this event is from the keyboard input. Upon this event, the client software sends the "logout message"

and transits to the "**LOGOUT_SENT**" state. (You can see there is the same issue as login, it can get stuck. It is not required to handle such cases in this task.)

11) At the "**LOGOUT_SENT**" state, the client may receive the "logout ack" from the server and transits to the initial "**OFFLINE**" state. Upon successful logout, the client software clears any variables used for the session and restores to a status like just started.
12) In any state, if the incoming event does not follow the previous rules, the client transits to the "**OFFLINE**" state and prints out a line indicating that an error happens and send the "session reset message" to the server.
13) In any state, if the client receives a "session reset message", it directly transits to the initial "**OFFLINE**" state and print out a line indicating the session is destroyed.

Note that **the client may need to run two threads**, so that one thread monitors the keyboard and the other keeps receiving messages from the server. Be careful about data race and synchronization among multiple threads. **Alternatively, you may use the I/O multiplexing method shown in the socket programming lecture. Usually when dealing with events from multiple sources, I/O multiplexing can be easier than multiple threads**. The server does not need to use multiple threads because UDP is connectionless (yes, we intentionally use UDP to make it easier). In this case, the server will still block at recvfrom(). However, you can configure timeout on a socket to set up a limit of the waiting time. See this link:

https://stackoverflow.com/questions/2876024/linux-is-there-a-read-or-recv-from-socket-with-timeout

You might also wonder how to write the code to implement this state diagram on the client. Here is some example code that may help. Note that this code uses I/O multiplexing. You need to understand the behavior of the code before using it.

Note that my example code is in pure C, which might not look perfect in a modern perspective, and my coding style emphasizes the structure, which might not be good from a production point of view nor from clarity. You may need to encapsulate the processing of each event into a separate function.

```
#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>

struct header {

    char magic1;
```

```
    char magic2;
    char opcode;
    char payload_len;

    uint32_t token;
    uint32_t msg_id;
};

const int h_size = sizeof(struct header);

// These are the constants indicating the states.
#define STATE_OFFLINE          0
#define STATE_LOGIN_SENT       1
#define STATE_ONLINE           2
// Now you can define other states in a similar fashion.

// These are the constants indicating the events.
// All events starting with EVENT_USER_ are generated by a human user.
#define EVENT_USER_LOGIN               0
#define EVENT_USER_POST                1
// Now you can define other events from the user.
......
#define EVENT_USER_INVALID             79

// All events starting with EVENT_NET_ are generated by receiving a msg
// from the network. We deliberately use larger numbers to help debug.
#define EVENT_NET_LOGIN_SUCCESSFUL      80
#define EVENT_NET_POST_ACK              81
// Now you can define other events from the network.
......
#define EVENT_NET_INVALID               255

// These are the constants indicating the opcodes.
#define OPCODE_RESET                    0x00
#define OPCODE_MUST_LOGIN_FIRST_ERROR   0xF0
#define OPCODE_LOGIN                    0x10
// Now you can define other opcodes in a similar fashion.
......

int main() {

    char user_input[1024];

    int ret;
    int sockfd = 0;
    char send_buffer[1024];
    char recv_buffer[1024];
    struct sockaddr_in serv_addr;
    struct sockaddr_in my_addr;
    int maxfd;
    fd_set read_set;
    FD_ZERO(&read_set);
```

```
// You just need one socket file descriptor. I made a mistake previously
// and defined two socket file descriptors.
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd < 0) {
  printf("socket() error: %s.\n", strerror(errno));
    return -1;
}

// The "serv_addr" is the server's address and port number,
// i.e, the destination address if the client needs to send something.
// Note that this "serv_addr" must match with the address in the
// "UDP receive" code.
// We assume the server is also running on the same machine, and
// hence, the IP address of the server is just "127.0.0.1".
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(32000);

// The "my_addr" is the client's address and port number used for
// receiving responses from the server.
// Note that this is a local address, not a remote address.
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
my_addr.sin_port = htons(some_semi_random_port_number);


// Bind "my_addr" to the socket for receiving messages from the server.
bind(sockfd,
     (struct sockaddr *) &my_addr,
     sizeof(my_addr));

maxfd = sockfd + 1; // Note that the file descriptor of stdin is "0"

int state = STATE_OFFLINE;
int event;
uint32_t token; // Assume the token is a 32-bit integer

// This is a pointer of the type "struct header" but it always points
// to the first byte of the "send_buffer", i.e., if we dereference this
// pointer, we get the first 12 bytes in the "send_buffer" in the format
// of the structure, which is very convenient.
struct header *ph_send = (struct header *)send_buffer;
// So as the receive buffer.
struct header *ph_recv = (struct header *)recv_buffer;

while (1) {
```

```
        // Use select to wait on keyboard input or socket receiving.
        FD_SET(fileno(stdin), &read_set);
        FD_SET(sockfd, &read_set);

        select(maxfd, &read_set, NULL, NULL, NULL);

        if (FD_ISSET(fileno(stdin), &read_set)) {

            // Now we know there is a keyboard input event
            // TODO: Figure out which event and process it according to the
            // current state

            fgets(user_input, sizeof(user_input), stdin);

            // Note that in this parse function, you need to check the
            // user input and figure out what event it is. Basically it
            // will be a long sequence of if (strncmp(user_input, ...) == 0)
            // and if none of the "if" matches, return EVENT_USER_INVALID
            event = parse_the_event_from_the_input_string(...)

            // You can also add a line to print the "event" for debugging.

            if (event == EVENT_USER_LOGIN) {
                if (state == STATE_OFFLINE) {

                    // CAUTION: we do not need to parse the user ID and
                    // and password string, assuming they are always in the
                    // correct format. The server will parse it anyway.

                    char *id_password = user_input + 6 // skip the "login#"
                    int m = strlen(id_password);

                    ph_send->magic1 = MAGIC_1;
                    ph_send->magic2 = MAGIC_2;
                    ph_send->opcode = OPCODE_LOGIN;
                    ph_send->payload_len = m;
                    ph_send->token = 0;
                    ph_send->msg_id = 0;

                    memcpy(send_buffer + h_size, id_password, m);

                    sendto(sockfd, send_buffer, h_size + m, 0,
                        (struct sockaddr *) &serv_addr, sizeof(serv_add));

                    // Once the corresponding action finishes, transit to
                    // the login_sent state
                    state = LOGIN_SENT;
```

```
        } else {

            // TODO: handle errors if the event happens in a state
            // that is not expected. Basically just print an error
            // message and doing nothing. Note that if a user types
            // something invalid, it does not need to trigger a
            // session reset.

        }

    } else if (event == EVENT_USER_POST) {

            // Note that this is similar to the login msg.
            // Actually, these messages are carefully designed to
            // somewhat minimize the processing on the client side.
            // If you look at the "subscribe", "unsubscribe", "post"
            // and "retrieve", they are all similar, i.e., just fill
            // the header and copy the user input after the "#" as
            // the payload of the message, then just send the msg.

            char *text = user_input + 5 // skip the "post#"
            int m = strlen(text);

            ph_send->magic1 = MAGIC_1;
            ph_send->magic2 = MAGIC_2;
            ph_send->opcode = OPCODE_POST;
            ph_send->payload_len = m;
            ph_send->token = token;
            ph_send->msg_id = 0;

            memcpy(send_buffer + h_size, text, m);

            sendto(sockfd, send_buffer, h_size + m, 0,
                (struct sockaddr *) &serv_addr, sizeof(serv_add));


    } else if (event == EVENT_USER_RESET) {

        // TODO: You may add another command like "reset#" so as to
        // facilitate testing. In this case, a user just need to
        // type this line to generate a reset message.

        // You can add more commands as you like to help debugging.
        // For example, I can add a command "state#" to instruct the
        // client program to print the current state without chang
        // -ing anything.
```

```
        } else if (event == .../* some other event */) {

            // TODO: process other event

        }

    }
    if (FD_ISSET(sockfd, &read_set)) {

        // Now we know there is an event from the network
        // TODO: Figure out which event and process it according to the
        // current state

        ret = recv(sockfd, recv_buffer, sizeof(recv_buffer), 0);

        event = parse_the_event_from_the_received_message(...)

        if (event == EVENT_NET_LOGIN_SUCCESSFUL) {
            if (state == STATE_LOGIN_SENT) {

                token = ph_recv->token;

                // TODO: print a line of "login_ack#successful"
                state = STATE_ONLINE;

            } else {

                // A spurious msg is received. Just reset the session.
                // You can define a function "send_reset()" for
                // convenience because it might be used in many places.
                send_reset(sockfd, send_buffer);

                state = STATE_OFFLINE;
            }
        } else if (event == EVENT_NET_LOGIN_FAILED) {
            if (state == STATE_LOGIN_SENT) {

                // TODO: print a line of "login_ack#failed"
                state = STATE_OFFLINE;

            } else {

                send_reset(sockfd, send_buffer);

                state = STATE_OFFLINE;
```

```
                }

            } else if (event == EVENT_NET_FORWARD) {
                if (state == STATE_ONLINE) {

                    // Just extract the payload and print the text.
                    char *text = recv_buffer + h_size;

                    printf("%s\n", text);

                    // Note that no state change is needed.

                } else {


                }

            } else if (event == ......) {

                // TODO: Process other events.


            }




        }

        // Now we finished processing the pending event. Just go back to the
        // beginning of the loop and waiting for another event.
        // Note that you can set a timeout for the select() function
        // to allow it to return regularly and check timeout related events.

    } // This is the end of the while loop

} // This is the end of main()
```

See the following state transition diagram for a session on the server. Note that the states of a session on the server are different from the states on the client because they are in two different programs running in two places. Also, the state of one session is different from the state of another session. You need one state variable associated with one session.

(continued in the next page)

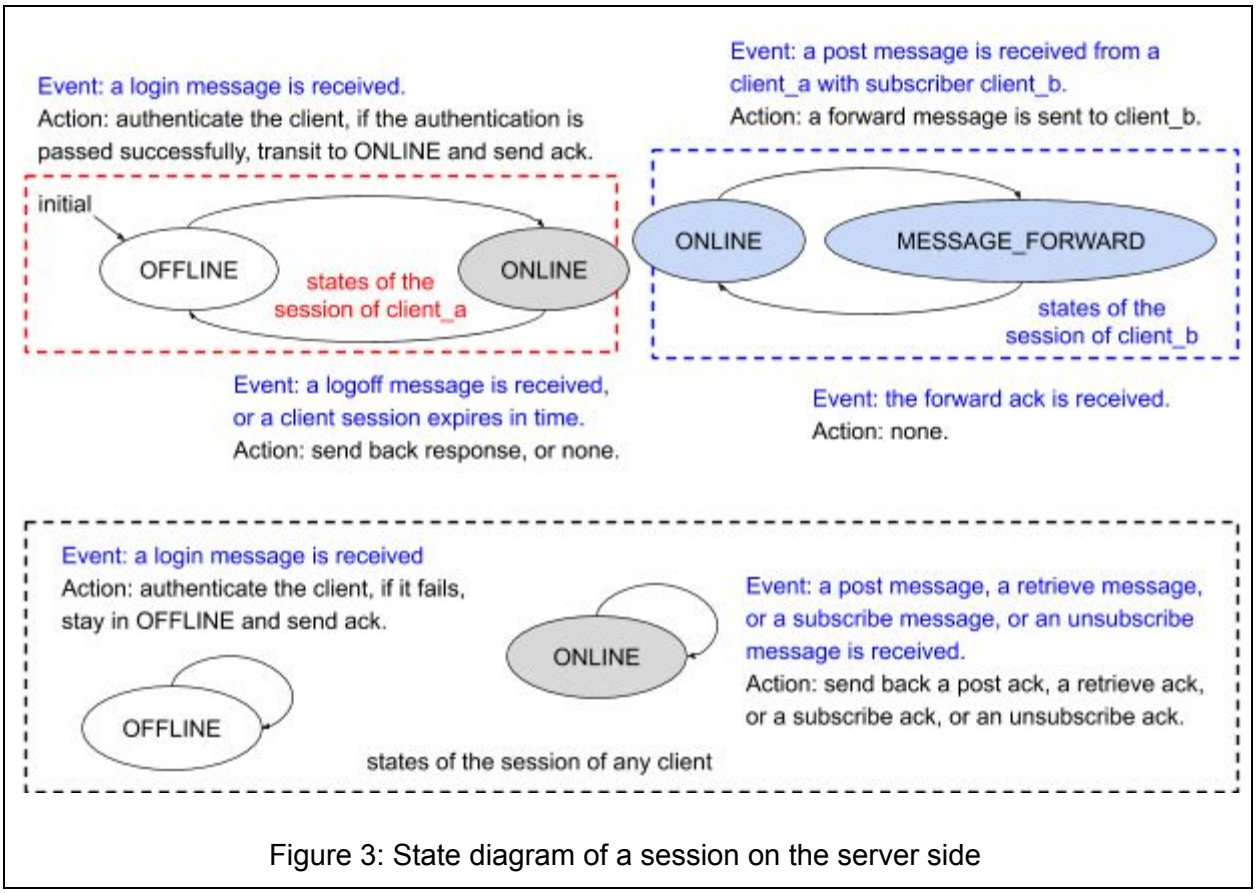


Figure 3: State diagram of a session on the server side

For a single session between a client and the server, there are several different types of events all from the network side. Note that a server can have multiple sessions at the same time. Each session has its own states.

1) Initially, all sessions are in the "**OFFLINE**" state.
2) At the "**OFFLINE**" state, the server can receive a "login message" from the client_a and the authentication is passed, the session corresponding to the client_a transits to the "**ONLINE**" state. If the authentication fails, it stays in the "**OFFLINE**" state. After that, the server sends back a "login ack" Note that only the session regarding client_a changes states. Other sessions are irrelevant to this message from client_a.
3) At the "**ONLINE**" state, the server may receive a "post message" from client_a, who has a subscriber client_b. In this case, the server immediately sends back a "post ack" to client_a, and the session of client_a does not change state (i.e., remain in the "**ONLINE**" state). The message forwarding to client_b may not be successful, and the response may indicate a failure. But the session of client_a does not need to know that. Meanwhile, the server sends a "forward message" to client_b and the session of client_b transits to the "**MESSAGE_FORWARD**" state, if client_b is online (i.e., if the session of client_b is in the "**ONLINE**" state). Note that upon receiving the "post message", the server needs to generate a message ID for it. You can use a global counter on the server and increment it every time a "post message" is received. The server also needs

to store the text line together with the message ID in a list so that later it can do the retrieve function. Since only a client can post a text line, this list is automatically sorted chronologically.

4) At the "**MESSAGE_FORWARD**" state, if the server receives a "forward ack" from client_b, the session of client_b transits back to the "**ONLINE**" state. Note that there is a tricky case that multiple clients can send messages all targeting client_b. In this case, even if the session of client_b is in the "**MESSAGE_FORWARD**" state, the server should still forward a message to client_b. However, the server needs to track which message gets an ack and which message does not get an ack, i.e., using the message ID. Once all messages get the responses, the session transits back to the "**ONLINE**" state.
5) At the "**ONLINE**" state, the server may receive a "retrieve message", a "subscribe message" or an "unsubscribe message". The server processes these messages accordingly and sends back a "retrieve ack", a "subscribe ack" or an "unsubscribe ack". Note that we do not need to introduce more states for these events. A state is introduced so that the software will wait in that state for some events out of the control of the software, usually some user input or messages from the network. Here for the server to finish these functions such as retrieving, subscribing, or unsubscribing, it does not need to wait. Instead, the server can process them immediately by just looking at the data saved locally. Also note that the server essentially serializes the message receiving, i.e., it receives one message at a time and processes one message at a time. Hence, when the server processes the retrieve message, even if it may take some time, the server is not interrupted by another message until it finishes the processing. This significantly simplifies our design but it also introduces performance bottleneck, i.e., the server can only utilize one CPU core by running only one thread from the beginning to the end. Note that the server keeps a list of all posted messages for the "retrieve n most recent text lines" function. To implement this function for a specific client, the server just needs to scan the list from the end that text lines are inserted, and for each message, get the publisher of the message, and check whether the publisher is in the subscription list of this client. If it is, just increment a counter and move on; Otherwise, do not increment the counter and move on. Once the counter reaches n or the other end of the list, just stop and package these n text lines into n retrieve ack. This also means that for each client, the server needs to maintain a subscription list. At last, when all retrieve acks are sent, the server sends out the "end of retrieve ack". There may be n acks, or less than n acks, or even no ack contains actual line of text, but there must be an "end of retrieve ack" to allow the client to properly determine the end and transit back to normal online state.
6) At the "**ONLINE**" state, the server may receive a "logout message" from a client. Upon this event, the server sends back a "logout ack" and the session transits to the "**OFFLINE**" state.
7) Upon receiving a message, the server updates a timestamp of the corresponding session. This timestamp is used to keep a memory of the last time that the client is active. The server also regularly checks this timestamp for each session. If the client is inactive for 1 minute or some time period you configured, the corresponding session

transits to the "**OFFLINE**" state, the token of the session expires, and the server sends a "session reset message" to the client. Note that this message does not need an ack.

8) In any state, if the incoming event does not follow the previous rules, the session on the server transits to the "**OFFLINE**" state and put a line in a log file indicating that an error happens and send the "session reset message" to the client.
9) In any state, if the server receives a "session reset message", the corresponding session directly transits to the initial "**OFFLINE**" state and the server put a line in a log file indicating the session is destroyed.

The code structure is similar to the client. However, all events are from the network.

```
#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>

#include <time.h>

struct header {

    char magic1;
    char magic2;
    char opcode;
    char payload_len;

    uint32_t token;
    uint32_t msg_id;
};

const int h_size = sizeof(struct header);

// These are the constants indicating the states.
// CAUTION: These states have nothing to do with the states on the client.
#define STATE_OFFLINE          0
#define STATE_ONLINE           1
#define STATE_MSG_FORWARD      2
// Now you can define other states in a similar fashion.

// These are the events
// CAUTION: These events have nothing to do with the states on the client.
#define EVENT_NET_LOGIN                 80
#define EVENT_NET_POST                  81
// Now you can define other events from the network.
......
#define EVENT_NET_INVALID               255
```

```
// These are the constants indicating the opcodes.
// CAUTION: These opcodes must agree on both sides.
#define OPCODE_RESET                    0x00
#define OPCODE_MUST_LOGIN_FIRST_ERROR   0xF0
#define OPCODE_LOGIN                    0x10
// Now you can define other opcodes in a similar fashion.
......


// This is a data structure that holds important information on a session.
struct session {

    char client_id[32]; // Assume the client ID is less than 32 characters.
    struct sockaddr_in client_addr; // IP address and port of the client
                                    // for receiving messages from the
                                    // server.
    time_t last_time; // The last time when the server receives a message
                      // from this client.
    uint32_t token;         // The token of this session.
    int state;        // The state of this session, 0 is "OFFLINE", etc.

    // TODO: You may need to add more information such as the subscription
    // list, password, etc.
};

// TODO: You may need to add more structures to hold global information
// such as all registered clients, the list of all posted messages, etc.
// Initially all sessions are in the OFFLINE state.

int main() {

    int ret;
    int sockfd;
    struct sockaddr_in serv_addr, cli_addr;
    char send_buffer[1024];
    char recv_buffer[1024];
    int recv_len;
    socklen_t len;

    // You may need to use a std::map to hold all the sessions to find a
    // session given a token. I just use an array just for demonstration.
    // Assume we are dealing with at most 16 clients, and this array of
    // the session structure is essentially our user database.
    struct session session_array[16];

    // Now you need to load all users' information and fill this array.
    // Optionally, you can just hardcode each user.

    // This current_session is a variable temporarily hold the session upon
    // an event.
    struct session *current_session;
    int token;
```

```
    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(&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(32000);

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

    // Same as that in the client code.
    struct header *ph_send = (struct header *)send_buffer;
    struct header *ph_recv = (struct header *)recv_buffer;

    while (1) {

        // Note that the program will still block on recvfrom()
        // You may call select() only on this socket file descriptor with
        // a timeout, or set a timeout using the socket options.

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

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




        // Now we know there is an event from the network
        // TODO: Figure out which event and process it according to the
        // current state of the session referred.


        int token = extract_token_from_the_received_binary_msg(...)
        // This is the current session we are working with.
        struct session *cs = find_the_session_by_token(...)
        int event = parse_the_event_from_the_datagram(...)
```

```
    // Record the last time that this session is active.
    current_session->last_time = time();

    if (event == EVENT_LOGIN) {

        // For a login message, the current_session should be NULL and
        // the token is 0. For other messages, they should be valid.

        char *id_password = recv_buffer + h_size;

        char *delimiter = strchr(id_password, '&');
        char *password = delimiter + 1;
        *delimiter = 0; // Add a null terminator
        // Note that this null terminator can break the user ID
        // and the password without allocating other buffers.
        char *user_id = id_password;

        delimiter = strchr(password, '\n');
        *delimiter = 0; // Add a null terminator
        // Note that since we did not process it on the client side,
        // and since it is always typed by a user, there must be a
        // trailing new line. We just write a null terminator on this
        // place to terminate the password string.




        // The server need to reply a msg anyway, and this reply msg
        // contains only the header
        ph_send->magic1 = MAGIC_1;
        ph_send->magic2 = MAGIC_2;
        ph_send->payload_len = 0;
        ph_send->msg_id = 0;


        int login_success = check_id_password(user_id, password);
        if (login_success > 0) {

            // This means the login is successful.

            ph_send->opcode = OPCODE_SUCCESSFUL_LOGIN_ACK;
            ph_send->token = generate_a_random_token();

            cs = find_this_client_in_the_session_array();
            cs->state = ONLINE;
            cs->token = ph_send->token;
```

```
            cs->last_time = right_now();
            cs->client_addr = cli_addr;

        } else {

            ph_send->opcode = OPCODE_FAILED_LOGIN_ACK;
            ph_send->token = 0;

        }

        sendto(sockfd, send_buffer, h_size, 0,
            (struct sockaddr *) &cli_addr, sizeof(cli_addr));



    } else if (event == EVENT_NET_POST) {

        // TODO: Check the state of the client that sends this post msg,
        // i.e., check cs->state.

        // Now we assume it is ONLINE, because I do not want to ident
        // the following code in another layer.

        for each target session subscribed to this publisher {

            char *text = recv_buffer + h_size;
            char *payload = send_buffer + h_size;

            // This formatting the "<client_a>some_text" in the payload
            // of the forward msg, and hence, the client does not need
            // to format it, i.e., the client can just print it out.
            snprintf(payload, sizeof(send_buffer) - h_size, "<%s>%s",
                cs->client_id, text);

            int m = strlen(payload);

            // "target" is the session structure of the target client.
            target->state = STATE_MSG_FORWARD;

            ph_send->magic1 = MAGIC_1;
            ph_send->magic2 = MAGIC_2;
            ph_send->opcode = OPCODE_FORWARD;
            ph_send->payload_len = m;
            ph_send->msg_id = 0; // Note that I didn't use msg_id here.

            sendto(sockfd, send_buffer, h_size, 0,
                (struct sockaddr *) &target->client_addr,
```

```
                    sizeof(target->client_addr));


            }

            // TODO: send back the post ack to this publisher.

            // TODO: put the posted text line into a global list.


        } else if (event == ...) {

            // TODO: process other events

        }

        time_t current_time = time();

        // Now you may check the time of clients, i.e., scan all sessions.
        // For each session, if the current time has passed 5 minutes plus
        // the last time of the session, the session expires.
        // TODO: check session liveliness


    } // This is the end of the while loop

    return 0;
} // This is the end of main()
```

This event-driven method is the typical way to program a server or client using the TCP/IP network, especially when dealing with multiple connections. It is generally called the "**Reactor**" pattern, i.e., the program always waits for events and reacts to an event (it does not actively make a move but only react).

In many applications, there are many event sources other than user input, network, and timer. The states may be quite complicated and the combination of some conditions may trigger a custom defined event. While the program is handling an event, another event can be generated, especially a blocking I/O operation is needed. For example, if your server received a request for downloading a file, the request from the network is an event, and handling this event requires you to generate another event of I/O operation (i.e., reading a file). Note that reading a file is a blocking I/O operation, which means your process may hang up there for a while when the operating system kernel asks the disk to spin and put the data into the main memory from an external bus outside the CPU. Even if you use SSD which does not spin, you still need to wait for the bulk data transmission on the SATA or PCI-E bus. While your process calls a blocking I/O function, it can be put into a blocking state by the operating system kernel, which means it is

not running on the CPU and making any progress until that I/O operation is finished by the operating system kernel. Using this "Reactor", you can not wait for the file I/O to be finished because you are only running one thread for all types of events. If the only thread blocks, even for a fraction of a second, it will not be able to react to other events. Assume you are managing a server that handles thousands of clients with the event loop but blocks from time to time in the event handler, you will see a weird situation that all clients are slowed down, but the CPU usage of your server is very low. That means the server program is not slowed by insufficient hardware resources, but an event handler that slacks on I/O operation. This is not noticeable if there are only a few clients and the I/O speed is fast enough.

Once the program is getting larger and more complicated, writing everything from scratch would consume much more time and demand much more skills for the programmer. Thus, we rely on existing event-driven frameworks such as libevent (in C), ACE (in C++), boost asio (in C++), Netty (in Java), etc. With those frameworks, first, you define all possible events and write handlers. Then, you register the events and handlers to the framework. If an event is triggered in a handler, you just call the API of the framework to inject the event.

Usually, the server program is not just large and complicated in lines of code, but also large at runtime, i.e., it needs many server machines to work together to cope with heavy network traffic with millions of requests per second from the globe. Think about how many search requests are sent to Google and how many machines are needed to meet the requests. In this case, we need a set of distributed middleware. For example, there are message passing middleware such as Rabbit/MQ, Apache Kafka; there are mapreduce frameworks such as Hadoop; there are distributed data accessing and indexing middleware such as Google BigTable / GFS, Apache Hive / HBase / Hadoop HDFS, Apache Cassandra, etc. You can have a look at the "Big Data Computing" related classes in our school or online for more information. A trend in recent years is the service-oriented architecture (older) and microserver architecture (more recent). The idea is the encapsulation of segments of business logic and functions into loosely coupled services, deployment of the service inside a container, and alignment of the development team and operation of the services to the business structure. We will not discuss the details of the software architecture, but all of them, all those distributed pieces that make a business running, have the foundation of the TCP/IP network and socket programming.

Deliverables:

1. Please record another short video (less than **ten minutes**) demonstrating the functions like that in the previous assignment, and submit a shareable link of the video. You may need at least three clients (e.g., open three terminals) to demonstrate the functions, where one is the publisher, one has subscribed to the publisher, and the other has not

subscribed to the publisher. **Please narrate in the video by either recording voice or type in the an editor window while doing the recording.**

2. Please also submit your code on Canvas if you do not intend to publish your code on GitHub.

Grading rubrics:

There is 1.5 points demonstrating each of the following function:

1. Login + error handling, i.e., if a client posts a line of text before login, an error is generated and handled, while after login, the line of text can be posted.
2. Logout + error handling, similar to the previous one.
3. Subscribe + posting, i.e., show a publisher and another client, before the client subscribes to the publisher, message feeds can not be seen after the publisher posts a line, while after the client subscribes to the publisher, message feeds can be seen.
4. Unsubscribe + posting, similar to the previous one.
5. Retrieve one line + retrieve 20 lines. You need to post a few lines before the retrieval. These posts are not required to be incorporated in the video demo.
6. Session reset triggered by the client, i.e., the client causes the server to send a reset msg. Note that you need to ask the client to deliberately generate a spurious message to trigger the session reset. You can introduce another event and action upon typing a special text line just for testing.
7. Session reset triggered by the server, i.e., the server causes the client to send a reset msg. Similarly, you need to implement a way to generate a spurious message on the server. You can introduce a new type of message and when the server receives this message, it does this just for testing.
8. Session timeout.

In total there are 12 points.

Task 2: Improve this messaging program (for bonus points).

This task is **optional** and depends on the effort, you may receive up to 10% extra credit for the whole class. You don't need to implement all the listed requirements for the extra credit. Do what you prefer and do as much as you can. Finally, put it in your resume as a network software development project experience.

1) **Message Reliability**: In the previous task, we do not retransmit a message if it is lost. Since it uses UDP, message loss is completely possible. Similarly, we do not retransmit an ack if it is lost. None of the messages is reliably delivered. This can cause a few problems, for example, the client may get stuck in login if the login ack is lost. In task 1, we simply assume all messages are guaranteed to be delivered reliably and we ignored handling such error situations. Also, there are other errors we ignored, for example, what if a sendto() returns a failure? We just assume this will never happen in task 1. However, in reality it is not what we have assumed. As a result, you need to add the timeout and retransmission function in your application layer protocol and provide guaranteed delivery of every message even using UDP. You also need to ask the server to keep track of which message is sent and which ack is expected. Actually we already have an ack for each message since we use the request-response paradigm and the state transition diagram has already done a lot of work for this reliability function. Note that your message is still sent over UDP, and each message is an individual datagram. You can modify the protocol and message format as you like to implement reliable delivery of the message. You also need to reason about your design that your protocol is really reliable.
2) **Posting Pictures**: You can merge the file uploading and downloading in the first part of this assignment so that your client program can both send UDP messages and upload a file. Additionally, you can also implement the file post function. For example, client_a can upload a file xyz.png to the server, and client_b can receive the live feed of that file. This is a convenient way to implement the function of posting a picture instead of just a text line. Note that a file must be transferred reliably, i.e., in TCP or using your own UDP based reliable transport protocol in the application layer. An easy way of implementing file post forwarding is separating the notification and the actual file. The publisher always just put a file on the server using the previously designed TCP based file uploading protocol, and the subscriber will receive a message of "file notification message" sent by the server like the "forward message" implemented still in UDP. In this "file notification message", there is the file name. Once the subscriber client receives this message, it just uses the previously designed TCP based file downloading protocol to retrieve the file. Note that you need to merge the UDP code and the TCP code together and carefully manage the multiple threads.
3) **User Interface**: You can write a GUI instead of typing the message in the command line. You can use the GTK+ library, the QT library, or the WxWidget library for the GUI in C++ on Linux (WxWidget is a cross platform but the socket API needs POSIX compatible systems such as Linux or Mac). This GUI will resemble Facebook Messenger or Twitter.

4) **Security**: In the previous task, we send the password and the token in plain text, which is not secure. Someone might think about SSL or similar heavyweight mechanism. However, it is not necessary. Assume the server and the client share a secret password. You can use simple cryptography mechanisms such as AES encryption and SHA-256 hashing to construct a way to implement secure authentication and encrypt every message so that only the right client and the server can decrypt. Note that the key point here is that both the client and the server share a secret password. Only the right client knows this secret.
5) **Persistence**: You can see in our design, all client information and all messages are kept in the memory of the server. If the server program stops or crashes, nearly everything is lost. Twitter definitely does not work in this way. Hence, you can improve the message persistence by using a database and store necessary information in the database. You can use PostgreSQL, MariaDB, SQLite or other database management systems. You can also use a noSQL database instead.
6) **Availability over the public Internet**: You can put the server on a cloud platform such as Amazon AWS or Google Cloud Platform (used to be Google AppEngine) to make the message service available over the public Internet. Alternatively, you can rent a VPS (or just use a short period trial of some VPS) with a public IP address to host your server so that your client can reach it as long as the client is connected to the Internet. Be careful that your client may be behind a router or firewall that does NAT and hence, you may need to modify your login message to "tunnel" through this NAT enabled device. Otherwise, even if the client can reach the server in the forward direction, the server may not be able to reach the client in the backward direction.
7) **Other Features**: You may propose and implement other non trivial features that can improve the program.

Deliverables:

Please write a one-page document that briefly explains the improvement and record a demo video (less than **ten minutes**) on the extra features you have implemented. **Please do not merge this video with the video for the previous task. This means if you are intending to do the extra work, you still need to finish the basic work first and make the extended one as a separate project. My suggestion is that you should nail the core of the protocol first and then think about extended features next.** Please create an account on GitHub and push your code there, make it an open-source project. Also in the blackboard submission, please include your GitHub source code link. However, if you do not want to release your code on GitHub, please submit the code on Canvas.

Grading rubrics:

There is a 5% extra credit (equivalent to five points in other assignments) for implementing one of the previously listed improvement features. You can get 10% extra credit if you have implemented two or more. However, the total amount of extra credit is 10%, including the extra

credit of this programming assignment and the presentation in class. This means if you presented in class and get 5% extra credit, then implemented two extra features in this assignment and get 10% extra credit, the total extra credit will be saturated at 10%. However, if you are not selected for presentation in class, you can still try to devote some effort here to get the maximum amount of possible extra credit. I understand that many programmers are not good at speaking in public, and hence, you have an equal chance of winning the extra credit by presentation or no presentation. You may be able to get A+ in this class with the extra credit.

Also, I would like to encourage you to work on the extra credit part and eventually make it a project experience that you can show on your resume. Last year when I taught this class, one of the students finished two extra features (the GUI + posting pictures, which are the most popular ones) and created a downloadable Java application by putting the server in the Amazon cloud. He got a job offer from a top-tier telecommunication and software company and he said it was this project experience that made the interviewers made the decision (I don't know whether he did a live demo of the program but I believe if you can do a live demo of this application it will definitely help you to increase the chances to get a job offer).

(continued in the next page)

Task 3. Answer the following questions (no hands-on lab or coding required):

Note that this task is not optional.

A Wireshark capture of synthetic TCP traffic between PC1 (10.0.5.11/24) the client, and PC2 (10.0.5.22/24) the server, follows. Use it to answer the following questions.

```
No. Time    Source    Dest.     Protocol Info
 1 0.000000 10.0.5.11 10.0.5.22 TCP 3062>4444 [SYN] Seq=4012935996 Ack=0 Win=5840 Len=0
 2 0.000285 10.0.5.22 10.0.5.11 TCP 4444>3062 [SYN, ACK] Seq=3987339890 Ack=4012935997 Win=5792 Len=0
 3 0.000345 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012935997 Ack=3987339891 Win=5840 Len=0
 4 0.000940 10.0.5.11 10.0.5.22 TCP 3062>4444 [PSH, ACK] Seq=4012935997 Ack=3987339891 Win=5840 Len=1024
 5 0.001116 10.0.5.11 10.0.5.22 TCP 3062>4444 [PSH, ACK] Seq=4012937021 Ack=3987339891 Win=5840 Len=1024
 6 0.002851 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012937021 Win=7168 Len=0
 7 0.002939 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012938045 Ack=3987339891 Win=5840 Len=1448
 8 0.002952 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012939493 Ack=3987339891 Win=5840 Len=1448
 9 0.003027 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012938045 Win=9216 Len=0
10 0.003051 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012940941 Ack=3987339891 Win=5840 Len=1448
11 0.003060 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012942389 Ack=3987339891 Win=5840 Len=1448
12 0.008083 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012939493 Win=11584 Len=0
13 0.008175 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012943837 Ack=3987339891 Win=5840 Len=1448
14 0.008187 10.0.5.11 10.0.5.22 TCP 3062>4444 [FIN, PSH, ACK] Seq=4012945285 Ack=3987339891 Win=5840 Len=952
15 0.008147 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012940941 Win=14480 Len=0
16 0.008251 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012942389 Win=17376 Len=0
17 0.008646 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012943837 Win=20272 Len=0
18 0.011128 10.0.5.22 10.0.5.11 TCP 4444>3062 [ACK] Seq=3987339891 Ack=4012945285 Win=23168 Len=0
19 0.011810 10.0.5.22 10.0.5.11 TCP 4444>3062 [FIN, ACK] Seq=3987339891 Ack=4012946238 Win=26064 Len=0
20 0.011879 10.0.5.11 10.0.5.22 TCP 3062>4444 [ACK] Seq=4012946238 Ack=3987339892 Win=5840 Len=0
```

Note that I put an image here for formatting. It is also shown as a table after the questions on the last page. The table might be more helpful.

(a) How can you identify the packets involved in opening the TCP connection? What is the initial sequence number (ISN) of the TCP client and the TCP server (Hint: there is one ISN on each direction)?

(b) What is the sequence number used in the first byte of application data sent from the TCP client to the TCP server?

(c) Determine the values of the receiving window sizes for the TCP client and the TCP server. How do they change? Note that TCP is full-duplex, there is a receiving window in each direction.

(d) How many packets are transmitted by PC1 and how many packets are transmitted by PC2? Is there any retransmission of a TCP segment (with actual data)? Are there any duplicate ACKs?

(e) Inspect the TCP headers. How many types of flags do you observe (such as ACK)? What do they mean?

(f) How can you identify the packets that are involved in closing the TCP connection? Which end can initiate the close?

(g) What does it mean for the TCP connection to be full duplex?

Deliverables:

1. Please briefly answer each question.
2. Please submit a PDF file containing the answers on Canvas, like that in assignment 1.

Grading rubrics:

There are two points for this task.

No.	Time	Source	Dest.	Protocol Info	Flags	
1	0.000000	10.0.5.11	10.0.5.22	TCP 3062>4444	[SYN]	Seq=4012935996 Ack=0 Win=5840 Len=0
2	0.000285	10.0.5.22	10.0.5.11	TCP 4444>3062	[SYN,ACK]	Seq=3987339890 Ack=4012935997 Win=5792 Len=0
3	0.000345	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012935997 Ack=3987339891 Win=5840 Len=0
4	0.000940	10.0.5.11	10.0.5.22	TCP 3062>4444	[PSH,ACK]	Seq=4012935997 Ack=3987339891 Win=5840 Len=1024
5	0.001116	10.0.5.11	10.0.5.22	TCP 3062>4444	[PSH,ACK]	Seq=4012937021 Ack=3987339891 Win=5840 Len=1024
6	0.002851	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012937021 Win=7168 Len=0
7	0.002939	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012938045 Ack=3987339891 Win=5840 Len=1448
8	0.002952	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012939493 Ack=3987339891 Win=5840 Len=1448
9	0.003027	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012938045 Win=9216 Len=0
10	0.003051	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012940941 Ack=3987339891 Win=5840 Len=1448
11	0.003060	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012942389 Ack=3987339891 Win=5840 Len=1448
12	0.008083	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012939493 Win=11584 Len=0
13	0.008175	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012943837 Ack=3987339891 Win=5840 Len=1448
14	0.008187	10.0.5.11	10.0.5.22	TCP 3062>4444	[FIN,PSH, ACK]	Seq=4012945285 Ack=3987339891 Win=5840 Len=952
15	0.008147	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012940941 Win=14480 Len=0
16	0.008251	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012942389 Win=17376 Len=0
17	0.008646	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012943837 Win=20272 Len=0
18	0.011128	10.0.5.22	10.0.5.11	TCP 4444>3062	[ACK]	Seq=3987339891 Ack=4012945285 Win=23168 Len=0
19	0.011810	10.0.5.22	10.0.5.11	TCP 4444>3062	[FIN,ACK]	Seq=3987339891 Ack=4012946238 Win=26064 Len=0
20	0.011879	10.0.5.11	10.0.5.22	TCP 3062>4444	[ACK]	Seq=4012946238 Ack=3987339892 Win=5840 Len=0