CSE 434 Computer Networks
(Fall 2018) Assignment 3 (part two)

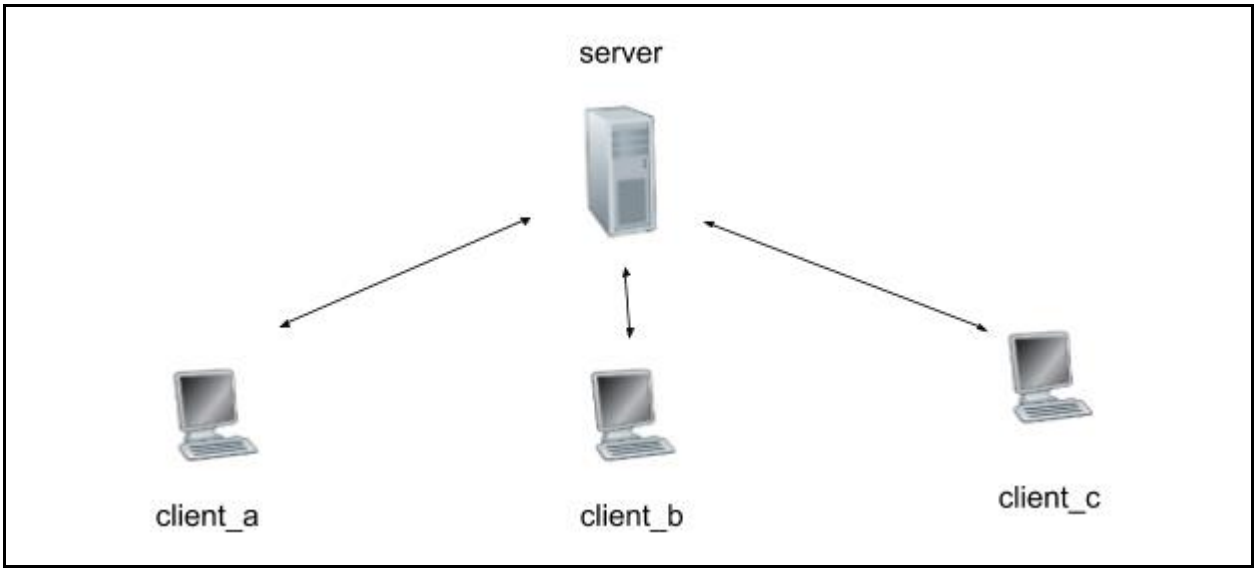
Duo Lu <duolu@asu.edu>

This assignment has two part. This is the second part, due at 11:59 pm on November 28, 2018. Please start early so that you can have time to prepare your finals for other classes. This is a programming assignment, which is designed to allow you to gain hands-on experience in Linux socket programming. The grading is effort-based. You can discuss with your classmate or use any resource available online. However, you are required to write your own code.

Note that in this assignment you can reuse and augment the code you wrote for the first part of assignment 3. No need to write everything from scratch. Alternatively, you may use a programming language other than C/C++, i.e., we do not restrict the usage of C/C++ and the Linux system anymore.

1. UDP based message sending and receiving.

In the first part of this assignment, you can communicate with the server using a UDP protocol invented by you. However, in that protocol, the server only echoes back whatever you sent. In the second part, you are going to use the server as a "hub" to allow clients on the "spoke" to communicate with each other, and every message is forwarded by the server. See the following instructions and the sequence diagram after the instructions for the details.



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. You can type a line "client_a->server#login<password>". Here "client_a" is the client ID and "password" is the password of this client. Each client has a unique ID. 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 "duolu->server#login<123456>" if my ID is "duolu" and my password "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 to the server in a UDP datagram. Note that the "red" lines are typed, not the message sent. Instead, the "blue" lines are sent.
4. The server keeps a list of all users with their passwords, stored in plain text in a file. At the start of the server program, it reads each line into the memory and stores them in an array. 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 flexible in the real product.**
5. When the server received the line "client_a->server#login<password><ip><port>", it compare with the user information read from the file, and authenticate the user. If the password matches, the server sends back a line "server->client_a#Success<XYZABC>", and if the password does not match, the server sends back "server->client_a#Error: password does not match!". Here "XYZABC" is a six-character token randomly generated by the server containing letters, numbers, and special characters. It is usually unique for each client. For example, it can be something like "05&a7F". The "ip" and "port" in the login message is the IP address and port number of the client, and the client program will receive an incoming message from that address and port number. The server needs to remember the token, the IP address, the port number for each client, and it keeps a record on which user is online and which user is offline.
6. Once the client receives the response on a successful login, it also needs to remember the token.
7. You can type a line "client_a->client_b#<XYZABC><msg_id>some_message" on the client, and this line will be sent to the server in a UDP datagram. Here "XYZABC" is the token that the client obtains from the server at login. Generally, the client will put this token in every message that it sends to the server until logoff, as an indication of the login state. The "msg_id" is a randomly generated 10-digit number to uniquely identify a message. Note that both the token and the message ID has fixed length. **You may not need to type the token and the message id if the client software remembers them for you. Instead, the client software can append it**. This means you may just type "client_a->client_b#some_message".
8. Once the server receives "client_a->client_b#<XYZABC><msg_id>some_message" from client_a, the server compares the token "XYZABC" in the message and the token stored on the server at login time of client_a. There are three possible cases.
    a. If they do not match, the server sends back a line "server->client_a#<XYZABC><msg_id>Error: token error!" to client_a.
    b. If they match and client_b is offline (i.e., client_b does not log in or has sent logoff or its login session has expired, which will be discussed later), the server sends back a line "server->client_a#<XYZABC><msg_id>Error: destination offline!" to client_a.
    c. If they match and client_b is also online (i.e., client_b has successfully logged in), the server forward this message to client_b, using the IP address and port number remembered at login of client_b. Then, the server sends back a line "server->client_a#<XYZABC><msg_id>Success: some_message" to client_a. Like that in the first part of this assignment "some_message" is echoed back. Note that this success message does not necessarily indicate that the message is

successfully delivered to client_b. It only indicates that the message is successfully delivered to the server

9. The forwarded message to client_b has the exact format as the message sent by client_a, i.e., "client_a->client_b#<ABCXYZ><msg_id>some_message". However, the token "XYZABC" in this message is substituted by the token "ABCXYZ" generated for client_b at login. The token is used as a secret between each client and the server to authenticate that each message is sent by the legitimate client or the server. No one can spoof a client or the server as long as he or she does not know the token.
10. Once client_b receives the message, it displays it on the screen, and sends back a response "client_b->server#<ABCXYZ><msg_id>Success: some_message". Note that in this message the token "ABCXYZ" is the token of client_b, the "msg_id" is never changed, and the message body is echoed back to the server.
11. You can type a line "client_a->server#logoff" to instruct the client to log off. In this case, the client_a can send a message "client_a->server#logoff<XYZABC>" to the server to log off, and the server sends back a line "server->client_a#Success<XYZABC>". Once the client finishes log off, it will not receive any message from the server.
12. The server regularly checks the last time that a client sends and receives a message. If the client is not active for 5 minutes, the server considers it offline.
13. Note that **the client may need to run two threads**, one monitoring the keyboard and the other keeping receiving messages from the server. **Alternatively, you may use the I/O multiplexing method shown the socket programming lecture**. The server does not need to use multiple threads because UDP is connectionless. In this case, you 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
14. More details will be added to clarify the requirements based on your comments on the document.

Here is a sequence diagram explaining the protocol.

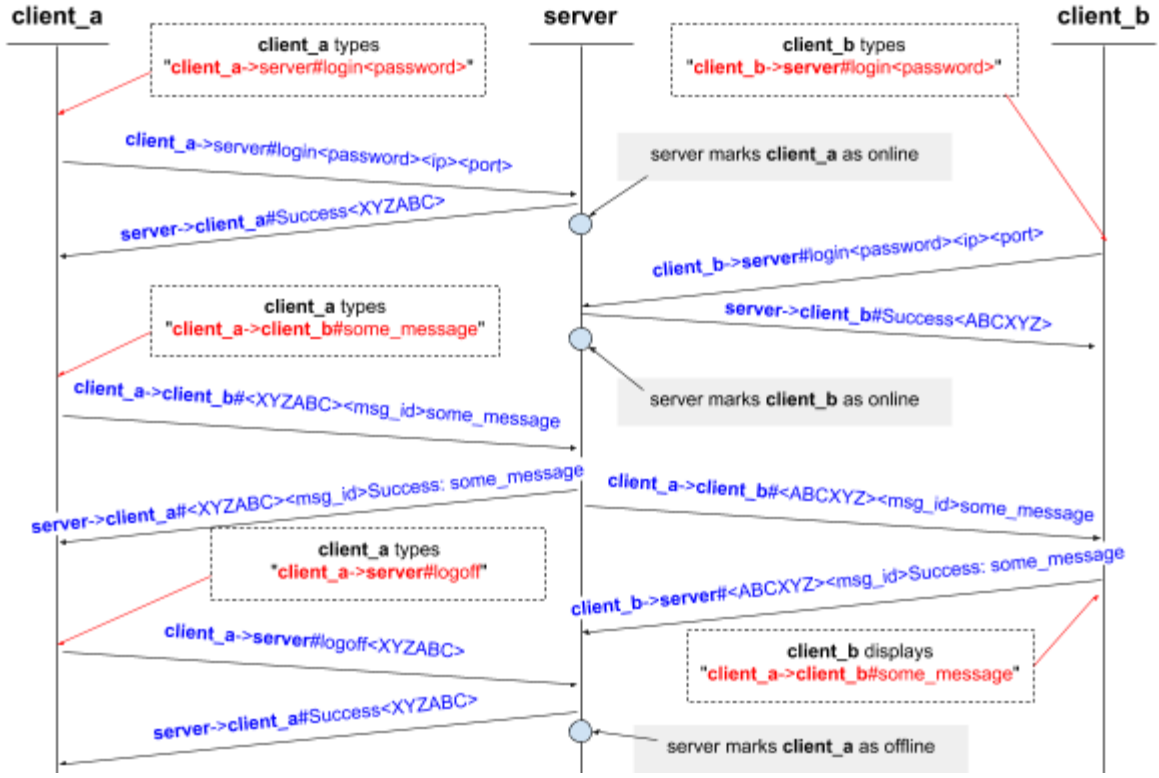


Figure 1: Sequence diagram of the messaging protocol.

Please write a document that briefly explains your program and show a demo with a few screenshots. You also need to submit your code on the blackboard.

Some of you have difficulty in the programming because of the code structure. Here are some detailed instructions.

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 happened before, i.e., they need to declare a few variables to hold the "states" of communication. Typically we call this a "session", i.e., a stateful connection between a client and a server. One client can hold one session to a specific server, but one server needs to hold multiple sessions, one for each client.

In our protocol, a session can have two states, either "online" or "offline". See the following diagram for the client.

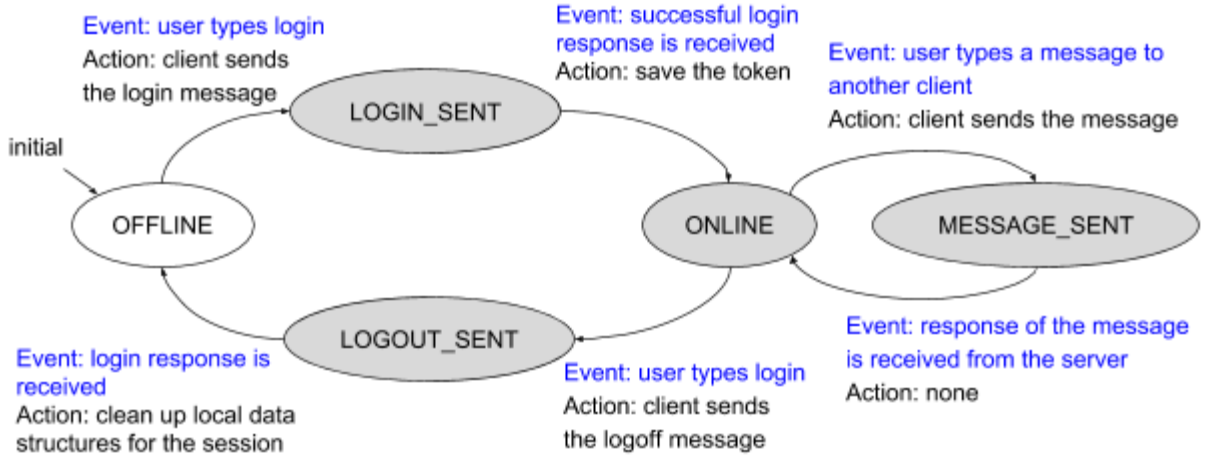


Figure 2: State diagram of a session on the client side

You might wonder what triggers the session state to change. It is the event, i.e., events on the edge of the state diagram. This is similar to the TCP state machine we saw in the lecture. The client may encounter six different types of events from two different sources:

1. Initially, the client software is in the "**OFFLINE**" state.
2. At the "**OFFLINE**" state, the user can types "client_a->server#login<password>", and this event is from the keyboard input. Upon this event, the client software sends the login message like "client_a->server#login<password><ip><port>" and transits to the "**LOGIN_SENT**" state. All other events happen in this state will make the client software hold in this state.
3. At the "**LOGIN_SENT**" state, the client may receive the login response sent back by the server like "server->client_a#Success<XYZABC>" 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 response indicates a failure or an error, it goes back the "**OFFLINE**" state. (Think about what if the server is down and it does not send back any response?) All other events happen in this state will make the client software hold in this state.
4. At the "**ONLINE**" sate, the user can type a line "client_a->client_b#some_message", which is an event from the keyboard input. The corresponding action is sending a message like "client_a->client_b#<XYZABC><msg_id>some_message" to the server. Then, the client software transits to the "**MESSAGE_SENT**" state. If some error happens, e.g., the client may receive some spurious message, the client may go back to the "**OFFLINE**" state. This is a design issue and it depends on you. The protocol does not specify every detail.
5. At the "**MESSAGE_SENT**" state, if the client receives the response 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. Similar to log in, at "**ONLINE**" sate, the user can type a line "client_a->server#logoff", and this event is from the keyboard input. Upon this event, the client software sends the logoff message like "client_a->server#logoff<XYZABC>", and transits to the "**LOGOUT_SENT**" state.

7. At the "**LOGOUT_SENT**" state, the client may receive the log off response from the server like "server->client_a#Success<XYZABC>", 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.

For **a single client session between client_a and the server**, there are five 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, like "client_a->server#login<password><ip><port>", and the session corresponding to the client_a transits to the "**LOGIN_RECEIVED**" state. Note that only the session regarding to client_a changes states. Other sessions are irrelevant to this message from client_a.
3. At the "**LOGIN_RECEIVED**" state, the server compares the client ID and the password. If they match with the stored information on the server, the server generates a token and sends back a response like "server->client_a#Success<XYZABC>". Then, the session transits to the "**ONLINE**" state. If the authentication is unsuccessful, the server sends back a response like "server->client_a#Error: password does not match!" and the session transits back to the "**OFFLINE**" state.
4. At the "**ONLINE**" state, the server may receive a message from client_a targeting client_b. In this case, the server immediately sends back a message to client_a like "server->client_a#<XYZABC><msg_id>Success: some_message", and the session of client_a does not change state (i.e., remain at the "**ONLINE**" state). The message forwarding may not be successful, and the response may indicate a failure. But the session state of client_a does not change.
5. Meanwhile, upon receiving a message from client_a targeting client_b, the server forwards the 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).
6. At the "**MESSAGE_FORWARD**" state, if the server receives a response message from client_b like "client_b->server#<ABCXYZ><msg_id>Success: some_message", the session of client_b transits back to the "**ONLINE**" state. Note that there is a tricky case that multiple other 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 can still forward a message to client_b. However, the server needs to track which message gets the response and which message does not get the response (using the msg_id, which is assumed to be unique). Once all messages get the responses, the session transits back to the "**ONLINE**" state.
7. At the "**ONLINE**" state, the server may receive a logoff message from a client, like "client_a->server#logoff<XYZABC>". Upon this event, the server sends back a response like "server->client_a#Success<XYZABC>" and the session transits to the "**OFFLINE**" state.
8. Alternatively, the server regularly checks the last time that a message is received for each session. If the client is inactive for 5 minutes, the corresponding session transits to the "**OFFLINE**" state, and the token expires. Note that the client may not know this because nothing is sent to notify the client. The client may still send a message to the server.

However, from the server's perspective, the client is put in the "**OFFLINE**" state, and the server will send back a response indicating the client must log in again.

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. There is a similar pattern called "Proactor" that relies on asynchronous I/O instead of I/O multiplexing (the Proactor pattern is used less widely, which will not be discussed).

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 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 the 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 the development 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.

2. Improve this instant message program (optional).

The second task is optional and depends on the effort, you may receive up to 8% extra credit for the whole class. This task is specially designed for those students who are not selected to present and receive the 8% extra credit for the presentation so that you can put extra effort to earn extra credit. However, if you are selected to present, you can still finish this task and receive the extra credit if you want. You don't need to implement all the listed requirement for the extra credit. Do what you like and do as much as you can. Finally, put it in your resume as a network software development project experience.

1. In step 7 to 10 of the previous tasks, we do not retransmit a message if it is lost. Similarly, we do not retransmit a response if it is lost. None of the messages is reliably delivered. You can add the timeout and retransmission function in your application layer protocol and provide guaranteed delivery of every message. Note that your message is still sent over UDP, and each message is an individual datagram. The response that echoes the message back can be regarded as a way of acknowledgment. You can design the protocol as you like to implement reliable delivery of the message.
2. You can write a GUI instead of typing the message in the command line. If you choose to use C/C++ on Linux, you can use the GTK+ library or the QT library for the GUI. If you choose to use a different language or on a different platform, you have more choices to implement the GUI part.
3. You can merge the file uploading part in the first part of this assignment so that your client program can both send UDP message and upload a file. Additionally, you can also implement the file downloading function. For example, client_a can upload a file xyz.txt to the server, and client_b can download that file. Moreover, can you design and implement a simple protocol that allows the server to forward a file like forwarding a UDP based message? **Note that a file must be transferred reliably**, i.e., in TCP or using your own UDP based reliable transporting protocol in the application layer.
4. In step of 3 and step 5, 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. You may propose and implement other non trivial features that can improve the program.

Please write a one-page document that briefly explains the improvement and record a 3-minute demo video if want to get the extra credit. You do not need to submit the code. Instead, 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. **If you do not want to release your code on GitHub, please submit the code on the blackboard.**