# YaChat - A Chat System (Client and Server)

EE 382N, Spring 2019

Dr. Ramesh Yerraballi

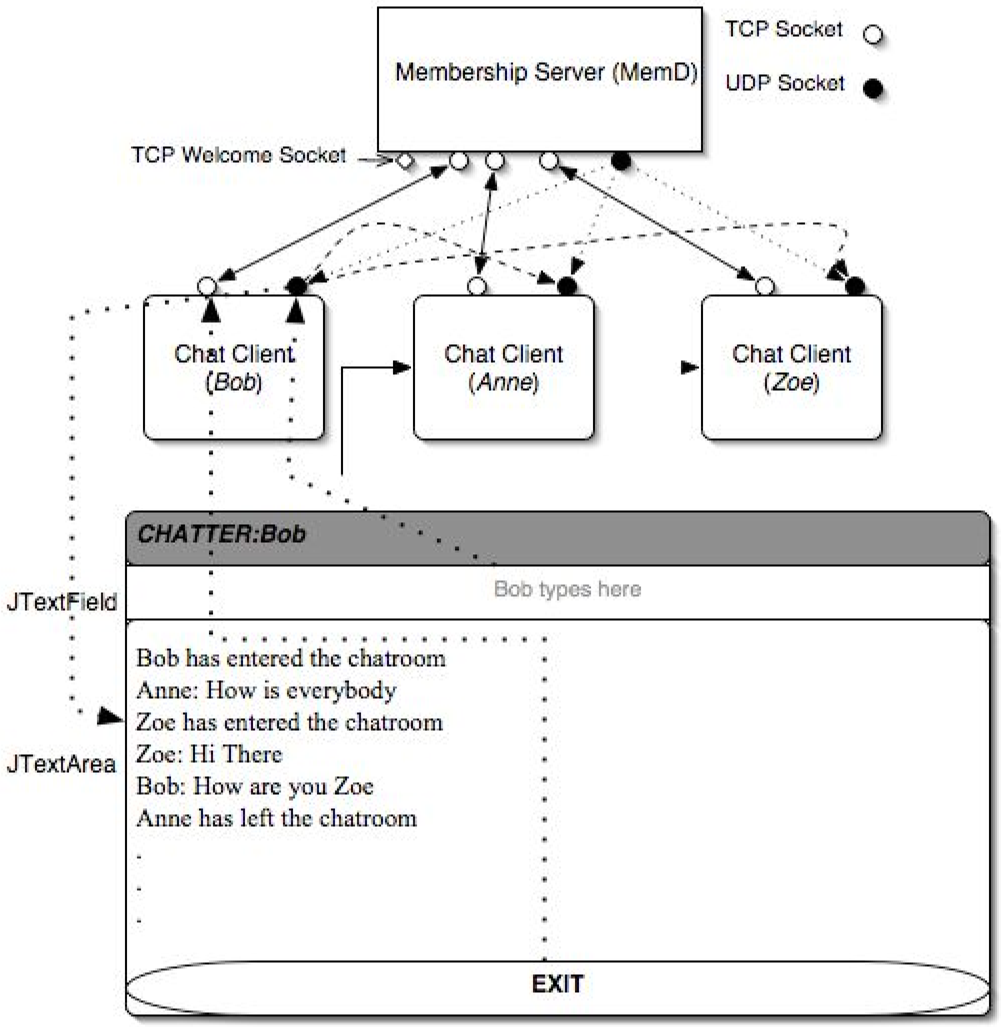
The University of Texas at Austin

Due: First Iteration: Sunday 2/16 Midnight

Second Iteration: Sunday 3/9 Midnight

## Objective

Implement the Client and Server part of a Chatroom facility following the pre-defined protocol. The implementation follows a classic client-server paradigm. The single server maintains and conveys membership information for the various clients. Server is only involved for entry and exit of Clients. Clients communicate messages with each other directly (using UDP) without the Server's involvement.



## Details

You have to implement a chat client (Chatter) that can be used by users anywhere in the Internet to talk to each other using a Membership server (MemD - Membership Daemon). The outline of the client (which can be either GUI-based or CLI-based) is as follows:

1. Create a TCP socket to connect to the server at its Welcome Socket. The server must be running at a known port, which is given as a command-line argument to the client when it is run (see below). Also create a UDP Socket which will be used to communicate messages.
2. Send its `screen_name IP_Address` and `UDP_Port` to the server once connected. Let the Server validate the `screen_name`. The `UDP_Port` is from the UDP Socket created in the previous step.
3. If validated, receive a list of members who are already in the chatroom (including yourself). Specifically, the `screen_name IP_Address` and `UDP_Port` of each member are sent.
4. Continuously read UDP socket for messages from other clients (chatters) and display messages.
5. Continuously read input typed in the client. Display it on the client side and send it to all the clients over UDP (to their corresponding UDP ports) formatted as shown below in the Protocol.
6. If the chat user quits the chat then send a message over TCP to the membership server. The membership server will confirm your quit on your UDP socket. Receive confirmation and then quit.
7. When one of the other chatters quits, you will receive a "quit" message over UDP informing you this. In response to this remove, the specific member from your list of active members.

Given that the client is doing two things simultaneously, reading the console and UDP socket, you may either want to make it multi-threaded or use some other approach to achieve this. The outline of the Membership server (MemD - Membership Daemon) is as follows:

1. The server has to be Multi-threaded (like the `ConTCPServer` example discussed in class).
2. The Server waits for connections from clients (Chatters) on the TCP Welcome Socket (port is taken as a command-line arg). Each Chatter is accepted on a separate TCP socket (connectionSocket) and a thread is created to serve the Chatter (we will call this the servant thread).
3. The first message received by the servant thread from the Chatter (on the connectionSocket) is the Chatter's identity (`screen_name`, `IP_Address` and `UDP_Port`). The Servant thread validates the `screen_name` by looking it up in the Membership list (a Data Structure maintained at the Server).
    a. If there is no other member in the list with the same requested `screen_name` then, the new Chatter is added to the List and an Acceptance message is sent back to the Chatter. The acceptance message contains a list of the identities of all the Chatters currently in the chatroom.
    b. If there is a member with the requested `screen_name` already in the list then, it sends a Rejection message to the client and quits (note the main thread is still running).

The Servant thread now blocks on reading the connectionSocket to be notified when the Chatter exits the chatroom.

1. When a new client is added to the membership list (3-a above), the server sends a message to all the Chatters (in the Membership list) at their UDP ports conveying to them the identity of the new Chatter. Note that the message is sent to ALL members in the list including the Chatter who just joined.
2. When a Chatter exits the chatroom the Chatter notifies the corresponding Servant thread over its connectionSocket. The servant thread then, removes the Chatter from the Membership list and sends a “Exit” notification to ALL members (including the Chatter who is exiting) in the Membership list at their UDP Ports. It then quits (again, the main thread is still running)

## Implementation

You are expected to implement this project in two main iterations. In the first iteration you will write the **client** code and use my **server** solution to test it. In the second iteration, you will write the **server** code and use my **client** solution to test it. In the second iteration, you can also use your own client code developed in the first iteration in conjunction with my client solution to test the robustness of your **server** solution.

**First Iteration:**

The Membership Server (`server.YMemD`) provided to you is part of the `ChatSystem.jar` archive. This has to be run first. For testing purposes, you can run the server and the multiple clients on your local machine. Once you have it working, you can test your client by running the server on a remote machine and running your clients on your local machine.

*Run Server as:*

```
      java -cp ChatSystem.jar server.YMemD <MemD_welcome_tcp_port>
```

*Example (on localhost):*

```
      $ java -cp ChatSystem.jar server.YMemD 7676
```

You client solution (*assuming you are developing in Java*) will be part of an archive called `YaChat.jar`.

You will run your Client as:

```
      java -cp YaChat.jar client.Chatter <screen_name>
<MemD_server_hostname> <MemD_welcome_tcp_port>
```

That is, your client will take 3 inputs - the screen name, and the two coordinates of the MemD server.

*Examples*

(on localhost - open one cmd window per chatter and run in each window):

```
      $ java -cp YaChat.jar client.Chatter Anne localhost 7676
      $ java -cp YaChat.jar client.Chatter Bob localhost 7676
      ...
      $ java -cp YaChat.jar client.Chatter Zoe localhost 7676
```

## Protocol - Client Perspective

Here is the interesting part of the project. Your code for the client has to follow a protocol. What this implies is that your client, if implemented correctly (protocol compliant) should be able to talk to my Membership Server that follows the same protocol. What is more, it should also work with somebody else's protocol compliant client implementation. Here are the conventions used in the protocol description.

- Keywords (Commands) are in uppercase
- A blank space is indicated by "¤"
- The newline character is '\n'.
- Angled braces <> are used to enclose information which has special meaning. Please note that the actual messages do not have these angled braces.

| Message | Meaning |
|---|---|
| [*Send*]<br>`HELO¤<screen_name>¤<IP>¤<Port> \n` | This message is sent as the first greeting from the client to the server immediately after connecting. The Port is the UDP Port at which you intend to receive and send chat messages. The `screen_name` must not have spaces in it. [TCP] |
| [*Send*]<br>`MESG¤<screen_name>:¤<message>\n` | This message is sent to the UDP ports of all members in the membership list, when a chat user types in a `message` [UDP] |
| [*Send*]<br>`EXIT\n` | When the Chat Client wants to terminate (or exit) the chat it sends this to the Membership Server over TCP.<br>The exit should take effect ONLY when the client receives a response (see below) back from the server over the UDP Socket (acknowledging the exit) and then terminate. [TCP] |
| [*Recv*]<br>`ACPT¤<SN1>¤<IP1>¤<PORT1>:`<br>`<SN2>¤<IP2>¤<PORT2>:`<br>`...`<br>`...`<br>`<SNn>¤<IPn>¤<PORTn>\n` | The server sends this message in response to the Greeting, to acknowledge the validity of the `screen_name` and to inform the Chatter Client of the identities of ALL the Chatters (***including yourself***).<br>Each identity is separated by a ":". [TCP] |
| [*Recv*]<br>`RJCT¤<screen_name>\n` | The server sends this message in response to the Greeting, to let the Chat Client know that the `screen_name` is already in use. [TCP] |
| [*Recv*]<br>`MESG¤<screen_name>:¤<message>\n` | This is a message received on the UDP Socket. It is another Chatter's chat message. Parse it and display the message (minus the `MESG` string) [UDP] |
| [*Recv*]<br>`JOIN¤<screen_name>¤<IP>¤<Port>\n` | This is a message received on the UDP Socket. It is sent by the Server to indicate that a Chatter (it |

| | |
|---|---|
| | could be yourself) has joined the chatroom. If the identity is yours then use this as a confirmation that you have been registered in the Server's Membership List. If it is a "new" Chatter, add this Chatter identity to the local Membership list. [UDP] |
| [Recv] `EXIT¤<screen_name>\n` | This is a message received on the UDP Socket. It is from the Membership Server notifying the exit of a member from the chatroom. Parse it and display an appropriate message (Elvis has left the Building!!); Remove from local list. [UDP] |

**Second Iteration:**
You have to implement a Membership Server (`server.MemD.jar`, if implementing in java) that can be contacted by users anywhere in the Internet to talk to each other using a Chatter Client provided to you (`client.YChatter` which is part of the `ChatSystem.jar` archive). Your Membership Server has to be run first. For testing purposes, you can run the server and the multiple clients on your local machine.

*Run Server as*:
```
      java -cp YaChat.jar server.MemD <MemD_welcome_tcp_port>
```

*Example (on localhost)*:
```
      $ java -cp YaChat.jar server.MemD 7676
```

*Run the provided Client as*:
```
      java -cp ChatSystem.jar client.YChatter <screen_name>
<MemD_server_hostname> <MemD_welcome_tcp_port>
```

*Examples*
(on localhost - open one cmd window per chatter and run in each window):
```
      % java -cp ChatSystem.jar client.YChatter Anne localhost 7676
      % java -cp ChatSystem.jar client.YChatter Bob localhost 7676
      ...
      % java -cp ChatSystem.jar client.YChatter Zoe localhost 7676
```

## Protocol - Server Perspective

Here is the Server side of the Protocol:

| Message | Meaning |
|---|---|
| [*Send*] `ACPT¤<SN1>¤<IP1>¤<PORT1>:<SN2>¤<IP2>¤<PORT2>:` | The server sends this message in response to the Greeting, to acknowledge the validity of the screen name and to inform the Chatter Client of |

| | |
|---|---|
| ...<br>...<br>`<SNn>¤<IPn>¤<PORTn>\n` | the Identities of the ALL Chatters (***including yourself***).<br>Each identity is separated by a ":". [TCP] |
| [*Send*]<br>`RJCT¤<screen_name>\n` | Meaning same as in Iteration 1 [TCP] |
| [*Send*]<br>`JOIN¤<screen_name>¤<IP>¤<Port>\n` | Notification sent to ALL Chatter clients over their UDP ports to let them know that a new member has entered the chatroom. [UDP] |
| [*Send*]<br>`EXIT¤<screen_name>\n` | Notification sent to ALL Chatter clients over their UDP ports to let them know that a member has left. [UDP] |
| [*Recv*]<br>`HELO¤<screen_name>¤<IP>¤<Port>\n` | A Chatter client sends this to enter the chatroom; [TCP] |
| [*Recv*]<br>`EXIT\n` | A Chatter client sends this to indicate exit from the chatroom; [TCP] |

## Submission

Submit your `YaChat.jar` file including source (.java) and class (.class) files:

- `client.Chatter` (Client solution from the first iteration)
- `server.MemD` (Server solution from the second iteration)

If you are doing the project in python or C then you will submit a zip archive with clear running instructions for the TA.