

NYU-Shanghai ICS Chat System: Spec and Implementation Guide

[Overall architecture](#)

[State machine](#)

[Protocol code](#)

[Module: indexer](#)

[Module: chat utilities](#)

[Module: membership management](#)

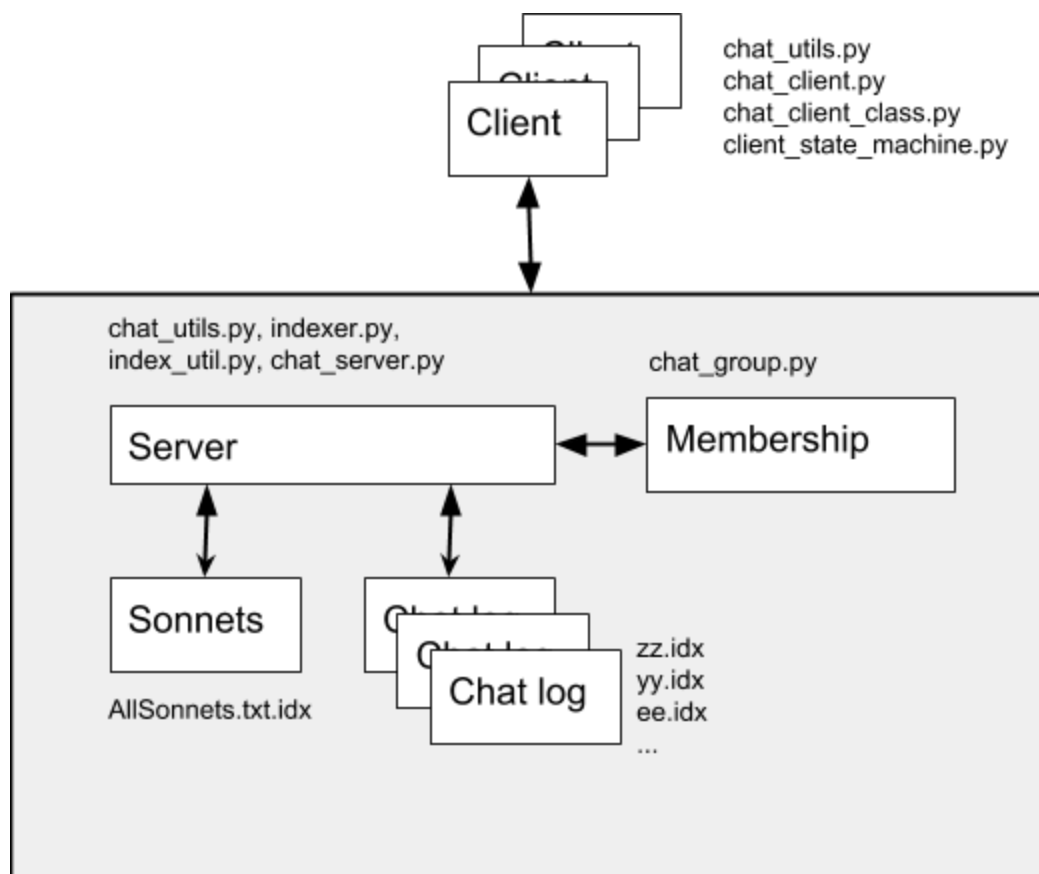
[Module: client side state machine](#)

[Module: server side](#)

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:



```
chat new — python3.4 — 80x24
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 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.

```

Terminal Shell Edit View Window Help
Spyder (Python 3.4)
chat new -- python3.4 -- 80x24
checking for new connections..
checking logged clients..
checking new clients..
checking for new connections..
new client...
checking logged clients..
checking new clients..
ee logged in
checking for new connections..
checking logged clients..
checking new clients..
checking for new connections..
checking logged clients..
yy is talking already, connect!
['ee', 'yy', 'zz']
checking new clients..
checking for new connections..
checking logged clients..
checking new clients..
checking for new connections..
checking logged clients..
checking new clients..
checking for new connections..
[

chat new -- python3.4 -- 80x24
? _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: -----
{}

Request from yy
You are connected with yy. Chat away!

-----

[yy] hi zz
hi yy
(ee joined)

[ee] hello yy and zz
what's up, ee? weird name dude!

chat new -- python3.4 -- 80x24
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, ee!
who
Here are all the users in the system:
Users: -----
{'ee': 0, 'yy': 1, 'zz': 1}
Groups: -----
{1: ['yy', 'zz']}

c yy
You are connected with yy
Connect to yy. Chat away!

-----

hi zz
[zz] hi yy
(ee joined)

[ee] hello yy and zz
[zz] what's up, ee? weird name dude!

S: RW End-of-lines: LF Encoding: UTF-8 Line: 107 Column: 25 Memory: 45 %

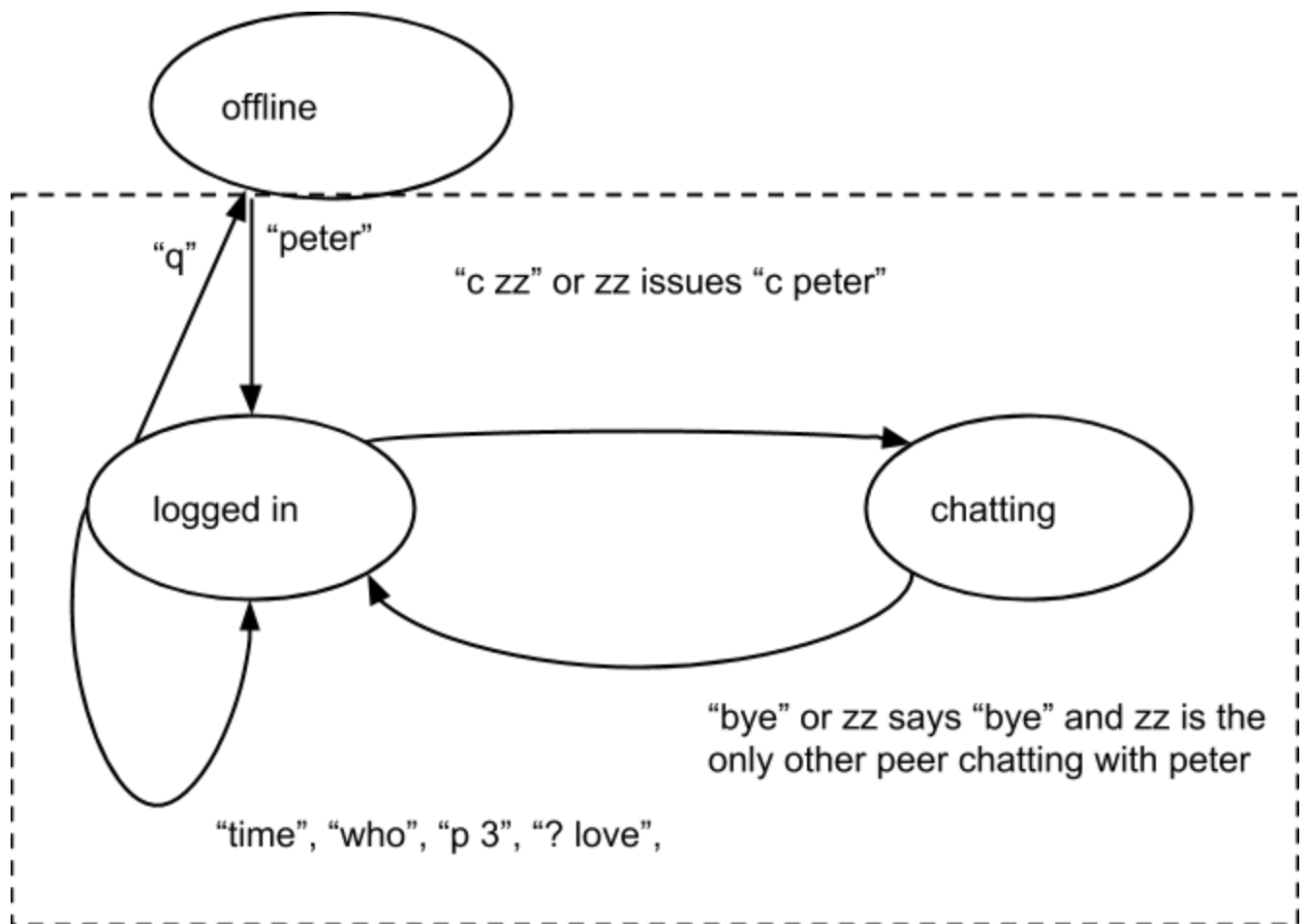
```

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
<code>{"action":"login", "name":"zz"}</code>	Client to server (when user inputs his nickname)	<code>{"action":"login", "status":"ok"} : login successful</code> <code>{"action":"login", "status":"duplicate"} : name already exists</code>
<code>{"action":"connect", "target":"zz"}</code>	Client to server (when user type: "c peter")	<code>{"action":"connect", "status":"success"} : zz connects to peter successfully</code> <code>{"action":"connect", "status":"busy"} : peter is busy (unused)</code> <code>{"action":"connect", "status":"self"} : when a user tries to connect to himself</code> <code>{"action":"connect", "status":"no-user"} : peter is not online</code>
<code>{"action":"time"}</code>	Client to server (when user type: "time")	Respond with string encoding the time
<code>{"action":"list"}</code>	Client to server (when user type: "who")	Respond with the members and the chat groups in the system
<code>{"action":"search", "target":"'love'"}</code>	Client to server (when user type: "? love")	Respond with chat history the chats that contains 'love'
<code>{"action":"poem", "target":"3"}</code>	Client to server (when user type: "p 3")	Respond with sonnet #3 (or III)
<code>{"action":"exchange", "message":"'hi'"}</code>	Client to server (when user type: "hi")	No response, just pass the text (i.e. 'hi') to every peer in zz's group
<code>{"action":"disconnect"}</code>	Client to server (when user type: "bye")	No response needed; zz gets off the chat group
<code>{"action":"disconnect"}</code>	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.

```
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

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 have 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`.

```
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 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 <i>self.out_msg</i> 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": 'hi' }	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

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

```
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

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

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": "connect", "action": "exchange", and "action": "disconnect" will be a bit more challenging. In "action": "connect" and "action": "disconnect", you need to call the appropriate methods of the group management.

For "action": "connect", the message you receive will be of the form {"action": "connect", "target": "<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 `"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.

For `"action": "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"