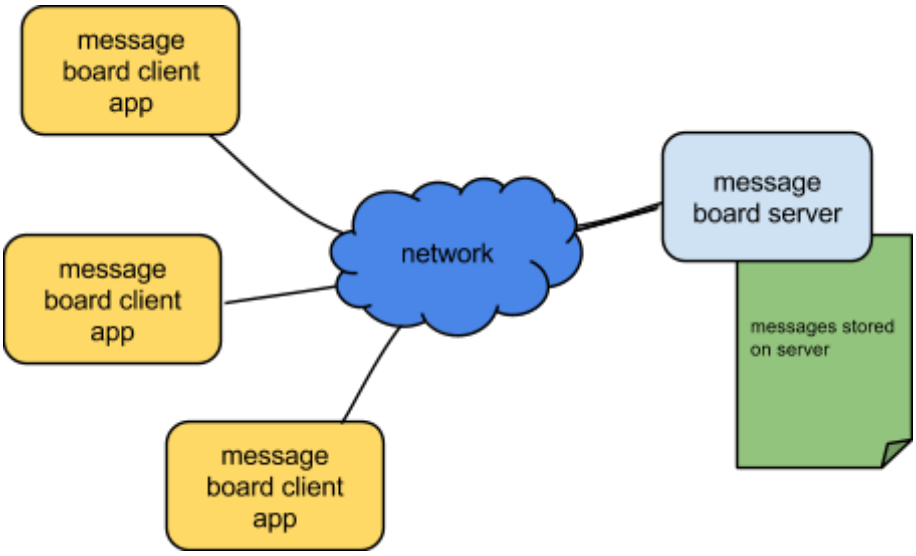
COSC 465: Computer Networking
Spring 2014
Project 1: Message board application with UDP sockets
Due: 30 January 2014, 11:59:59pm
Note: You are welcome and encouraged to work in pairs if you wish

Overview

For the first project, you will write code to complete the networking components of a "message board" client application. The idea of a message board application is similar to a chat room: a central server stores messages posted by any number of clients. Clients can post new messages to the message board, and retrieve the current list of messages stored on the server.

The diagram below depicts the overall structure of the message board application:



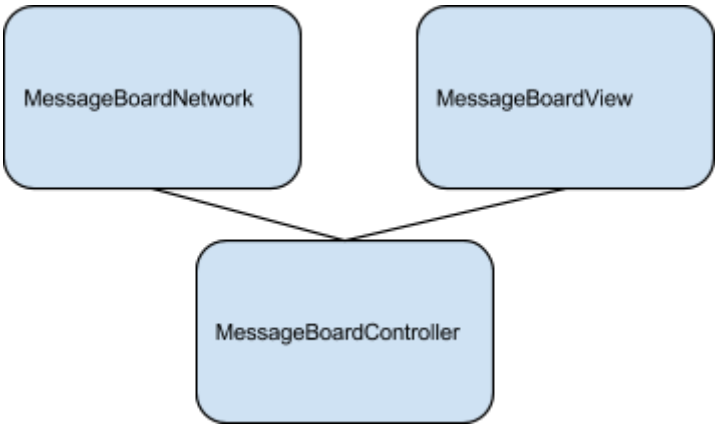
Model-view-controller pattern

To complete the implementation of a message board client, you will follow a software architecture pattern known as "Model-View-Controller". In this (very widely used) object-oriented pattern, an application is divided into three components:

1. A Model class/object, which encapsulates and provides access to data used by the application. Any data used in the application (e.g., messages on the message board) are retrieved through model methods, and any updates to data used in the application are made through model methods. Any "business logic" in the application should generally be located in the model.
2. A View class/object, which encapsulates and provides access to some application display, such as a graphical user interface or web page. Any updates to what the application displays to a user are made through view methods.
3. A Controller class/object, which coordinates access between the model and the view. You can think of it as a go-between or middle-man between the model and the view.

For example, any GUI events (e.g., a mouse click on a button) might result in an action invoked in the controller to retrieve fresh data from the model and call view methods to display the new data.

In this project, there are three Python classes, following the MVC design pattern, as depicted below:



Following the MVC pattern, the view will encapsulate methods for displaying messages in a graphical interface and for dispatching events (e.g., mouse clicks and new messages typed by a user) to the controller, the model class (MessageBoardNetwork) is responsible for sending new messages to the message board server, and for retrieving current messages from the server, and the controller is used to coordinate the entire application.

Your tasks

Your tasks are to complete the MessageBoardNetwork and MessageBoardController classes. The MessageBoardView class is completely written for you.

In the MessageBoardNetwork class:

- Implement the two methods get_messages, and post_message, as well as the constructor (__init__ method).
- The `get_messages` method will need to use Python UDP sockets calls to send a request message to the server to retrieve all messages stored on the server. This method doesn't take any parameters, and should just return a list of strings, where each string is a message that was retrieved from the server.
- Similarly, the `post_message` method will use socket calls to send ("post") a new message to the message board server. This method should accept two parameters: the user name (string) posting the message, and the message text itself.
- The `__init__` method accepts two parameters: the IP address of the message board server, and the UDP port that the server is listening on. You'll need these two items when sending messages to the server (as described below).
- Details on the message formats for requesting messages from the server, or for posting a new message to the server are described below.

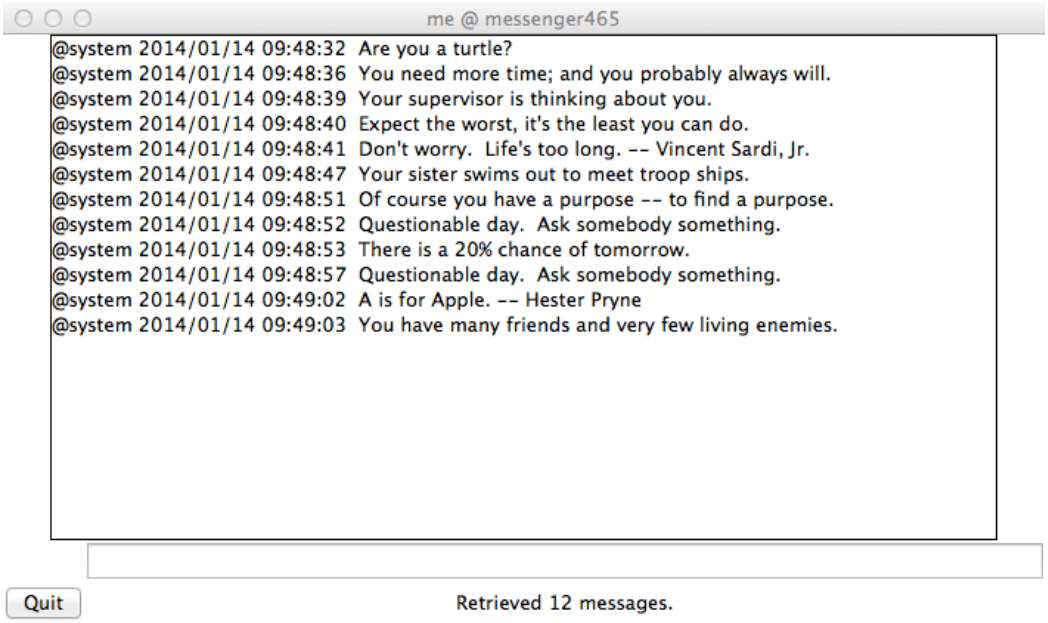
- You will want to think about how to return error indications from the get/post methods in order to inform the controller that something has gone wrong. Exactly how you convey errors to the controller is up to you.

In the MessageBoardController class:

- Complete the `post_message_callback` function. This method is automatically invoked when a user types a new message in the user interface (as described below). This method should call the `post_message` method in the MessageBoardNetwork class to perform sending the message to the server, and then update the view in any appropriate way (e.g., update the status bar to show that the operation was successful or not).
- Complete the `retrieve_messages` function. This method should call the `get_messages` method in the MessageBoardNetwork class to retrieve the current set of messages on the server. This method should then call view methods to display the messages. If there is an error, the view should be updated in some other appropriate way (e.g., update the status bar to show that the operation failed).
- Note that the controller class is already set up to call the `retrieve_messages` method every second (see the first line in the `run` method, that sets up the `retrieve_messages` method to be called after 1 second, then the same line as the first thing in the `retrieve_messages` method). What this means is that any new messages posted to the server should be retrieved and displayed after no more than 1 second or so. This configuration will enable your application to have a relatively up-to-date view of messages posted to the server.

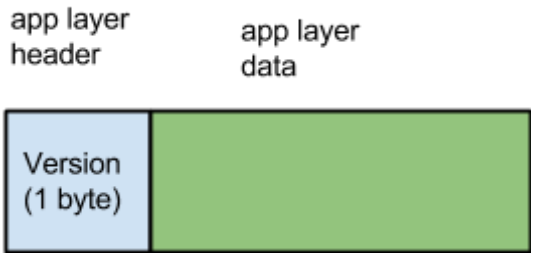
The MessageBoardView class encapsulates a simple graphical interface for the message board client application. It has two useful methods for you to use:

- `set_status` accepts one string as an argument, and displays the string in the "status bar" of the graphical interface of the application. In the example window below, the status bar currently says "Retrieved 12 messages."
- `set_list_items` accepts a list of strings as a parameter, and displays those strings line-by-line in the main portion of the window. In the example window below, there are exactly 12 lines of text displayed in the window, one for each message that was retrieved from the server. Each message (i.e., each line below) retrieved from the server contains a user name, time stamp, and message all as strings.
- You'll also notice a Quit button which quits the application, and a text entry box (just above the status bar). If a user types a message into the text entry box, this string is passed to the `post_message_callback` function in the controller, to be delivered to the server as a new message for the message board.



Network message formats, version A

All network messages to and from the message board server are composed of an **application layer header**, followed by **application layer data**, as depicted below.



For this project, the application layer header consists only of a version identifier as a single byte (one character): the letter A.

The application layer data depends on whether the message is a request for messages from the server, a request to post a new message to the server, or a response generated from one of these requests, as described next.

Requesting messages from the server

To request messages from the server, the application data must consist of the three-character string GET. Note that the entire message sent to the server will thus be the four-character string "AGET" (i.e., the one-character app layer header and the three-character request string).

Posting a new message to the server

To request that a new message be posted to the server, the application data must consist of the following:

- The string POST
- One space
- The username string
- The two-character string "::"
- The message string

The username string can be no more than 8 characters, and the message string can be no more than 60 characters.

As an example, if the user "me" is posting the message "eat more ketchup", the full request string should be "APOST me::eat more ketchup" (notice that the application layer header byte still must appear first).

Note that because the string "::" is used as a delimiter between the user name and message, it MUST NOT appear in the username or message, otherwise the server has no way to determine the correct username or message.

Responses from either GET or POST requests

In response to a POST request, the application data response from the server will either be:

- The two character string OK, if the operation is successful, or
- The five character string ERROR, a space, then an explanation string describing what went wrong. For example, if you send a POST request including a username that is too long, the server will respond with "ERROR invalid user name (either too long or too short".

In response to a GET request, the application data response from the server will either be:

- The two character string OK, followed by a space, followed by the messages retrieved from the server. The set of messages will be in the following format:
    - user1::timestamp1::message1::user2::timestamp2::message2::user3::timestamp3::message3 …
    - That is, for each message, the user, timestamp and message text are separated by the two-character string "::", and each full message (consisting of user, timestamp and text) are separated by "::".
    - For example, the following string is an example of what the application data might look like if there are exactly three messages stored on the server:
        - `OK @system::2014/01/15 14:19:51::Something's rotten in the state of Denmark. -- Shakespeare::@system::2014/01/15 14:20:06::Don't kiss an elephant on the lips today.::@system::2014/01/15 14:20:09::Save energy: be apathetic.`

    - The maximum number of messages that you will ever receive from the server is 20. If more than 20 messages are posted to the server, the server drops the oldest one, keeping only the newest 20 messages (and thus will only respond with 20 messages, at maximum)
    - Note that if there are no messages on the server, the response will be "OK " (OK followed by one space)
- The five character string ERROR, a space, then an explanation string describing what went wrong. For example, if you send a GET request that includes some other text besides GET, the server will respond with "ERROR some other garbage included with GET request"

Python Socket Library (and friends)

You'll need to use the Python socket library for sending and receiving network messages and communicating with the message board server. An overview of the relevant operations in that library is listed below. For the full documentation, see http://docs.python.org/2/library/socket.html.

- `socket.socket()`: this function creates and returns a new socket object. Like it's C counterpart (described in the textbook), it takes three parameters: the socket family name (which should be AF_INET), the socket type (which should be SOCK_DGRAM for this project), and the protocol (which should just be zero, causing the socket function to create and return a UDP socket).

The socket object returned from the above function has three methods you should be aware of (which are analogous to the same functions in the C sockets library, as described in the textbook):

- `sendto(data, addr)`. This method takes two parameters: the data to send (as a string), and an address tuple. The address tuple should consist of the (IP address, port) pair that the data should be sent to. Recall that the `__init__` method in the MessageBoardNetwork class should receive these items as parameters; you'll need to hold on to these parameters (in instance variables) so that you can use them with `sendto`. This method returns the number of bytes sent, or throws an exception on error.
- `recvfrom(buflen)`. This method takes one parameter: the maximum amount of data to receive on the socket (in number of bytes). You can simply use the value 1400 for this project. This method returns a (string, address) tuple where the string is the data received, and the address is a tuple containing the IP address and port of the host from which the data was received.
- `close()`. This method closes the socket.

In addition, you will need to use the `select` function in the Python `select` module. The `select` function allows a programmer to determine whether a socket has data ready to be received on it. By default, a call to `recvfrom` will *block* forever, which would make our application

unresponsive. You're going to use `select` to avoid blocking and to determine when we should give up on waiting for a response from the server.

The `select` function in Python takes three parameters: a list of sockets to be read from, a list of sockets to be written to, a list of sockets to detect errors on, and a timeout value in seconds, i.e., `select(readlist, writelist, errlist, timeout)`.

The pattern that you should use in your interactions with the message board server is as follows:

- Send a request to the server, using the `sendto` socket method
- Call `select`, using only your socket in the readlist (but with empty lists for writelist and errlist), and a value of 0.1 for the timeout (100 millseconds, or 1 tenth of a second).

The return value from `select` is a tuple of three lists corresponding to the first three arguments; each contains the subset of the corresponding file descriptors that are ready. So, for example, if your socket has data waiting on it to be received, the readlist returned will have one element. If the server did not respond in time, or the server is not running, or you sent the request to the wrong address (thus there is no server that can be respond), the readlist will be empty.

Full documentation for select is available here: http://docs.python.org/2/library/select.html#select.select.

For each of the GET and POST requests, you should only try the operation *once*. If it fails either due to the server not responding, or an ERROR response by the server, you should *not* retry the operation. Instead, just display some kind of message (in the status bar of the view) indicating the failure.

Testing

A compiled version of the server is posted on Moodle for testing purposes, along with the starter template for this project. To start the server, open a terminal and type:

```
$ python MBserver465.pyc
```

You should see some output from the server as it runs. It will randomly add messages to the message board, just to make things semi-interesting.

To run your client program and test with the server, you can open a terminal and type:

```
$ python messenger465_client_template.py
```

It will ask for a user name (you can type anything you like) and open the view. Initially, it will not do anything: you'll need to fix that!

Once you've tested the code on your own computer, you can try testing it with a remote server. A server is running on a host with IP address 149.43.80.25. To start your client program and communicate with this server, you can open terminal and type:

```
$ python messenger465_client_template.py --host 149.43.80.25
```

(Note that you may not be able to access the server from off campus due to firewall issues.)