1 Exercise 2: A More Robust Messaging System for Healthcare Professionals

1.1 General Instructions

Submit your solution for this exercise via Blackboard. Before submitting it, test all your lab work on one of the lab machines, making sure that your code solution runs in the lab environment, assuming that your code will be tested in one of those machines. Although the machines in the labs of the Kilburn Building are all dual-boot (with Windows and Linux), we will be using Linux.

1.2 Getting Ready to Start

In the first exercise, we built a very simple messaging system to be used by healthcare professionals using a page of PHP, hosted by a web server, to act as a central 'state store'. Building this far-from-optimal system should have highlighted several issues. In this exercise, we will have the opportunity to revisit those issues and develop a more robust solution, by building a messaging setup using a proper message passing client and server.

It should be able to:

- Allow any number of clients to connect, and identify themselves with a screen name.
- Allow any user to send messages to all other users.
- Allow any user to send a private message to a particular user.

Get started by downloading the files you need from the Blackboard page of this course unit, and putting them in a suitably named sub-directory of your `~/COMP28112` directory.

You should have the following files:

- `ex2utils.py`: this is a Python module containing a couple of classes that you will need to extend.
- `client.py` : a very simple skeleton code. You will use this code to implement your client.
- `server.py` : some simple server examples for you to experiment with.

Let's start by testing out the server example we have provided. Run the `server.py` program, giving it two command-line parameters, the first of which is the host on which the server is to run (which is going to be whichever machine you're on right now), and the second of which is a port number. For example,

```
python3 ./server.py localhost 8090
```

The server should respond by saying 'Echo server has started'. Now, typically using another shell, connect to the server with telnet, using a command such as

```
telnet localhost 8090
```

and after the usual preamble about addresses and escape characters has appeared, try typing a line of text in to telnet and pressing return. You should see the same line of text reflected back to you, but in capital letters. If you do this in several attempts, you may eventually note that if the server did not close cleanly in the previous interaction, the port may still be held. If this happens, try closing the server (e.g., type the 'ctrl' and 'c' keys simultaneously) or use another port altogether.

Now, have a look at the `server.py` code using your favourite text editor, and we will walk through what has actually happened here.

At the top of the file are some reasonably detailed comments on how the server implementation works; skip past these for now. Next, you will see the usual 'import' section, which in this case uses 'sys' (Python's way of getting access to boring operating system functions such as 'exit' to cleanly quit a program or access the command line parameters), and 'ex2utils' (which is the module we have provided and from which, in this case, we import the 'Server' functions).

The next line of the file creates a Python class called `EchoServer`, which inherits its functionality from the Server class we have provided in `ex2utils.py`. Look at the definition of the `EchoServer` class and the methods it contains, so that you can work out what these methods do. The file also contains an alternative server implementation called `EgoServer`. Now, look at the last few lines starting from the comment

```
# Parse the IP address and port you wish to listen on
```

These last few lines of code simply take the command line arguments for IP address and port, create an instance of our server, and start it going. The Server superclass we have provided then attaches itself to that port, and waits for incoming connections, calling the methods provided in `EchoServer` (`onStart, onMessage`, etc.) whenever something interesting happens, such as a client connecting, or a message being received. If you like, swap the instance of `EchoServer` for an instance of `EgoServer`, and try talking to the server once more with telnet; it should be fairly obvious what is going on here.

Now, let's return to the comments at the top of the file; these document all the different methods that can be overridden to make your own server implementation. Make sure you have read the comments thoroughly, and ask for help if they do not make sense.

1.3 Task 2.1: Your own basic server

The given server code already acknowledges when it starts by displaying a message on the screen. Now it's your turn to write some code. Your first task is to create a server (call it `myserver.py`) that simply acknowledges that a client has connected, that it has received a message, and that a client has disconnected. All you need to do here is print out a suitable message on the screen (server terminal window) for each event. To print a message in the server terminal, a function is provided in the parent Server class. This function is named `printOutput`, receives a string as input and prints the given message in the server terminal. You should be able to test all of these eventualities using telnet, as we did for `EchoServer` and `EgoServer`.

Hint: your server is going to be very similar to our `EchoServer` implementation but with a few more methods implemented (and these methods are explained in a lot of detail at the top of the file!)

Deliverables for Task 2.1 (1 mark)

Your server code must be able to respond when a client connects, when a message is received and when a client disconnects by displaying the *proper* message on the screen, using the `printOutput` function, as follows:

- The message to be printed when a new client connects **must** say *'new client connected'*.
- The message to be printed when a new client disconnects **must** say *'a client disconnected'*.
- When a message is received, the message must be shown in the server terminal.

1.4 Task 2.2: Counting clients

Still using telnet as a client, arrange for your server to remember the number of currently active clients, printing out the new total on screen every time a client connects or disconnects.

Deliverables for Task 2.2 (1 mark)

Your server code must be able to remember how many clients are connected at any one time, displaying a message on the screen in the following format:

{number} `active clients`

where *{number}* is replaced with the actual number (e.g., "2 active clients") each time a client connects or disconnects.

1.5 Task 2.3: Accepting and parsing commands

So now you have a basic 'generic' server; it will sit and listen for incoming connections, and handle messages sent to it in text by clients. The next task is to do something more meaningful with the messages. Later, we'll arrange for them to be sent back to clients to make a chat system, but first we need to establish a protocol for how we're going to interpret messages. It should be clear by now that anything you send (e.g. via telnet) to this server ends up appearing in the `onMessage` method as a string of text. We could, at this stage, build a simple server that simply 'reflects' all the messages that are sent to it from clients back to any clients that have connected (i.e. so a message sent from one person gets sent to all the others). But it would be nice to do something more sophisticated. For example, we should really allow clients to register a name, so you can tell who sent which message. It would also be nice to be able to send messages either to all members of the chat session, or privately to specific users (in which case, of course, we need to know their name!).

Achieving this means we'll need to encode some more meaning in the text that's being exchanged; for example, what 'screen name' is being used by which user, or the name of another user to which you'd like to send a private message. Following on from the style used by `http` and `smtp` servers, we'll simply establish a protocol based on text commands that look like this:

```
<COMMAND> <some parameters>
```

so we might have something like

```
HELLO Distributed Systems
```

Where 'HELLO' becomes the command, and 'Distributed Systems' becomes the parameters.

Write some code in the `onMessage` method to do exactly this, and test using telnet that you can indeed parse out a command and its parameters.

Deliverables for Task 2.3 (1 mark)

Your server code should be able to accept messages sent from the client in some sensible format, enabling interpretation of messages as they are sent to the server and displaying the commands and their parameters on screen.

1.6 Task 2.4: Designing the protocol

Step away from the keyboard; time to think, not write code.

Now, we come to the crux of this exercise, the design of a chat system. You should by now understand the capabilities of the server, which are a lot more useful for a chat system than our previous PHP-based server. We can accept connections from an arbitrary number of clients and, unlike the server from Exercise 1, remember that those connections exist. We can accept messages sent from those clients and are free to decide what messages are meaningful or useful.

This next task requires you to design a suitable set of messages that, together, form a protocol for your chat system. You can do this with pen and paper or, if you must, with a text editor - but don't be tempted to try to implement these until you're confident you have a sensible protocol

Your designed protocol should be able to handle at least the following:

1. Registration of new users with the server: each user needs to provide a 'screen name' to the server and ensure that the same screen name should not be assigned to any other user. When a new user registers with the server their name must be displayed on the server terminal in the following format:

   {username} `registered`

   with *{username}* being replaced with the actual username (e.g., 'Mike registered')

2. On a user request:

   (a) send a message to everyone that's connected to the server. The sent message should be printed in the terminal of all the registered users in the following format:

   `message from` *{username}*`:` *{message_text}*

   (e.g., 'message from James: hello everyone').

   (b) send a message to a specific user. The sent message should be printed in the terminal of the specific user in the following format:

   `message from` *{username}*`:` *{message_text}*

   (e.g., 'message from James: hi Alex).

   (c) send a list of registered/connected users.

3. Notifying the user if the command is not valid by sending the message: *'unknown command'*; and when the user is not registered by sending the message *'not registered'*. **Make sure you use precisely these messages**.

4. Some way of handling close connection requests from the users (see Task 2.6 description).

Also, you should provide a JSON file named `commands.json` that demonstrates how the protocol should be used (as shown in the example below). This file must correctly include your proposed command (in the `"command"` attribute under each task) for each task specified in the file (which should appear in your file exactly as shown below - i.e., `"register"`, `"send_all"`, `"send_one"`, `"online_users"`, `"close_connection"`).

When describing your commands in your `commands.json` file (in `"command"`), you may make use two special tokens to indicate where arbitrary values should be placed, as the example given for the

'send_one' task, below, demonstrates: (*%user%*, *%msg%*). These tokens will be replaced with the appropriate values during testing, e.g., *%user%* will be replaced with the username of an already registered user, and *%msg%* will be replaced with a message. See the `send_one` function below to see how you can use the special tokens.

```
// commands.json
{
    "register": {
        "description": "Register a new user",
        "command": ""
    },
    "send_all": {
        "description": "Send message to all the users",
        "command": ""
    },
    "send_one": {
        "description": "Send a message to a specific user",
        "command": "send_to %user% %msg%"

    },
    "online_users": {
        "description": "Get the list of all online users",
        "command": ""
    },
    "close_connection": {
        "description": "Disconnect the current client",
        "command": ""
    }
}
```

Hint: remember to design your protocol so that it can be tested from telnet; i.e. don't make it difficult to actually type your commands!

Deliverables for Task 2.4 (0 marks)

You should submit the JSON file with the correct name and format.

1.7 Task 2.5: Implement your protocol in the server

Now you should implement your protocol; most of the work will take place in your `onMessage` method, though you may find that you want to use the `onConnect` method too for initialising user-specific data (and perhaps even the `onStart` method for initialising server-wide variables). You have already seen (or written) enough code in this lab exercise to have covered all the important bits of functionality you need, e.g. decoding a message, sending a message back over a socket to a client, etc.

Deliverables for Task 2.5 (5 marks)

A server implementation (in a file called `myserver.py`) that responds to the protocol you designed in Task 2.4. You should update the same `myserver.py` file that you have already created for the first three tasks.

1.8 Task 2.6: Writing the client

By now you should have established a protocol, and tested that your server can respond sensibly to your protocol using telnet. This next task involves writing a Python client that makes using your server a little more friendly.

We have provided you with an empty skeleton for your client code, in `client.py`, which should be enough to get you going. Copy this file to make your own version called `myclient.py`. As with the server implementation, it's done by subclassing (extending) a Python class that we have provided. An important thing to note here is that the client skeleton deals with the problem of messages arriving at any time – even in the middle of 'raw input' – so the problem we saw in the previous lab is avoided. If you'd like to see how much effort and code is actually required to achieve what's apparently a simple thing, then have a look through the implementation of the Client class in `ex2utils.py`, but don't worry too much about all the details.

The basic task here is simply to implement your own protocol from the client's point of view, i.e. your client code should be able to

- Connect to the server.
- Register your screen name. If the screen name is already taken then it must try a different screen name until it successfully registers itself with the server.
- Repeatedly requests input from the user, and then sends the message typed by the user to the server. Note that the message should be interpreted and processed by the server (not the client).
- Close the connection. For example, If a user wants to disconnect then a message must be sent to the server to close the connection and the client should only exit after receiving the message ('Client exiting') from the server. **Make sure you use precisely this message**.

Deliverables for Task 2.6 (5 marks)

A client program (`myclient.py`) capable of interacting with your server.

VERY IMPORTANT: Update the header of this file by providing a text description of how to run your code and how it should be tested. In other words, describe how to install and run your code, as well as how to test it, by providing the order in which the commands of your protocol should be called to provide a complete demonstration of all the functionality provided within the protocol.

1.9 Submission instructions

You will need to submit `myserver.py`, `myclient.py` and `commands.json` files to finish this exercise. If you didn't get as far as creating your own `myclient.py`, just submit a blank file for this.

You shall submit your work in Gradescope (accessible via Blackboard). You are expected to submit a **zip folder** with the name `comp28112_ex2_username.zip` containing your code files (including the JSON file).

Don't forget to replace `username` with your own username consisting of 8 alphanumeric characters (e.g. a12345zz) before submitting your work.