COMP260 – Distributed programming	Assessed Worksheet 1
Chat Client	

Introduction

In this worksheet, you will analyse, design and architect a client application for an existing chat server application that will allow users to chat to each other. The server can be considered as a black box with a known protocol (discussed below). You will be given a bare bones client; your challenges are to: 1. Develop a suitable client architecture that will allow synchronous data transmission and reception using threads, 2. Allow data to be shared between UI and networking contexts using a suitable architectural approach, and 3. Design suitable protocol requests and responses that will enable several specific functions to be implemented across both client and server applications.

This worksheet assumes that the client will be developed in Python & PyQt (example code is provided), though any suitable application framework can be considered (C#/Unity, C++/Unreal etc). Please ask me for guidance before embarking with other frameworks.

Service Design

The service consists of a black-box server for windows which can be initialised with ip address and port details allowing it to be used for local testing (localhost) or to run as a remote service. The server application will output a debug log to console allowing developers and users to see any issues during development or operations.

The service also consists of a bare-bones Python client using PyQt. The client UI has been designed and implemented to a point (all widgets will report debug messages. Architecturally, the client uses PyQt's QTimer, creating a periodically called function 'timerEvent' that is called by the main Qt context, allowing Qt widgets to be updated. This is a standard Qt design pattern where threads can update data in their context which Qt can respond to in the timer event. Attempting to update Qt widgets directly from a non-Qt thread will cause access violations sooner or later and should be avoided. To stop access issues between the two contexts, a global Lock object is used to control access.

```python
import time
import threading

class SharedData:
    def __init__(self):
        self.counter = 0

sharedData = SharedData()
sharedDataLock = threading.Lock()

def doOtherStuff():
    while True:
        sharedDataLock.acquire()
        print(str(sharedData.counter) +' doing other stuff')
        sharedData.counter += 1
        sharedDataLock.release()
        time.sleep(0.13)

if __name__ == '__main__':
    thread = threading.Thread(target = doOtherStuff,args=())
    thread.start()

    while True:
        sharedDataLock.acquire()
        print(str(sharedData.counter) +' main thread')
        time.sleep(1)
        sharedData.counter += 1
        sharedDataLock.release()
```

```python
import time
import threading

class SharedData:
    def __init__(self):
        self.counter = 0

sharedData = SharedData()
sharedDataLock = threading.Lock()

def doOtherStuff():
    while True:
        sharedDataLock.acquire()
        print(str(sharedData.counter) +' doing other stuff')
        sharedData.counter += 1
        sharedDataLock.release()
        time.sleep(0.13)


if __name__ == '__main__':
    thread = threading.Thread(target = doOtherStuff,args=())
    thread.start()

    while True:
        sharedDataLock.acquire()
        print(str(sharedData.counter) +' main thread')
        sharedData.counter += 1
        sharedDataLock.release()

        time.sleep(1)
```

Depending on how locks are used, code may run fast (right) or slow (left). Run the two examples above to see the difference in leaving lock.release() until after a sleep().

Data communications

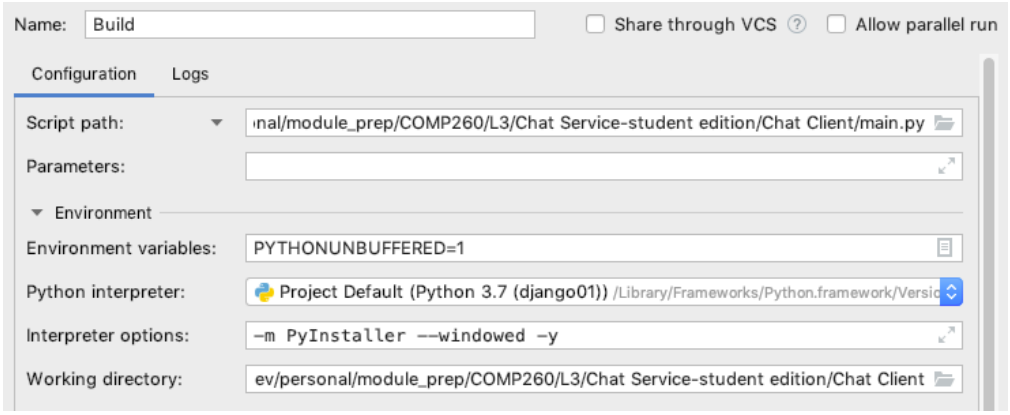
The client and server communicate using variable length packets (as discussed in workshop 1). Each packet consists of:

Size (2 bytes)	Payload (n bytes)

Payloads are different depending on context, but each payload will start with an ID (1-6) that defines different messages that can be sent between client and server. Data is managed as JSON.

ID	Content	Client -> Server	Server -> Client
1	Active user list 'users' - List of active clients Sent from server to client	Not valid	Client updates active user list 'clientList'
2	Request to change screen name 'name' – users new screen name Sent from client to server	Client sends 'userName' to server	Not valid
4	Send / receive private message 'target' – user 'msg' – message Sent from client to server Sent from server to client	Client sends 'userInput' if current 'clientList' entry is not 'All' Target is current 'clientList' entry	Client appends msg to 'chatOutput'
5	Send / receive public message 'msg' – message Sent from client to server Sent from server to client	Client sends 'userInput'	Client appends msg to 'chatOutput'
6	Send screen name to client Client screen name 'name' – user's screen name Sent from server to client	Not valid	Client sets 'userName' to 'name'

To aid with testing multiple clients should be used. This can be achieved by creating a Pycharm build using PyInstaller, e.g. :



To use PyInstaller, it must be added to the project through the package manager (in Pycharm).

The –windowed option will disable the standard output window and should only be used with PyQt. The -y option automatically accepts any requests during the build process. On successful completion, an exe of the python project will be located in the dist folder, it can then be run from either command line or as an icon in windows.

Tasks

1. Analysis
The client will be able to send messages to the server from the main Qt context, i.e. as a result of button presses and widget activity. However, to manage 1) connecting and maintaining a connection to the server and 2) receiving server messages, the client will need to maintain two threads: 1) the context that deals with whether the client is connected to the server or not and 2) a receive thread that will wait on server messages.

Create a suitable UML state diagram that details the client processes (Qt process, background thread & receive thread).

2. Design
Develop an implementation that will realise the analysis from part 1, in creating threads that will support the application & receive threads.

3.Implementation
Develop the client to implement the 5 different packets described above.

Final Submission
Combine the analysis and design documentation, and client source code into a folder and submit as part of 'Assignment 1 Computing Artefact' along with assessed worksheet 2 and your submission for the 'real-time gaming provision' part of the assignment.
Your work for both assessed worksheets and 'real-time gaming provision' parts of the assignment will be assessed in the viva session.

Marking Rubric – Assessed Worksheet 1: Chat Client

Learning Outcome Name	Learning Outcome Description	Criteria	Weighting	Refer for Resubmission	Adequate	Competent	Very Good	Excellent	Outstanding
Architect & Research	Integrate appropriate data structures and interoperating components into computing systems, with reference to their merits and flaws.	Analysis	25%	No analysis presented	Confusing or overtly simplistic UML diagrams	Rough UML diagrams	Fairly clear UML diagrams	Clear UML diagrams	Very clear UML diagrams
		Design	25%	No design presented	Code framework bears no relationship to UML No changes or changes to original client framework some of which are generally negative	Code framework bears some relationship to UML No changes to original client framework	Code framework follows UML Changes to original client framework some of which may be beneficial	Code framework closely matches UML Changes to original client framework some of which are beneficial	Code framework closely matches UML Significant positive improvements to original client framework
		Implementation	50%	No working software presented	Client supports a single packet type with no obvious issues	Client application supports some packet types with no obvious issues	Client application supports all packet types with some obvious issues	Client application supports all packet types with no obvious issues	Client application supports all packet types with no obvious issues Highlights issues in server application