23/2/2016

Rounak Das

Jadavpur University, BCSE-iv (a2), Roll-001210501036

Assignment 3

Chat Application

Problem Statement:

Write multi-client chat application consisting of both client and server programs. In this chat application simultaneously several client can communicate with each other. For this you need a single server program that clients connect to. The client programs send the chat text (input) to the server and then the server distributes that chat text to all the other clients. Each client then displays the text sent to it by the server. Server should be able to handle several clients concurrently. It should work fine as clients come and go. This can be implemented in two ways

1. Server can handle multiple clients simultaneously by forking a separate process for each client.
2. Server can create separate threads (instead of processes) to handle separate clients.

In general, the server program:

* Accepts connection requests from clients
* For each accepted connection start a process/thread
* Each process/thread reads data from the client and sends it to all other clients or selected clients
* When a process/thread detects that a client has disconnected it should free allotted resources and stop processing for that client.

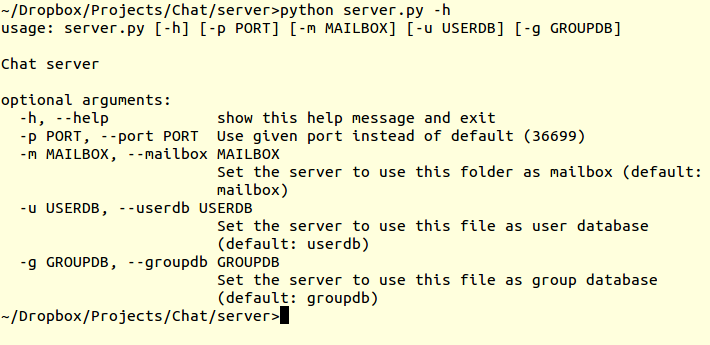
Design of the chat system:

The chat system is a centralized client-server application. The clients send their messaged to the server, which then relays the message to the client.

## Server:

At the server the system mimics POP is some ways. The relay mechanism is based on files. Each unique user has their own file (hereafter referred to as their mailbox). The threads/processes deposit messages intended for a user in their mailbox. Access to this mailboxes are serialized using mutex locks. This is to protect mailboxes from concurrent access by threads/processes. When a user wants to read their messages their mailbox is read and the messages are sent to them. Thereafter the mailbox is emptied. The mailbox acts as a queue that stores one message per line.

Full server options:

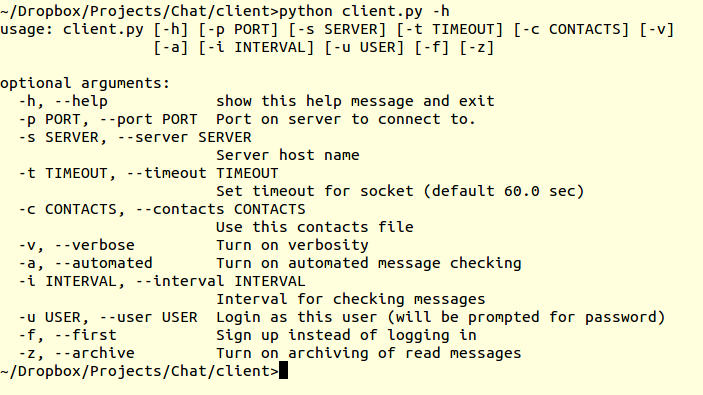


* *userdb* is the file containing the list of users
* *groupdb* is the file containing all multicast group definitions
* *mailbox*  is the folder where all the users’ mailboxes are stored
* *port*  can be reconfigured(default: 36699)

## Client:

At the user side the client maintains a constant TCP connection with the server. It reads commands from the CLI and builds a message to the server by parsing the command, then sends it to the server. The server response is then showed to the user. Since messages that have been retrieved are deleted at the server, the client has an option to archive the messages that it reads. The client can be configured to poll the server at regular intervals to check if the user has any messages to read. By default the server IP and port are read from a configuration file “chat.cfg” but this can be overridden via command lien options. The client may also be run to sign up for this service.

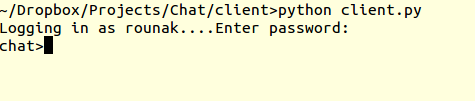
Full client options:

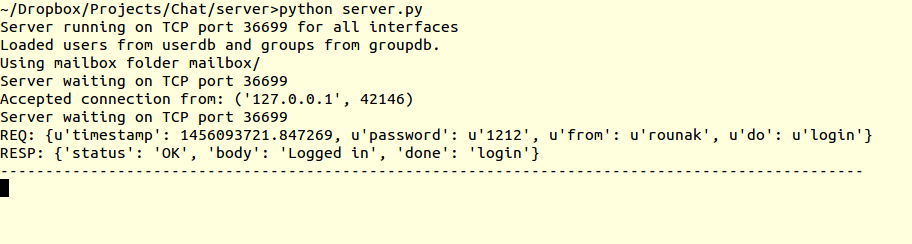


* server IP and port may be overridden here
* Timeout may be changed
* Contacts specifies a file from where local bindings of usernames to keywords can be saved; this is useful to remember who-is-who
* Verbose mode is for debugging
* If the client is set up to check for messages automatically, its polling interval can be changed
* If user is not specified in either “chat.cfg” or at the command line, the client prompts the user to specify a username
* If a new user wishes to login, they can specify the –f flag to sign up first
* Users may also all messages read during a session

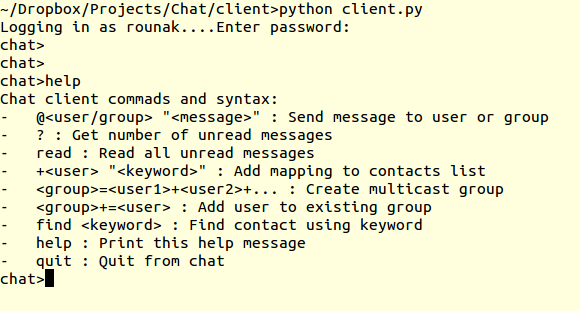
All interaction between the server and the client happens via exchange of JSON strings encoded as base64 strings. Python dictionaries are used to build and store requests and responses. These are serialized to JSON strings before being encoded as base64 strings for sending.

The login interaction is shown below:





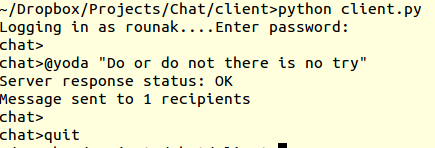
The client provides an intuitive CLI for sending and receiving messages.



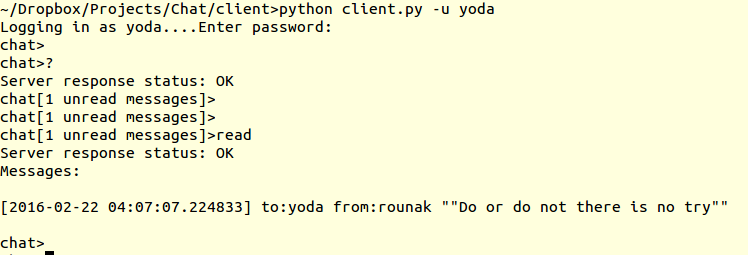
* *@user message*: sends a message to a user or group
* *?*: Gets the number of messages for the user without reading them
* *read*: reads the messages from the server
* *+user Keyword*: Add a local mapping of keyword to user
* *group=user1+user2+user3*: Creates a multicast group
* *group+=user*: Add a user to an existing group(only if you’re already a member)
* *find Keyword*: Find local mapping for keyword
* *quit*: Exit

Example:

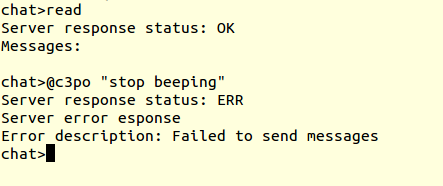
1. Send:



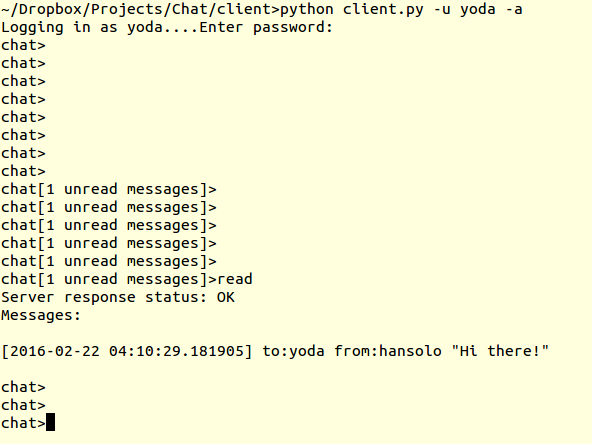
1. Receive:



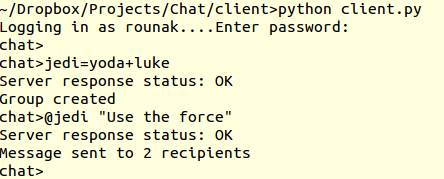
1. OK and ERROR responses from server:

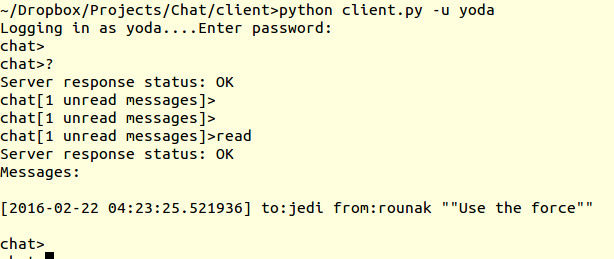


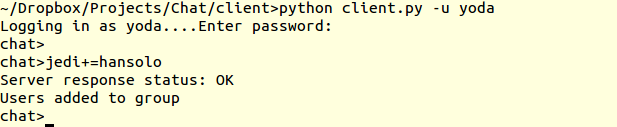
1. Automated message checking:

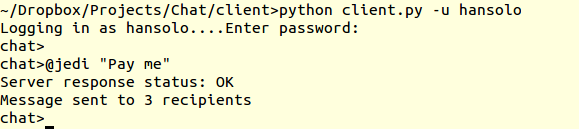


1. Group dynamics:

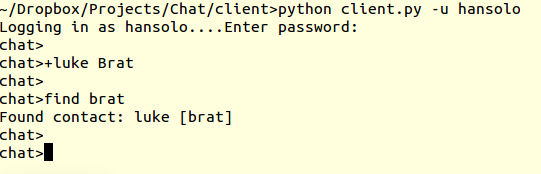








1. Contacts:



Performance:

## Settings:

### Threads:

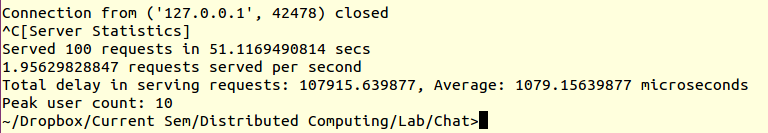
* Using pythons built-in threading module
* Using threading.Lock for mutex
* Each file has an associated lock
* Also locks for shared data structures

### Processes:

* Using pythons multiprocessing module
* Using multiprocessing’s Manager based Lock object for synchronization
* The mailbox folder has one global lock
* Locks for shared variable and file handles

## Testing methodology:

* Spawn separate processes one for each client
* Each client reads a set of commands from a file
* They send those commands to the server in quick succession (much quicker than humanly possible)
* At the server we measure:
  + Peak user count (number of concurrent threads/processes running at any time)
  + Total response time
  + Number of requests served
  + Uptime
  + E.g.



## Results:

The number of concurrent users was steadily decreased from 10 to 1 and we charted the average response time for each case thread vs process.

## Conclusions:

The result shows the process takes about twice as long on average to respond than processes. This is due to a number of reasons:

* Inherent slowness of pythons Manager server process against threads
* Accessing shared memory objects take longer than data in your own address space
* Locks in shared memory are slower
* Processes are much higher overhead for something as simple as writing to files(which is what the chat server mostly does)
* Threads are faster for light computing jobs (such as this)
* Processes would be faster if our workload was more computing than IO. Separate processes would be scheduled so that they get more time on the CPU and increase throughput. For threads that must share CPU time among threads of the same process they will get lesser share of the CPU and will be slower.

Code:

Code is available at: <https://github.com/DroidX86/CLIChat>