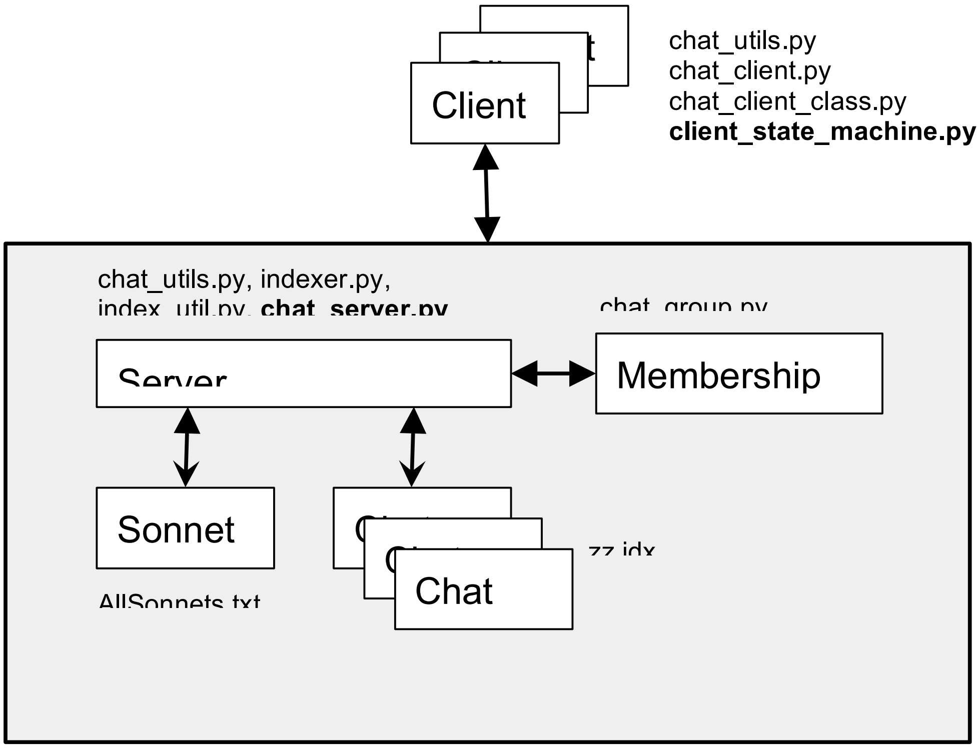
UP3 - NYU-Shanghai ICS Chat System: Spec and Implementation Guide

UP flow:

- Unit Project 1: indexer
- Unit Project 2: group management
- Unit Project 3:
    - 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 are shown above. In UP3, which will be spread over 2 weeks, you will complete **client_state_machine.py** (week 1) and **chat_server.py** (week 2).

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* they were talking directly. What actually happens is the server passing messages back and forth. It also adds other functionalities (such as indexing chat history).

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

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. (week 2)
- **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 follows:
- On one console: “python chat_server.py”. This starts the server.
- On another console: “python chat_cmdl_client.py”. This starts a client

In part 1, you will have a working copy of chat server’s *byte code* chat_server.pyc. Do “python chat_server.pyc” to start the server. In part 2, you will be given an incomplete chat_server.py which you need to implement.

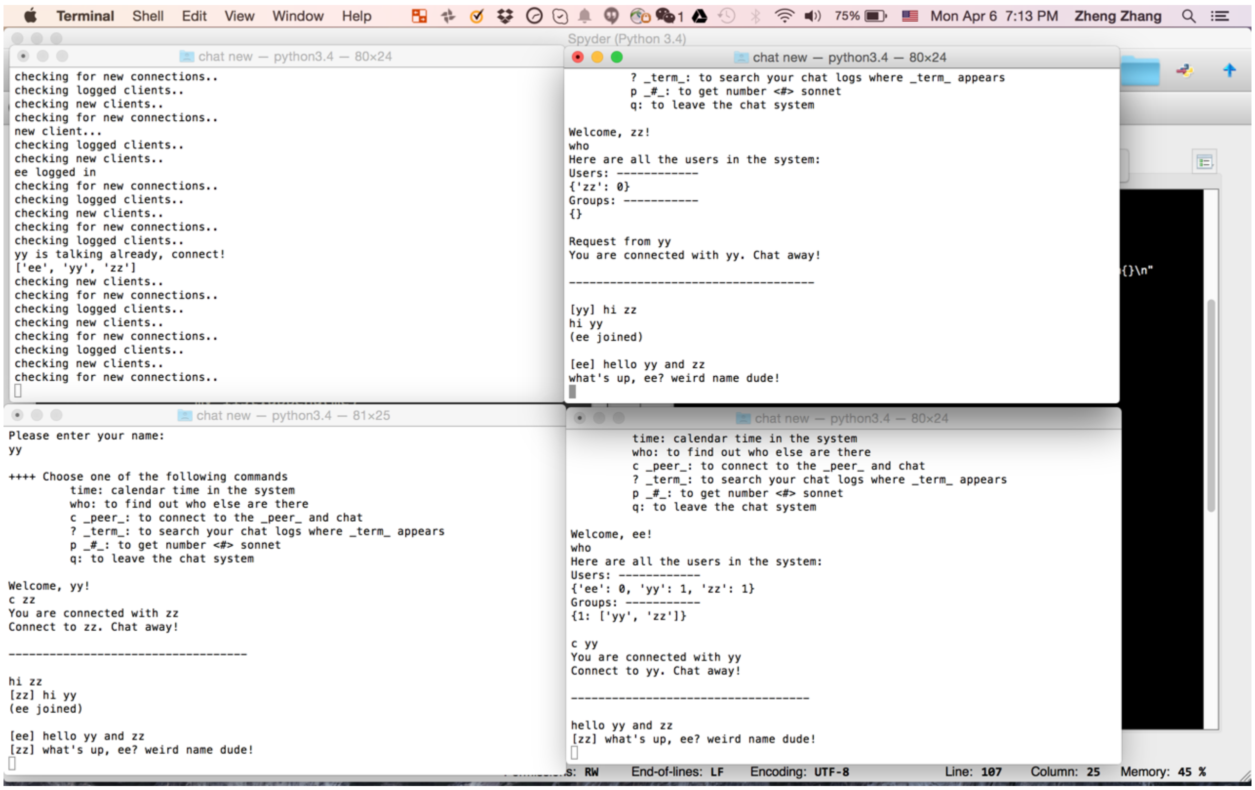
When client starts, it will ask you for a user name, once you enter it, you are logged in. Then the user follows the instructions. Here is one screenshot at the client:

```
chat new — python3.4 — 80×24
NYUSH0838LP-MX:chat new zhengzhang$ python chat_client.py
Welcome to ICS chat
Please enter your name:
zz

++++ Choose one of the following commands
         time: calendar time in the system
         who: to find out who else are there
         c _peer_: to connect to the _peer_ and chat
         ? _term_: to search your chat logs where _term_ appears
         p _#_: to get number <#> sonnet
         q: to leave the chat system

Welcome, zz!
who
Here are all the users in the system:
Users: ------------
{'zz': 0}
Groups: -----------
{}

time
Time is: 06.04.15,16:44
```

Below is a screenshot of 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.



UP3: Part One -- Client Side

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.

The following is a function in chat_state_machine.py:

```
    def proc(self, my_msg, peer_code, 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, M_TIME)
                    time_in = myrecv(self.s)
                    self.out_msg += "Time is: " + time_in
```

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

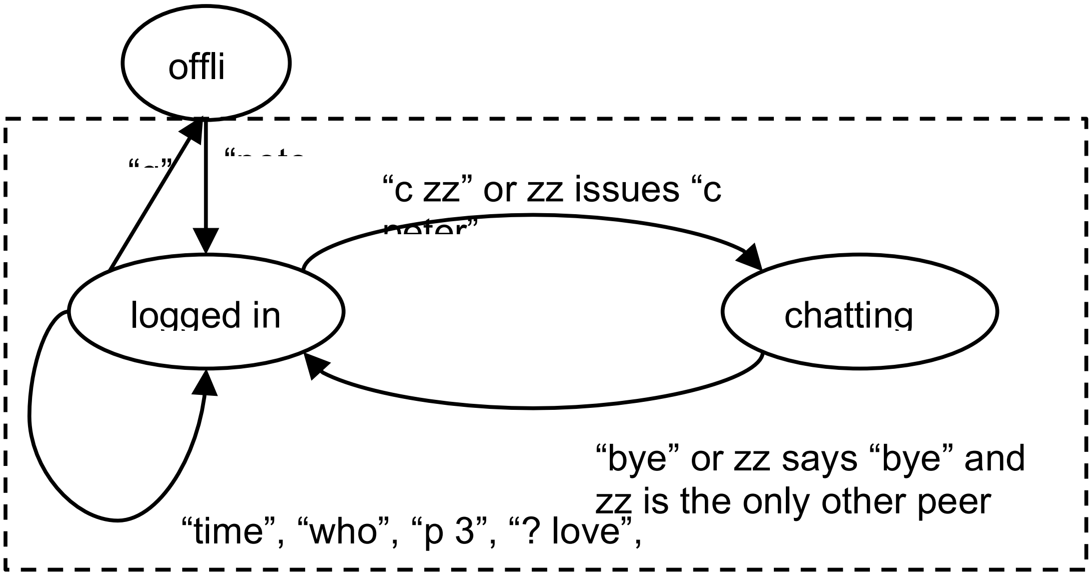
- my_msg: whatever the user typed in the console
- peer_code, peer_msg: the code and the associated incoming message from its peer.

We will explain code and sending/receiving messages shortly. The output of this function is stored in self.out_msg. The rest of the chat code will pick it up and display to the console.

Background: State Machines

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



Protocol code

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

The following table summarizes the code between the client and server. Each code (e.g. M_LOGIN) is a string (see utility). Keep in mind that M_LOGIN, M_CONNECT, etc are string variables so when you see something like M_LOGIN + ‘zz’ (such as in the first row of the table below), the + operator refers to string concatenation.

Message	Request	Response
M_LOGIN + 'zz'	Client to server (when user inputs his nickname)	M_LOGIN+'ok' : login successful M_LOGIN+'duplicate': name already exists
M_CONNECT +'peter'	Client to server (when user type: "c peter")	M_CONNECT+'ok': zz connects to peter successfully M_CONNECT+'busy': peter is busy (unused) M_CONNECT+'hey you': when a user tries to connect to himself Otherwise: peter is not online
M_DISCONNECT (you say ‘bye’)	Client to server (when user type: "bye")	No response needed; zz gets off the chat group
M_DISCONNECT (your last chat partner says ‘bye’)	Server to client (when the other user types "bye" and I am the only one	Sent when peter's only partner, zz, has left the chat group

	left)	
M_TIME	Client to server (when user type: "time")	Respond with string encoding the time
M_LIST	Client to server (when user type: "who")	Respond with the members and the chat groups in the system
M_SEARCH + 'love'	Client to server (when user type: "? love")	Respond with chat history the chats that contains 'love'
M_POEM + '3'	Client to server (when user type: "p 3")	Respond with sonnet #3 (or III)
M_EXCHANGE + 'hi'	Client to server (when user type: "hi")	No response, just pass the text (i.e. 'hi') to every peer in zz's group
M_EXCHANGE + 'bye'	Client to server (when user type: "bye")	No response needed, zz will issue M_DISCONNECT next. If there is only one peer zz is talking, then that peer will get a M_DISCONNECT request from server

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

Code and sending/receiving messages

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

```
4  M_UNDEF      = '0'
5  M_LOGIN      = '1'
6  M_CONNECT    = '2'
7  M_EXCHANGE   = '3'
8  M_LOGOUT     = '4'
9  M_DISCONNECT= '5'
10 M_SEARCH     = '6'
11 M_LIST       = '7'
12 M_POEM       = '8'
13 M_TIME       = '9'
14 
15 SERVER = (socket.gethostname(), 1112)
16 
17 menu = "\n++++ Choose one of the following commands\n \
18         time: calendar time in the system\n \
19         who: to find out who else are there\n \
20         c _peer_: to connect to the _peer_ and chat\n \
21         ? _term_: to search your chat logs where _term_ appears\n \
22         p _#_: to get number <#> sonnet\n \
23         q: to leave the chat system\n\n"
24 
```

line 4-13 etc. are a list of codes that describe an event internal to client and server. For instance when a user first logs in, her chat client will send a message with the code M_LOGIN to server.

Later she asks for the time, and her client sends a M_TIME code (i.e. sending '9' over to server), and so on so forth. See the table defined earlier.

In the future, you can extend the system by defining your own code.

Line 15 gives the server address 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.

```
25 S_OFFLINE   = 0
26 S_CONNECTED = 1
27 S_LOGGEDIN  = 2
28 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.

Messages and sockets:
For programs to talk to each other over the internet, they use *sockets*. This is an advanced topic we will not cover in any detail. For now, think of a socket as the telephone number 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 code has already set up the sockets, so you don't have to implement them. 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.

Implementing the proc logic

Clients move 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 M_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 M_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 M_POEM + 3 to server. Server responds with sonnet III	S_LOGGEDIN

From user: "c peter"	Send M_CONNECT to server. Connect to user peter	S_CHATTING
From user: "? raining"	Send M_SEARCH to server. Search all past chats containing keyword "raining"	S_LOGGEDIN
From peer: M_CONNECT	Accept the peering request The name of your peer will be in the argument *peer_msg* 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 M_EXCHANGE + "[my_name] this is a good day"	S_CHATTING
From user: "bye"	Send to sever M_DISCONNECT + "[my_name] bye"	S_LOGGEDIN
From peer: peer code is "M_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 -- Implementing the Server Logic

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)

Server message handling

The server, to be implemented in chat_server.py, will receive all sorts of messages from clients (messages such as M_CONNECT, M_EXCHANGE, etc). The main job of the server is to respond to all those messages. To handle them, the Server class maintains 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

```
class Server:
    def __init__(self):
        self.new_clients = [] #list of new sockets of which the use
        self.logged_name2sock = {} #dictionary mapping username to
        self.logged_sock2name = {} # dict mapping socket to user na
        self.all_sockets = []
        self.group = grp.Group()
        #start server
        self.server=socket.socket(socket.AF_INET, socket.SOCK_STREA
        self.server.bind(SERVER)
        self.server.listen(5)
        self.all_sockets.append(self.server)
        #initialize past chat indices
        self.indices={}
        # sonnet
        self.sonnet = indexer.PIndex("AllSonnets.txt")

```

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:

```
#==========================================================================
# 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(). Here are the first few lines of handle_msg(),

```
def handle_msg(self, from_sock):
    #read msg code
    msg = myrecv(from_sock)
    if len(msg) > 0:
        code = 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)*, giving us a string whose first character is one of the “M_CODEs” that were mentioned above (which is why we have the last line).

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

- M_CONNECT
- M_EXCHANGE
- M_LIST
- M_POEM
- M_SEARCH
- M_DISCONNECT

Here are when and how the two dictionaries are used:

- *self.logged_name2sock*: maps a client's name to its socket. When you want to broadcast a message from a peer to the rest of the members in her group, you will find other members by asking self.group, and then get their sockets from this dictionary, then send messages to them.
- *self.logged_sock2name*: the reverse of the above; map a socket to the client name. When a message from a socket arrives, you use this dictionary to find out which peer is using this socket.

The cases of M_LIST and M_SEARCH are the easiest to start. In both those cases, you only have to send back the appropriate message in a string. As a reminder, the first character of the message should always be the M_CODE). See what happens if you merely send back the string "17" (end the blocks with my_send(from_sock, M_CODE + 17) and run the server and clients).

For M_POEM, you are given an incorrect implementation. Your job is to correct it. Hint: Recall that when the code is M_POEM, the messaged you received is of the form M_POEM + sonnet_number.

M_CONNECT, M_EXCHANGE, and M_DISCONNECT will be a bit more challenging. In M_CONNECT and M_DISCONNECT, you need to call the appropriate methods of the group management.

For M_CONNECT, the message you receive will be of the form M_CONNECT + username, where username is who the client wishes to connect to. You will have to tell that user (and possibly everyone in user's group) that the client is connecting to her. Also, besides handling the legitimate case of a client trying to connect to another client, don't forget to handle the cases of a client attempting to connect to herself or a nonexistent client.

For M_EXCHANGE, the message you receive is of the form M_EXCHANGE + string 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.

For M_DISCONNECT, if somebody in a 2 person group disconnects, the server will have to tell the other person to disconnect as well.

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”