|  |  |
| --- | --- |
| **COMP260 – Distributed programming** | **Worksheet 4** |
| **Experimenting with the Chat Service** | |

**Introduction**

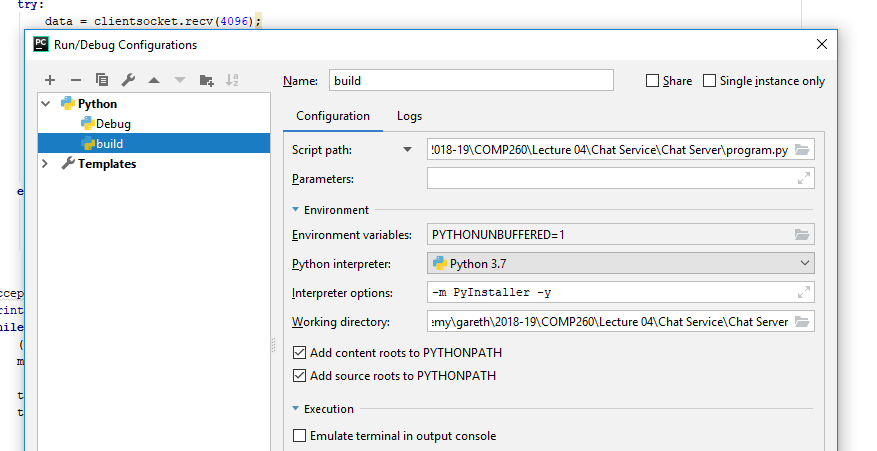
The goal of this worksheet is to experiment with the chat service that we walked through and demonstrated in this week’s lecture and use it as either a reference or starting point for your MUD service assignment. The chat service represents a fairly scalable and robust client server architecture that can be refactored into many different client/server roles.

**Networking**

Load the client and server projects and run them. They are set to use the local host (127.0.0.1) but they will work with any ip address (assuming the IT department hasn’t blocked the ports). Pair up with a colleague and get the apps to work across machines. This requires that both client and server have the same ip address and port defined. Go into a command prompt and type ipconfig, this will return the ip address of the PC.

**Make exes of your projects**

Python supports building to .exe files through pyInstaller. To do this, you need to install pyInstaller and create a new project configuration for build. It will need to have -m PyInstaller -y defined in the interpreter options and will make a build when you select the run (not debug) icon. On success, you will have a dist/<launch .py file name> folder with an exe in, this will run your project. If you are running the server app, the launch py file is program.py, so the folder is dist/program and the exe is program.exe. Copying the dist/program folder will allow you to distribute the application on other PCs without Python installed on them



If you make an exe of the client, it is easy to launch lots of instances, however, use –windowed in the build settings if you just want the PyQt window to show.

**Command processing**

In chat services, it’s very common to use #commands, a message that starts with a # to indicate to the server that the message sent is to be processed before being send to the other users.

Here are some common commands to think about implementing:

* **#name <my\_name> - change your name to something else if no-one is using it**
* **#ignore <user> - stop a user’s messages being displayed**
* **#unignore <user> - undo ignore**
* **#hide - don’t turn up in user lists**
* **#unhide – do turn up on user lists**
* **#private <user> - start a private only chat with user**
* **#public - stop the private chat**

Refactor the server to support these messages. By keeping all the application logic on the server, rather than the client, you are effectively creating a thin client that is just responsible for view (in the model-view-controller sense). This will make it far harder for your services to be hacked as all the important (model-controller) work is occurring on the remote server rather than the local client and socket connection.

**Refactor the apps into classes**

Last week, some of you were interested to see what the python projects would look like if the client and server functionality were in classes, rather than being global functions that run from \_\_main\_\_.

For the chat service, the server is probably the most useful code base to try this on (as the client is wrapped around the PyQt application stub and the threads pass a data object around already). Refactor the server and see if it makes the code easier or harder to work with.

**GUI Private messages**

Typically, people like to use the GUI to set up private messaging. It’s common to have a list of users in the chat client where an individual user can be highlighted and messaged. Rearchitect the client and the server to support this approach

**MUD refactoring**

The chat service is an ideal starting point for refactoring your MUD applications. Either use the chat service as a code framework for adding your MUD functionality or use the chat service as a reference for extending your existing MUD service.