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
    - Part 1: Client side state machine
    - Part 2: Server side message handling; Integration

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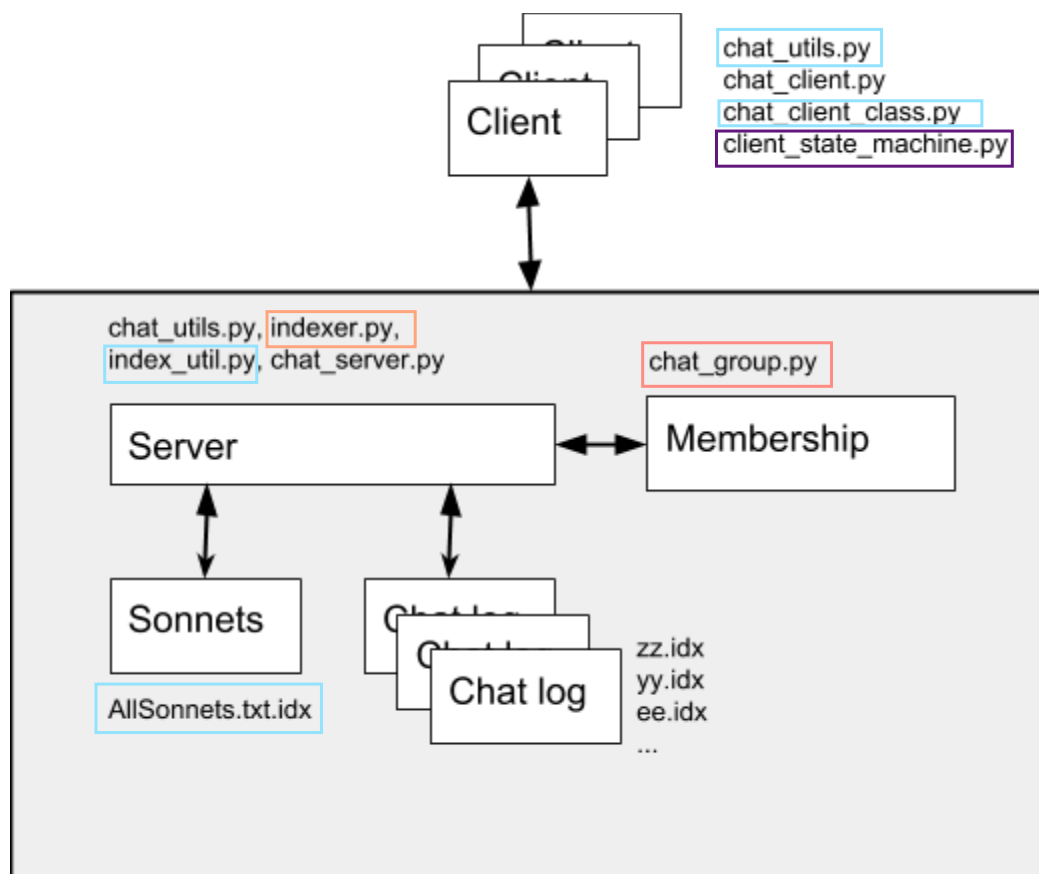
The overall architecture and main components of our chat system is shown above, along with the files that make up the system.

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_cmdl_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 chat_server.py". This starts the server.
- On another console: "python chat_cmdl_client.py". This starts a client

See appendix on how to run chat clients and server on separate machines.
When server starts, you should have this screenshot:

```
➡ Solutions to UP3 pwd # show current directory, make sure you have the file you are running
/Users/wy448/Desktop/chat_system_draft/Solutions to UP3
➡ Solutions to UP3 clear
➡ Solutions to UP3 pwd # show current directory, make sure you are under the right folder path
/Users/wy448/Desktop/chat_system_draft/Solutions to UP3
➡ Solutions to UP3 ls # list all the files, for Windows PC, use "dir"
AllSonnets.txt          chat_client_class.py    chat_utils.py           roman.txt.pk
README.md               chat_cmdl_client.py     client_state_machine.py support
Wen.idx                 chat_group.py           demo                    trash
__pycache__             chat_server.py          indexer.py
➡ Solutions to UP3 python chat_server.py # run the server python file
starting server...
```

Below is one screenshot with a starting server(left) and one client(right):


```
⌘1 python (python3.6)                                              ⌘2 python (python3.6)
__pycache__          chat_server.py        indexer.py            → Solutions to UP3 python chat_cmdl_client.py
→ Solutions to UP3 python chat_server.py # run the server python file   Welcome to ICS chat
starting server...                                                Please enter your name:
checking logged clients..                                         Wen
checking new clients..
checking for new connections..                                    ++++ Choose one of the following commands
new client...                                                             time: calendar time in the system
checking logged clients..                                                 who: to find out who else are there
checking new clients..                                                    c _peer_: to connect to the _peer_ and chat
Wen logged in                                                             ? _term_: to search your chat logs where _term_ appears
checking for new connections..                                            p _#_: to get number <#> sonnet
checking logged clients..                                                 q: to leave the chat system
checking new clients..
checking for new connections..                                    Welcome, Wen!
checking logged clients..                                         time
checking new clients..                                            Time is: 20.11.18,15:25
checking for new connections..                                    who
checking logged clients..                                         Here are all the users in the system:
search for Wen for love                                           Users: ------------
server side search:                                               {'Wen': 0}
checking new clients..                                            Groups: -----------
checking for new connections..                                    {}
checking logged clients..
Wen asks for  2                                                   ?love
here:                                                             'love' not found
 II.

When forty winters shall besiege thy brow,
 And dig deep trenches in thy beauty's field,                     p2
 Thy youth's proud livery so gazed on now,                        II.
 Will be a totter'd weed of small worth held:
 Then being asked, where all thy beauty lies,                     When forty winters shall besiege thy brow,
 Where all the treasure of thy lusty days;                         And dig deep trenches in thy beauty's field,
 To say, within thine own deep sunken eyes,                        Thy youth's proud livery so gazed on now,
 Were an all-eating shame, and thriftless praise.                  Will be a totter'd weed of small worth held:
 How much more praise deserv'd thy beauty's use,                   Then being asked, where all thy beauty lies,
 If thou couldst answer 'This fair child of mine
 Shall sum my count, and make my old excuse,'
 Proving his beauty by succession thine!
 This were to be new made when thou art old,
 And see thy blood warm when thou feel'st it cold.
checking new clients..
checking for new connections..
```

Below is a screenshot how chats start, the sequence is:
- Server(Upper-left) starts
- Wen (upper-right) joins first; issues a "who": he's the only one
- Joyce (lower-left) joins next: connects to Wen
- zz (lower-right) joins last: connects to Joyce and therefore joins the group conversation


```
⌘1 python (python3.6)                           ⌘2 python (python3.6)
checking logged clients..                       Welcome, Wen!
checking new clients..                          who
zz logged in                                    Here are all the users in the system:
checking for new connections..                  Users: ------------
checking logged clients..                       {'Wen': 0}
checking new clients..                          Groups: -----------
checking for new connections..                  {}
checking logged clients..
Joyce is talking already, connect!              Request from Joyce
['zz', 'Joyce', 'Wen']                          You are connected with Joyce. Chat away!
checking new clients..
checking for new connections..                  --------------------------------
checking logged clients..
checking new clients..                          (zz joined)
checking for new connections..
                                                [zz]Hi Joyce and Wen

⌘3 python (python3.6)                           ⌘4 python (python3.6)
        ? _term_: to search your chat logs where _term_ a   Welcome, zz!
ppears                                          who
        p _#_: to get number <#> sonnet        Here are all the users in the system:
        q: to leave the chat system            Users: ------------
                                                {'Wen': 1, 'Joyce': 1, 'zz': 0}
Welcome, Joyce!                                 Groups: -----------
c Wen                                           {1: ['Joyce', 'Wen']}
You are connected with Wen
Connect to Wen. Chat away!                      c Joyce
                                                You are connected with Joyce
--------------------------------                Connect to Joyce. Chat away!

(zz joined)                                     --------------------------------

[zz]Hi Joyce and Wen                            Hi Joyce and Wen
```

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 Peter")
- 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.

```
18   S_OFFLINE   = 0
19   S_CONNECTED = 1
20   S_LOGGEDIN  = 2
21   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

State machine

You need to update user state correctly according to the following graph.

```
18     S_OFFLINE    = 0
19     S_CONNECTED  = 1
20     S_LOGGEDIN   = 2
21     S_CHATTING   = 3
```

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.
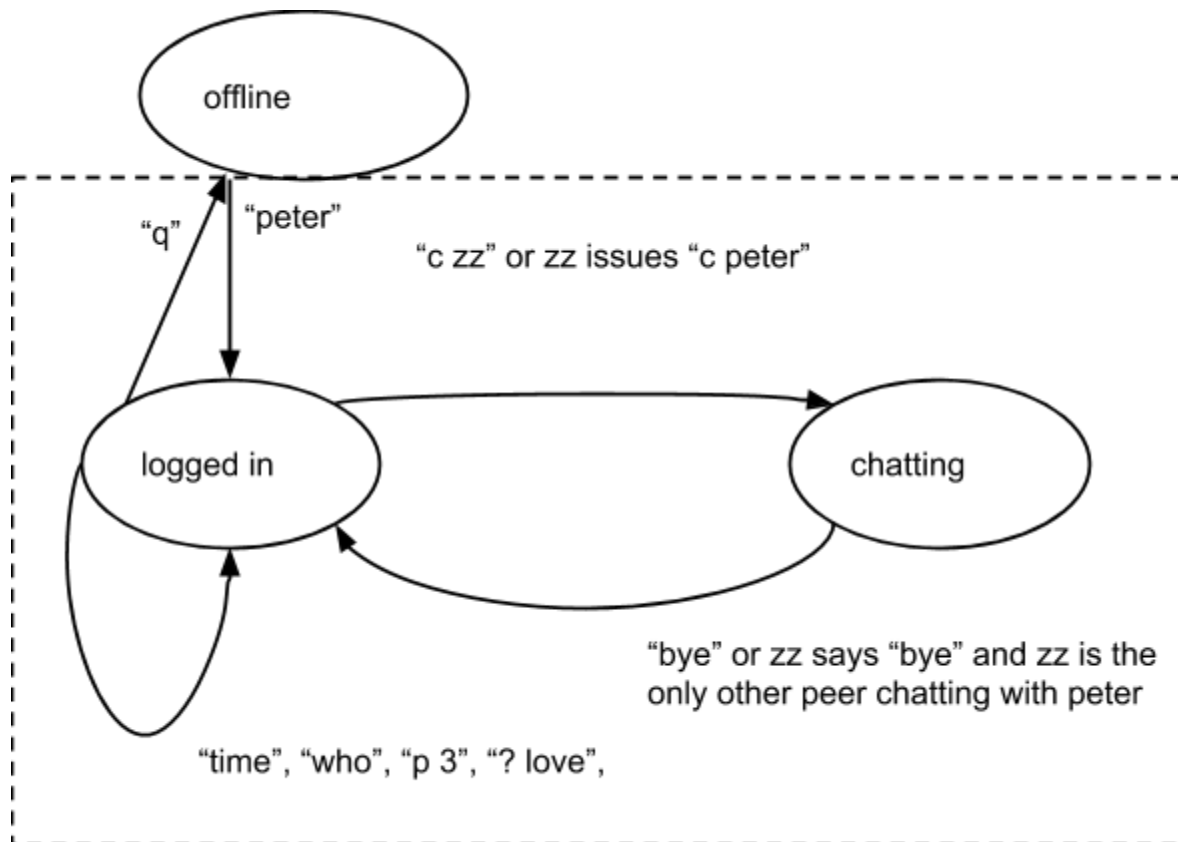
Protocol code

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 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.

All the following message is assumed you logged in as zz.

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

Action in state S_OFFLINE (to S_LOGGEDIN)

When user input: "his nickname",
server response: "login successfully" or "name already exists"

```
"client send": {"action": "login","name": "zz"}
"server respond": { "action": "login",
   "status": ["ok", "duplicate"],
   "msg": ["login sucessfully", "name already exists"]
}
```

Action in state S_LOGGEDIN

When user type: "time",
server responds with string encoding the time

```
"client send": { "action": "time" }
"server respond": { "action": "time", "msg": "13:40:33"}
```

When user type: "who",
server respond with the members and the chat groups in the system

```
"client send": { "action": "list"}
"server respond" {"action": "list", "msg": "{ zz :0, wen :1, cc :1}"}
```

When user type: "? love",
server respond with chat history the chats that contains 'love'

```
"client send": {"action": "search", "target": "love"}
"server respond": { "action": "search", "msg": ["(zz)12:50:01 i love ICS",
"(wen)13:22:22 I don't love ICS"] }
```

when user type: "p3", server respond with sonnet #3 (or III)

```
"client send"{ "action": "poem", "target": "3" }
"server respond"{ "action": "poem", "msg": "III. When forty winters shall besiege
thy brow,...."}
```

Action from S_LOGGEDIN to S_CHATTING

when user zz type: "c peter", server should:

1. let zz know he connects to peter successfully,
2. let peter(and others already chatting with peter) know zz joined
3. or let zz know what cause the unsuccessful connection

```
//Successfully
"client send": { "action": "connect","target": "peter"}
"server respond to zz": {
        "action": "connect", "status": "success", "msg": " connected to peter"}
"server respond to peter": {
        "action":"connect", "status":"request", "from":"zz", msg": "zz joined"}
"server respond to group member 1": {
        "action":"connect", "status":"request", "from":"zz", msg": "zz joined"}
"server respond to group member 2": {
        "action":"connect", "status":"request", "from":"zz", msg": "zz joined"}




//unsuccessfully case 1
"client send": { "action": "connect","target": "zz"}
"server respond to zz": {
        "action": "connect", "status": "self", "msg": "cannot connect to yourself"}

//unsuccessfully case 2
"client send": { "action": "connect","target": "peter"}
"server respond to zz": {
        "action": "connect", "status": "no-user", "msg": "peter is not
online(no-user)"}


```

when Peter(peer) type: "c zz(you)", zz will get connected to the Peter

```
"peer send":{"action": "connect","status": "request","from": "Peter"}
"server send"{ "action": "connect", "target": "zz","msg": "you are connected with
peter" }
```

Action in state S_CHATTING

when user type: "hi"

```
"client send":{ "action":"exchange","message": "hi","from":"zz" }
"server response to zz":
```

```
 No response, just pass the text (i.e. 'hi') to every peer in zz's group
"server response to other group member if exist":
{
  "action":"exchange", "from": "zz", "message": "hi"]
}
```

Action from state S_CHATTING to S_LOGGEDIN

when user type: "bye"

```
"Client send": {"action":"disconnect"}
"server send": No response needed; zz gets off the chat group
```

when the other user types "bye" and I am the only one left

```
"Server respond": { "action":"disconnect", "msg":"everyone left, you are alone"}
```

UP3 Implementation 1: client_state_machine.py

You only need to <mark>modify client_state_machine.py;</mark> 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. <mark>When chat_client initializes, it will has a member of the class ClassSM,</mark> 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
51          def proc(self, my_msg, peer_msg):
52              self.out_msg = ''
53      #===================================================================
54      # Once logged in, do a few things: get peer listing, connect, search
55      # And, of course, if you are so bored, just go
56      # This is event handling instate "S_LOGGEDIN"
57      #===================================================================
58              if self.state == S_LOGGEDIN:
59                  # todo: can't deal with multiple lines yet
60                  if len(my_msg) > 0:
61
62                      if my_msg == 'q':
63                          self.out_msg += 'See you next time!\n'
64                          self.state = S_OFFLINE
65
66                      elif my_msg == 'time':
67                          mysend(self.s, json.dumps({"action":"time"}))
68                          time_in = json.loads(myrecv(self.s))["results"]
69                          self.out_msg += "Time is: " + time_in
70
```

This is the function <mark>you need to complete</mark>. It takes three arguments:
- my_msg: this user's <mark>outgoing message</mark>
- peer_code, peer_msg: the code and the associated incoming message from its peer.

The output of this function <mark>is stored in self.out_msg</mark>.

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.

There are totally 2 <mark>**pass-statements**</mark> in the code to find where you need to complete, as below:

The first pass-statement:

```
107
108                if len(peer_msg) > 0:
109                    peer_msg = json.loads(peer_msg)
110                    if peer_msg["action"] == "connect":
111
112                        # ------------your code here-------#
113                        print(peer_msg)
114                        pass
115
116
117
118                        # ------------end of your code----#
119
120    #========================================================================
121    # Start chatting, 'bye' for quit
```

The second pass-statement:

```
120    #========================================================================
121    # Start chatting, 'bye' for quit
122    # This is event handling instate "S_CHATTING"
123    #========================================================================
124            elif self.state == S_CHATTING:
125                if len(my_msg) > 0:      # my stuff going out
126                    mysend(self.s, json.dumps({"action":"exchange", "from":"[" + self.
127                    if my_msg == 'bye':
128                        self.disconnect()
129                        self.state = S_LOGGEDIN
130                        self.peer = ''
131                if len(peer_msg) > 0:     # peer's stuff, coming in
132
133
134                    # ------------your code here-------#
135                    peer_msg = json.loads(peer_msg)
136                    print(peer_msg)
137                    pass
138
139
140
141                    # ------------end of your code----#
142
143                # Display the menu again
144                if self.state == S_LOGGEDIN:
145                    self.out_msg += menu
146    #========================================================================
147    # invalid state
148    #========================================================================
```

UP3 Implementation 2: chat_server.py

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

```
18    class Server:
19        def __init__(self):
20            self.new_clients = [] #list of new sockets of which the user id is not known
21            self.logged_name2sock = {} #dictionary mapping username to socket
22            self.logged_sock2name = {} # dict mapping socket to user name
23            self.all_sockets = []
24            self.group = grp.Group()
25            #start server
26            self.server=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
27            self.server.bind(SERVER)
28            print(SERVER)
29            self.server.listen(5)
30            self.all_sockets.append(self.server)
31            #initialize past chat indices
32            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:

```
183    #===============================================================
184    # main loop, loops *forever*
185    #===============================================================
186        def run(self):
187            print ('starting server...')
188            while(1):
189                read,write,error=select.select(self.all_sockets,[],[])
190                print('checking logged clients..')
191                for logc in list(self.logged_name2sock.values()):
192                    if logc in read:
193                        self.handle_msg(logc)
194                print('checking new clients..')
195                for newc in self.new_clients[:]:
196                    if newc in read:
197                        self.login(newc)
198                print('checking for new connections..')
199                if self.server in read :
200                    #new client request
201                    sock, address=self.server.accept()
202                    self.new_client(sock)
203
204        def main():
205            server=Server()
206            server.run()
207
208        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(),

```
89    #===============================================================
90    # main command switchboard
91    #===============================================================
92        def handle_msg(self, from_sock):
93            #read msg code
94            msg = myrecv(from_sock)
95            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, there are 5 pass-statements around the `"action":"time"` where you need to complete.

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

```
         mysend(from_sock, json.dumps({"action" poem , results :poem}))
179   #===============================================================
180   #              time
181   #===============================================================
182         elif msg["action"] == "time":
183             ctime = time.strftime('%d.%m.%y,%H:%M', time.localtime())
184             mysend(from_sock, json.dumps({"action":"time", "results":ctime}))
185   #===============================================================
186   #          search: : IMPLEMENT THIS
187   #===============================================================
```

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.

Order of implementation:

1. chat_server.py:

```python
elif msg["action"] == "poem":
    pass
```

2. chat_server.py

```python
elif msg["action"] == "list":
    pass
```

3. chat_state_machine.py (state = logged_in)

```python
if len(peer_msg) > 0:
    peer_msg = json.loads(peer_msg)
    if peer_msg["action"] == "connect":
        pass
```

4. chat_state_machine.py (state = chatting)chat_server.py

```python
if len(peer_msg) > 0:     # peer's stu
    peer_msg = json.loads(peer_msg)
    pass
```

```python
elif msg["action"] == "exchange":
    from_name = self.logged_sock2name[from_sock]
    # Finding the list of people to send to
    # and index message
    pass
    the_guys = self.group.list_me(from_name)[1:]
    for g in the_guys:
        to_sock = self.logged_name2sock[g]
        pass
```

5. chat_server.py

```python
elif msg["action"] == "search":
    pass # get search search_rslt
```

Running chat client and server on different machines

Use python chat_cmdl_client -d [ip address] to connect to remote server (within a subnet)
Example:

```
➡  Solutions to UP3 python chat_cmdl_client.py -d 127.0.0.1
Welcome to ICS chat
Please enter your name:
```

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"