Mail Server Documentation

Introduction

Simple mail server protocol is a messaging server that uses socket-based communications to communicate data between a client and server in a manner that obeys a well-defined set of rules, or *protocol*. It allows clients to send a message to the server, and then indicate who that message is for. The server will store the message until that user comes along and requests the message or messages that have accumulated.
Imagine yourself as a client to this messaging server. First, you need to register to the server with a username and password. Once you are registered, you can use different methods to login or logout, to store messages in the server, to send messages to others in the server, to check your mailbox, to delete any message you want and to count messages that are waiting for you in your mailbox.

How does it work?

There are two parts in mail server protocol. First, there is a server-side python script that is an actual mail server. Second, on a client-side, there should be a client python script that is used to connect to the server. There are several steps you have to follow in order to connect the client to the server.

a) Run the server code on a specific host and port. E.g. python server.py <host> <port>
Note: If you are not running client on localhost, then you should let your server's host be 0.0.0.0 (accepts any IP address that is trying connect to the server).
b) Run client code and specify the host and port number to which the server is listening to. E.g. python client.py host port
Note: If you are not running client from the localhost, you should specify the private ip address of the server as your host.

Once the client connects to the server, the client makes every single request to the server in the form of a string. E.g. "REGISTER <username> <password>"
The first time client connects to the server, it should register with a username and password. Then, the server creates an account for the client and assigns to the client a randomly-generated cookie to keep track of client's state. After receiving a cookie, client can now call different methods(checking mailbox, sending message, deleting message, etc) by only providing a cookie to the server.

If the client calls the “logout”method, the server deletes the client’s cookie and closes client’s session. The client can start a new session by calling a “login” method. Thus, the server keeps track of every client’s state by assigning a session cookie when the client registers or logs in. The server destroys the client’s cookie when the client logs out.

Server Structures

The server will have the following data structures:

- **Incoming Message Queue (IMQ)**. This is a list. Whenever a message comes in, the server should *append* it to this list.
- **User Mailboxes (MBX)**. The mailboxes are implemented as a dictionary. The keys for that dictionary are usernames. The values of each location in the dictionary are queues of messages.
- **Registered Accounts (RA)**. Implemented as a dictionary. The keys for that dictionary are usernames. The values of the dictionary are the passwords of the usernames.
- **Session Cookies (ID).** Implemented as a dictionary. The keys for dictionary are usernames. The values are session cookie codes for each usernames.
- **Assigned Cookies (AC).** Implemented as a list. After assigning a session cookie to a username, append session cookie to this list just to keep track of what cookies were assigned to prevent duplication.

Conventions

1. When a protocol element is surrounded by <angle brackets>, it is assumed that the brackets themselves are not part of the protocol. For example, "REGISTER <username>" suggests protocol strings of the form "REGISTER jadudm", where a single username, with no spaces or characters outside the range a-zA-Z0-9 will be used.

The API

Every single request the client makes to the server will be in the form of a string. So, the server is going to sit on a socket, listening for connections, and then have to *parse* the strings it receives. Based on the contents of those strings, it will do the correct action.

- `"REGISTER <username> <password>"`
  When the server receives a string that begins with the word REGISTER, it should interpret the second word in a string as a username and third word as a password. First, the server should store the username and password in RA(registered accounts) dictionary (username =key, password=value). Second, the server should add a key to the MBX for that username, and an empty list of messages should be initialized at that location in the MBX. Third, the server should generate a random session cookie code for this username. After it generates a cookie, it should check whether the cookie exists in AC(assigned cookies) list in order to prevent duplicates. If there's a duplicate of cookie in AC list, then the server should generate a new cookie. If cookie is not in AC list, it should store the username and newly generated session cookie in ID dictionary, username as a key and session cookie as a value. E.g. {"username":cookie_code}. It also should append the cookie into the AC list. When complete, the server should reply with `"OK <`user session cookie>".

- `“LOGIN <username> <password>”`
  When the server receives a string that begins with the word LOGIN, it should interpret the second word in a string as a username and a third word as a password. Then, it should follow the same process of generating a cookie and assigning it to the username just like it is described in the REGISTER method.

- `“LOGOUT <session cookie>”`
  When the server receives a string that begins with the word LOGOUT, it should interpret the rest of the string as a client's session cookie. Then, it should go through ID(session cookie) dictionary and check whether any of the values match with user-provided cookie.
  If it does, then the server should delete the the session cookie from ID and also remove its key which is username. Also, it should delete the session cookie from AC(assigned cookies) list.
  If not, then the server should reply back with “KO”.

- `"MESSAGE <content> <session cookie>"`
  When the server receives a string that begins with the word `MESSAGE`, it

should assume the first second part of the string is a message and the third part is a session cookie. The server should take a session cookie and check if exists in the ID dictionary. If it does, then the message should be placed on the IMQ, and an `"OK"` response should be sent. If it doesn't the server should reply back with "You should log in first."

- `"STORE <username> <session cookie>"`
  When the server receives a `STORE` comment, it should interpret the first second part of the string is a username and the third part is a session cookie. The server should take a session cookie and check if exists in the ID dictionary. If it does, then take the most recent message off of the IMQ and store it into the given user's mailbox. When done, the server should send an `"OK"` response. If the user does not exist, then the message should be left on the IMQ, and a `"KO"` result should be returned.

- `"COUNT <username> <session cookie>"`
  When the server receives a `COUNT` message, the server should reply with the number of messages waiting in the user's mailbox. It should reply with a string in the form `"COUNTED <n>"`. If the username does not exist, a `"KO"` result should be returned.

- `"DELMSG <username> <session cookie>"`
  When the server receives a `DELMSG` command, it will remove the first message from the given user's MBX queue. After removing the message, the server returns `"OK"`. If the user has no messages, or if the user does not exist, the server returns `"KO"`.

- `"GETMSG <username> <session cookie>"`
  When the server receives a `GETMSG` command, it returns the first message on the user's mailbox queue. If the user does not exist, or if the user has no messages, it returns `"KO"`.

- `"DUMP"`
  When the server receives a `DUMP` command, it should print the contents of the IMQ and MBX to its terminal. This is a debugging command; it allows the client to ask the server to print its contents, which is useful to the server author. It should always return `"OK"`.