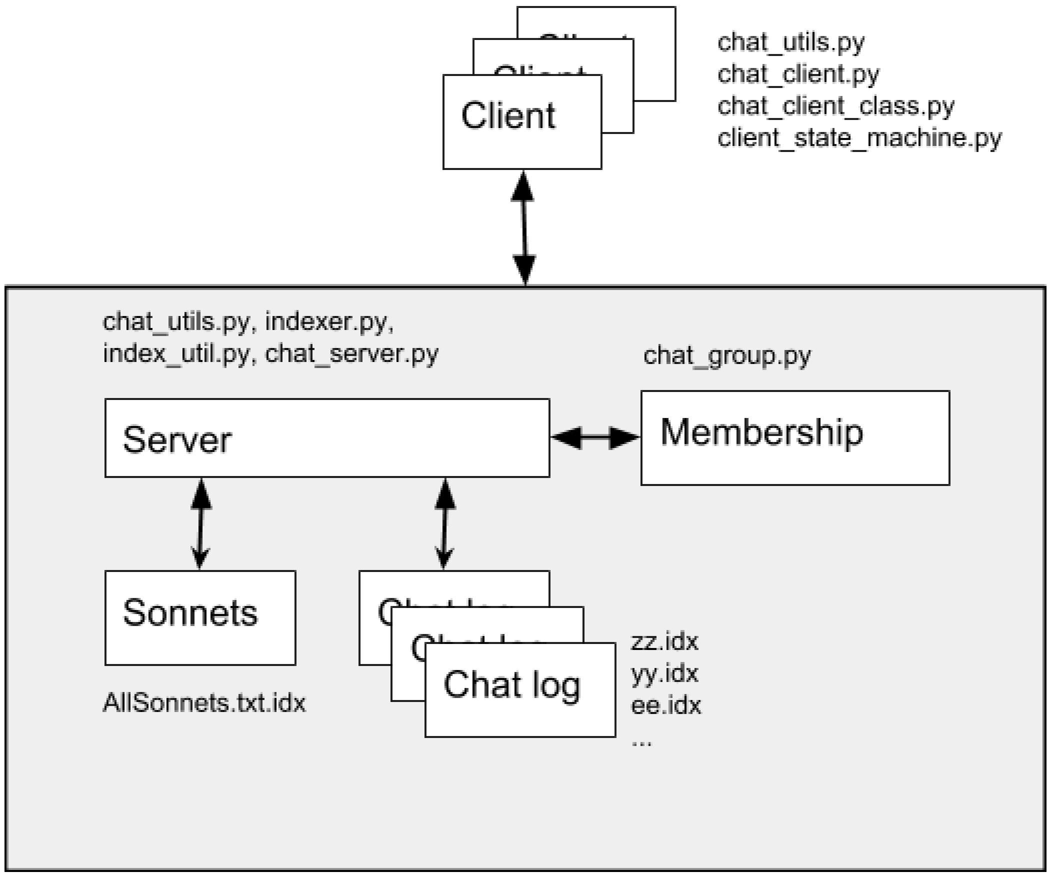
NYU-Shanghai ICS Chat System: Spec and Implementation Guide

Instructions:

- Unit Project 1: indexer
- Unit Project 2: group management
- Unit Project 3: total two weeks
    - Week 1: Client side state machine
    - Week 2: Server side message handling; Integration

Chat system architecture



The overall architecture and main components of our chat system is shown above, along with the files that make up the system.

This is a typical *distributed client-server system*, where multiple clients interact with a central server. Conceptually, this is how wechat is constructed. Clients interact with each other *as if* directly, what actually happens is, however, the server is passing messages back and forth, and adds other functionalities (such as indexing history).

There can be multiple clients, each of them is either idle, or actively participates in one chat session with a group of other clients. Think of a client as an ordinary user of WeChat. Our system is simple: **it allows chatting in one group only**.

The server has a few extra modules:
- A membership management module, to look at who is chatting with whom, for example.
- A chat log, one per user. This allows a user to search her past chatting history with keywords.
- A sonnet database, so a user can ask for a poem when she is not chatting.

Files that make up the system, client side:

- **chat_client.py**, **chat_client_class.py**: both are *given*. No need to change it; indeed, change at your risk! :)
- **chat_state_machine.py**: handles main events interacting with the chat system. YOU implement it.

Files that make up the system, server side

- **chat_server.py**: part of the code is given. YOU need to implement an event handling function.
- **indexer.py**: indexes messages and sonnets. You have implemented it in UP1.
- **AllSonnets.txt, roman.txt.pk**: sonnets and roman-to-numeral conversion, given.
- **chat_group.py**: membership handling. You have implemented in UP2.

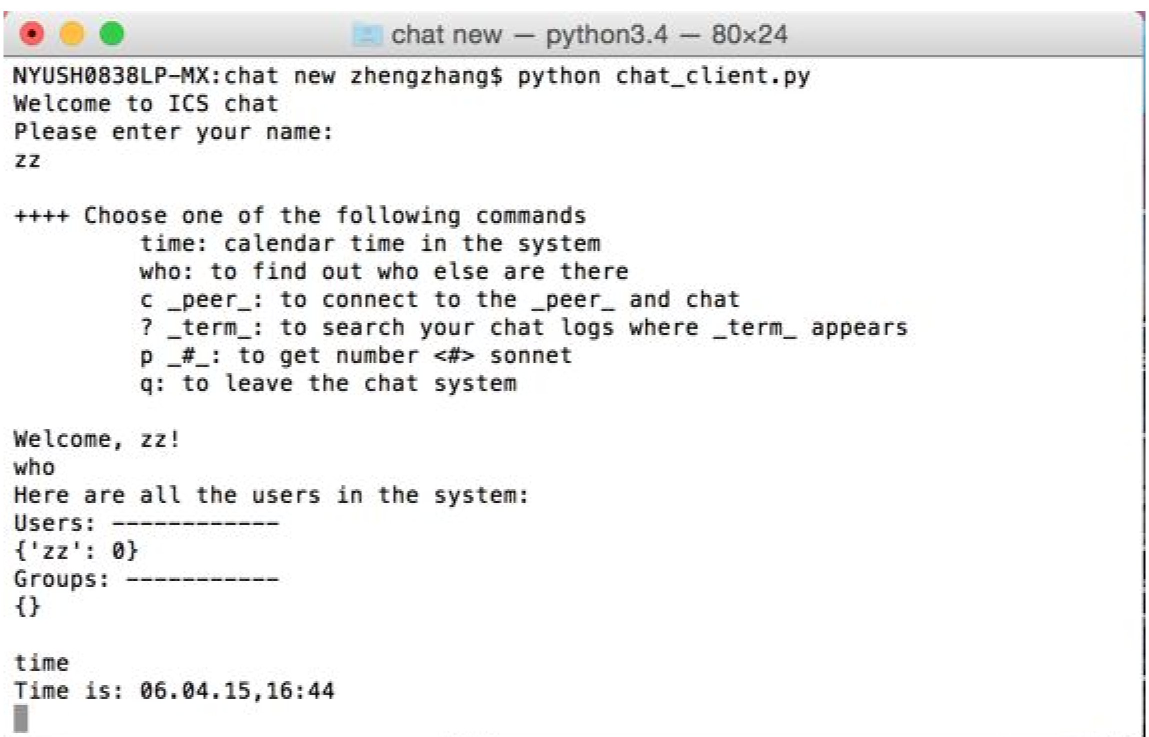
index_util.py and **chat_util.py** are utility files/modules we provide.

When completed, you can run it as the followings:

- On one console: “python python_server.py”. This starts the server.
- On another console: “python python_client.py”. This starts a client

See appendix on how to run chat clients and server on separate machines.

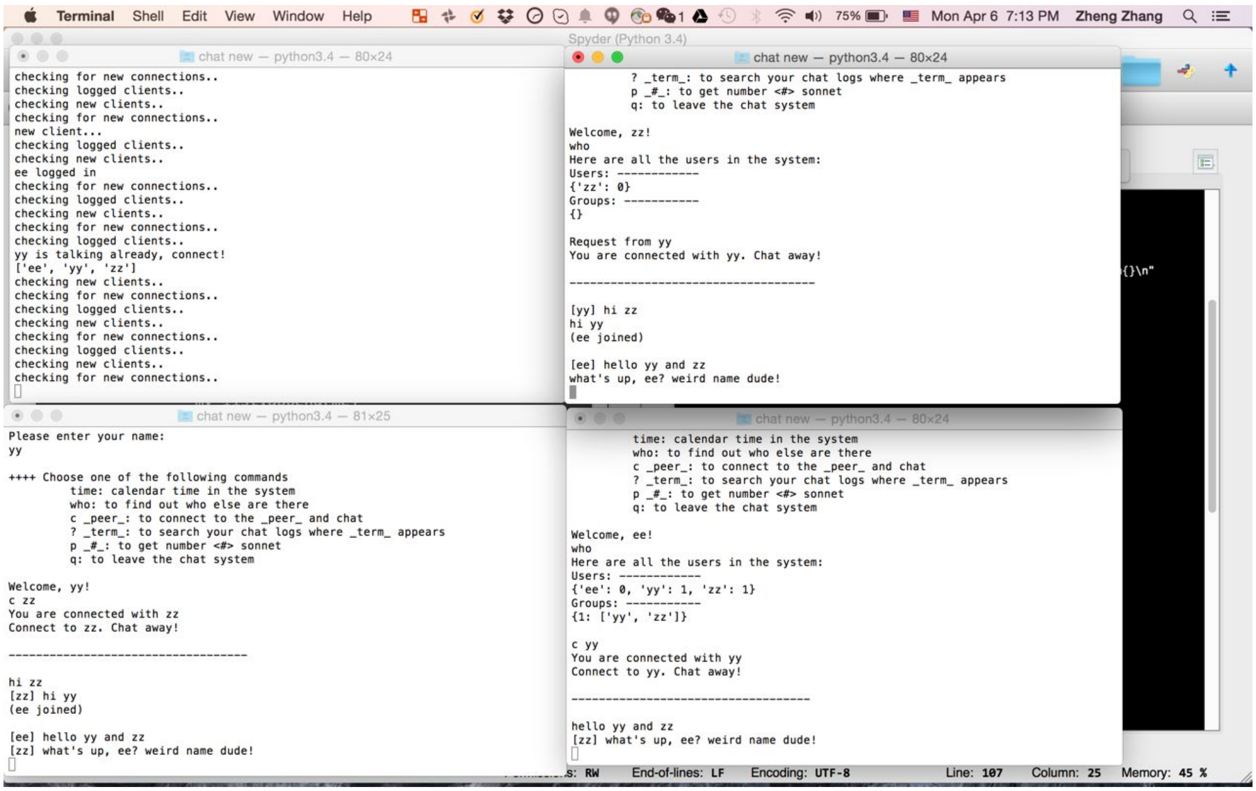
When client starts, it will ask you for a user name, once you enter it, you are logged in. Then the user follows the instructions. Here is one screenshot at the client:



Below is a screenshot how chats start, the sequence is:

- zz (upper-right) joins first; issues a “who”: he’s the only one
- yy (lower-left) joins next: connects to zz
- ee (lower-right) joins last: and connects to zz and therefore joins the group conversation

Upper-left is the screenshot of the server.

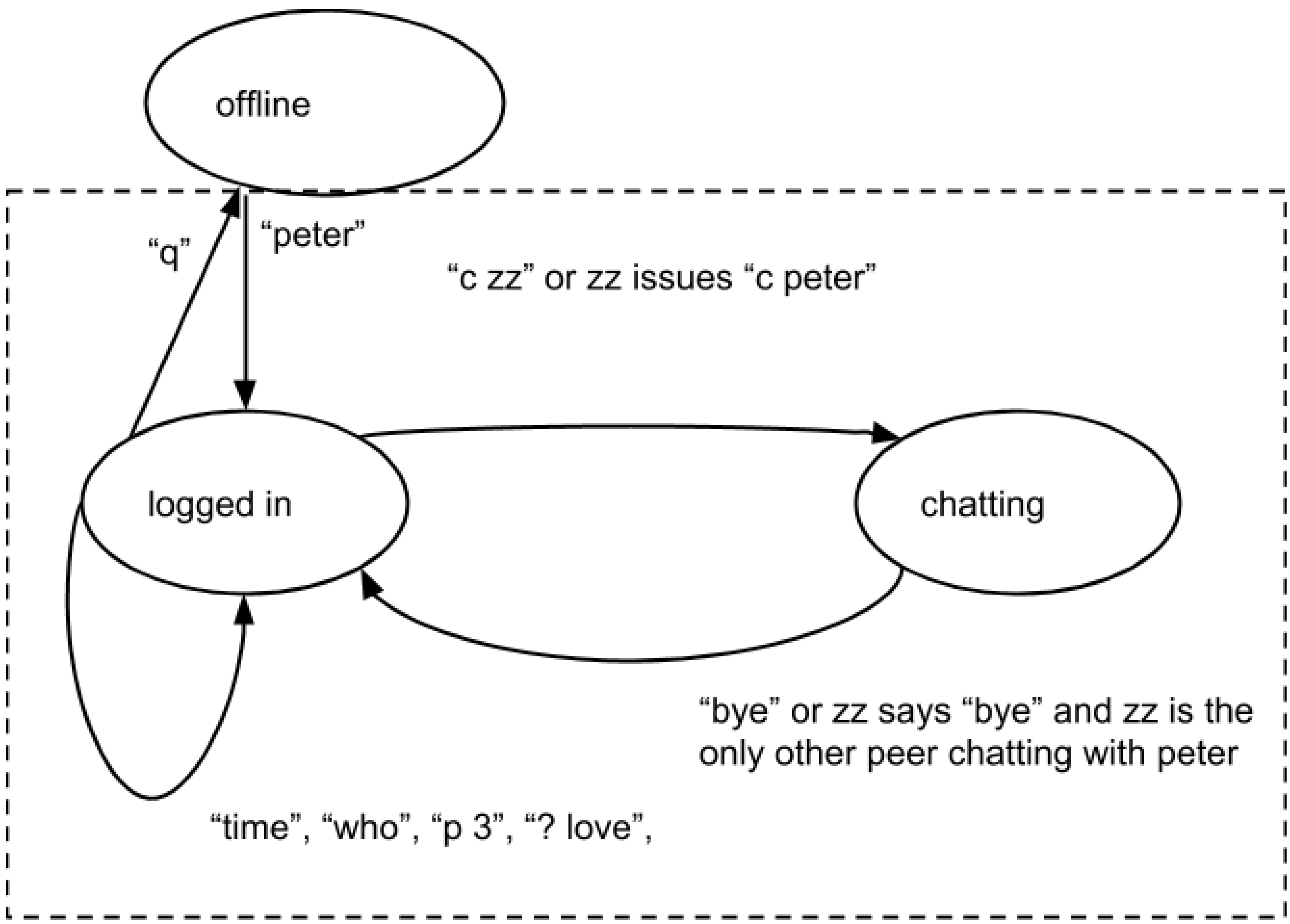


Backgrounds: state machine

Take a quick look at the wiki page: http://en.wikipedia.org/wiki/Finite-state_machine

The basic step of a state machine is to move from one state to the other, following some event. The transition might generate some actions. It will become more clear when we describe how to implement client_state_machine.py.

Below is a simplified state machine for our chat client. The dashed box is logics of client_state_machine.py.



Protocol code

Just like people must share a common language (or a set of symbols) in order to communicate, client and server share a set of codes so they understand the intent of a request.

The following table summarizes the messages between client and server. Each message (e.g. {"action":"login", "name":"zz"}) is a dictionary, when exchanged between server and client, they are packed /unpacked by json.dump/json.load.

Demo: An example of how json works can be find in the provided demo folder.

Action in state S_OFFLINE   Action in state S_LOGGEDIN   Action in state S_CHATTING

Client Socket Message	Request	Server Response
{"action":"login", "name":"zz"}	Client to server (when user inputs his nickname)	{"action":"login", "status":"ok"} : login successful {"action":"login", "status":"duplicate"}: name already exists
{"action":"time"}	Client to server (when user type: "time")	Respond with string encoding the time
{"action":"list"}	Client to server (when user type: "who")	Respond with the members and the chat groups in the system
{"action":"search", "target":'love'}	Client to server (when user type: "? love")	Respond with chat history the chats that contains 'love'
{"action":"poem", "target":"3"}	Client to server (when user type: "p 3")	Respond with sonnet #3 (or III)
{"action":"connect", "target":"peter"}	Client to server (when user type: "c peter")	{"action":"connect", "status":"success"}: zz connects to peter successfully Server should let peter (and others) know zz joined. {"action":"connect", "status":"busy"}: peter is busy (unused) {"action":"connect", "status":"self"}: when a user tries to connect to himself {"action":"connect", "status":"no-user"}: peter is not online
{"action":"connect", "status":"request", "from":"peter"}	Server to client (when peer type: "c user")	peer what to chat with user, server got message {"action":"connect", "target":"zz"}
{"action":"exchange","message": 'hi',"from":"zz" }	Client to server (when user type: "hi")	No response, just pass the text (i.e. 'hi') to every peer in zz's group

{"action":"disconnect"}	Client to server (when user type: "bye")	No response needed; zz gets off the chat group
{"action":"disconnect"}	Server to client (when the other user types "bye" and I am the only one left)	Sent when peter's only partner, zz, has left the chat group

Modules

The followings describe each module, and provide implementation guides when applicable. There are quite a few modules. However, you have done two critical ones already, and you only need to implement two functions, one at the client, another at the server.

Indexer and Group management

These are covered in UP1 and UP2.
Indexer:

- The class PIndex stores and indexes the sonnets
- The class Index indexes chats among clients, and responds to searches.

Group management:

- Records when a peer joins and leaves the system
- Respond to query of members in the system (via “who” command issued from the client)
- Let a peer connect to another (e.g. “c zz”)
- Let a peer quit a group (via “bye” from the client)

Utility functions

chat_utils.py is imported as a module. It has a few things worth mentioning:

```
6   CHAT_IP = socket.gethostbyname(socket.gethostname())
7   CHAT_PORT = 1112
8   SERVER = (CHAT_IP, CHAT_PORT)
9
10  menu = "\n++++ Choose one of the following commands\n \
11          time: calendar time in the system\n \
12          who: to find out who else are there\n \
13          c _peer_: to connect to the _peer_ and chat\n \
14          ? _term_: to search your chat logs where _term_ appears\n \
15          p _#_: to get number <#> sonnet\n \
16          q: to leave the chat system\n\n"
```

Line 6-8 gives the server address and port when a client connects to it. You don't need to worry about it. For now, the server runs on the same machine as a client. Later we will extend how to connect to a server running on a different machine.

```
S_OFFLINE   = 0
S_CONNECTED = 1
S_LOGGEDIN  = 2
S_CHATTING  = 3
```

The above are four states a client can be in. In fact, in our current implementation, we will not use S_CONNECTED.

A note on sockets:

For programs to talk to each other over the internet, they use *socket*. This is an advanced topic we will not cover. For now, think of socket as the telephone you dial in order to talk to your friend. We provide two utility routines:

- mysend(s, msg) takes a string *msg* and sends down a socket *s*.
- myrecv(s) returns a string in *msg*.

Our codes have already set up the sockets, so you don't have to implement them.

Demo: You can find 4 simple demo files in the demo folder.

client_demo.py

client_demo_multi_client.py

server_demo.py

server_demo_multi_client.py

UP3 Part 1: Client-side statement machine

You only need to modify client_state_machine.py; advanced students are encouraged to read chat_client.py (which is the main entry) and chat_client_class.py. We have already handled login, logout, setting up the connections etc. in these two files. When chat_client initializes, it will has a member of the class ClassSM, after login, it will enter S_LOGGEDIN, and that is where your work starts. That is, *proc* is only called *after* the client is at state S_LOGGEDIN.

```python
    def proc(self, my_msg, peer_msg):
        self.out_msg = ''
#==============================================================================
# Once logged in, do a few things: get peer listing, connect, search
# And, of course, if you are so bored, just go
# This is event handling instate "S_LOGGEDIN"
#==============================================================================
        if self.state == S_LOGGEDIN:
            # todo: can't deal with multiple lines yet
            if len(my_msg) > 0:

                if my_msg == 'q':
                    self.out_msg += 'See you next time!\n'
                    self.state = S_OFFLINE

                elif my_msg == 'time':
                    mysend(self.s, json.dumps({"action":"time"}))
                    time_in = json.loads(myrecv(self.s))["results"]
                    self.out_msg += "Time is: " + time_in
```

This is the function you need to complete. It takes three arguments:

- my_msg: this user's outgoing message
- peer_code, peer_msg: the code and the associated incoming message from its peer.

The output of this function is stored in self.out_msg.

The above code shows the example of handling "q" and "time". The way "time" command is written is typical: send a message through socket, and record anything to be output in self.out_msg.

A client moves between two states in the state machine: S_LOGGEDIN and S_CHATTING. In S_LOGGEDIN, here are the event and actions.

The following table shows state transition while In state S_LOGGEDIN:

Message	Action	**Next state**
From user: "time"	Send `{"action":"time"}` to server. Server responds with a string contains the current clock	S_LOGGEDIN
From user: "q"	Logout from chat system	S_OFFLINE
From user: "who"	Send `{"action":"list"}` to server. Server responds all users and their group info (by calling Group.list_all('user_name') function)	S_LOGGEDIN
From user: "p 3"	Send `{"action":"poem", "target":"3"}` to server. Server responds with sonnet III	S_LOGGEDIN
From user: "? love"	Send `{"action":"search", "target":'love'}` to server. Search all past chats containing keyword "raining"	S_LOGGEDIN
From user: "c peter"	Send `{"action":"connect", "target":"zz"}` to server. Connect to user peter	S_CHATTING
From peer: `{"action":"connect", "target":"zz"}`	Accept the peering request The name of your peer will be determined by socket-name dictionary. You may want to set *self.out_msg* to reflect that you have connected to a peer	S_CHATTING

While at state S_CHATTING:

Message	Action	**Next state**
From user: "this is a good day"	Send to server`{"action":"exchange", "message": 'this is a good day'q}`	S_CHATTING
From user: "bye"	Send to sever `{"action":"disconnect"}`	S_LOGGEDIN
From peer: peer code is `{"action":"disconnect"}`	(server will send this code if my peer has left and I am the only in the current group)	S_LOGGEDIN

UP3 Part 2: Server message handling

The server, to be implemented in chat_server.py, will receive all sorts of messages from clients (messages such as `{"action":"time"}`, `{"action":"disconnect"}`, etc). The main job of the server is to respond to all those messages. To handle them, the Server class maintains quite a number of dictionaries. The most important ones to keep in mind are:

- *self.logged_name2sock*: maps a client's name to its socket
- *self.logged_sock2name*: the reverse of the above; map a socket to the client name
- *self.group*: the group management part, bookkeeping the status of peers in the system
- *self.indices*: maps a client's name to its chat index

```python
class Server:
    def __init__(self):
        self.new_clients = [] #list of new sockets of which the user id is not known
        self.logged_name2sock = {} #dictionary mapping username to socket
        self.logged_sock2name = {} # dict mapping socket to user name
        self.all_sockets = []
        self.group = grp.Group()
        #start server
        self.server=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind(SERVER)
        print(SERVER)
        self.server.listen(5)
        self.all_sockets.append(self.server)
        #initialize past chat indices
        self.indices={}
```

In case you are wondering how the server kicks in, here is the main loop. You don't really need to understand a whole lot of it, but it's nice to have an idea:

```python
#==============================================================================
# main loop, loops *forever*
#==============================================================================
    def run(self):
        print ('starting server...')
        while(1):
           read,write,error=select.select(self.all_sockets,[],[])
           print('checking logged clients..')
           for logc in list(self.logged_name2sock.values()):
               if logc in read:
                   self.handle_msg(logc)
           print('checking new clients..')
           for newc in self.new_clients[:]:
               if newc in read:
                   self.login(newc)
           print('checking for new connections..')
           if self.server in read :
               #new client request
               sock, address=self.server.accept()
               self.new_client(sock)

def main():
    server=Server()
    server.run()

main()
```

Basically, the server loops through each of its sockets, deals with them appropriately, whether it be using them to receiving and handling messages, logging them in, or, in the special case of its own socket, accepting connection requests.

You need to complete the function handle_msg(). Looking at the first few lines of handle_msg(),

```python
#==============================================================================
# main command switchboard
#==============================================================================
    def handle_msg(self, from_sock):
        #read msg code
        msg = myrecv(from_sock)
        if len(msg) > 0:
```

we see that it takes an argument, *from_sock,* the socket of the client sending the message. To actually get the message, we need to use *myrecv(from_sock)*.
Now, based on what that code is, the server will send back a message to the client. In the case of a client connecting and disconnecting, the server's Group object's "connect" and "disconnect" functions will be called.

Your job will be to fill in the (el)if-blocks that handles

- {"action":"exchange", "message": "<a str>"}
- {"action":"list"}
- {"action":"poem", "target": "<sonnet number>"}
- {"action":"search", "target":'love'}

The cases of "action":"list" and "action":"list" are the easiest to start. In both those cases, you only have to send back the appropriate message in a string.
For "action":"poem", you are given an incorrect implementation. Your job is to correct it. Hint: Recall that when the code is "action":"poem", the messaged you received is of the form {"action":"poem", "target":"<sonnet number>"}.

"action":"exchange" will be a bit more challenging.
For "action":"exchange", the message you receive is of the form {"action":"exchange", "message": "<a str>"} where string is what was sent by the client. Send it to everyone in the client's group! Finally, don't forget to index each message. Otherwise, searching (the ? command) won't work.

Running chat client and server on different machines

Fun things to try

Here are some examples that you can do, creatively:

- In the state of S_ALONE, “ping blah blah”, the server responds with “pong blah blah”
- In the state of S_CHATTING, “_flip_ what said is true”, the server sends to the peers “[zz] _flip__ true is said what”