

11/11/2014

Do Something Cool with Python

“An Seo!”

by

Darren Dowdall

A00202053

Network Management Year 3

Athlone Institute of Technology

Table of Contents

Do Something Cool with Python.....	3
Purpose of the project.....	3
Background to the project.....	3
Initial Plan.....	3
Plan B.....	4
Choosing A Layout Manager.....	4
Approach.....	4
Synchronous I / Woe.....	5
Locks, Queues, Semaphores and the Select() Method.....	6
Twisted Framework.....	6
7Back to the Client.....	7
User Manual.....	8
System Prerequisites.....	8
Server.....	8
Client.....	8
Login :.....	8
Main GUI :.....	9
Specification Tables.....	12
Client Side.....	12
ChatApp Class : Init Function().....	12
ChatApp Class : Initial-UI().....	12
ChatApp Class : Close-Event().....	12
ChatApp : Continue-Event().....	13
ChatClient Class : Init Function().....	13
ChatClient Class : ClientUI Function().....	14
ChatClient Class : Threaded Receive Function().....	14
ChatClient Class : GrabText Function().....	15
ChatClient Class : AppendText Function().....	15
ChatClient Class : KeyNow Function().....	16
MainLine.....	16
Server Side.....	17
ChatProtocol Class : Init Function().....	17
ChatProtocol Class : ConnectionMade Function().....	17
ChatProtocol Class : ConnectionLost Function().....	17
ChatProtocol Class : LineReceived Function().....	18
ChatProtocol Class : HandleRegister Function().....	18
ChatProtocol Class : HandleChat Function().....	19
ChatProtocol Class : BroadcastMessage Function().....	19
ChatFactory Class : Init Function().....	19
ChatFactory Class : BuildProtocol Function().....	20
MainLine.....	20
Client Application : Login GUI.....	21
Chat Application : Main GUI.....	22
Twisted Chat Server.....	23
References.....	24
Diary:.....	25

Do Something Cool with Python

Purpose of the project

The purpose of this project was to create a working chat like application using Python (2 or 3). For this to be achieved the project would require both a client and server side script to be written. The client script would be presented to the user in a styled GUI whereas the server would be solely command line based.

The client application would be responsible for :

- presenting the user with a GUI login box
- presenting the user with the “An Seo!” chat GUI
- passing data to the server
- receiving data from the server
- displaying new messages to the client interface

The server would be responsible for :

- controlling the connections to multiple clients
- passing messages between clients
- authenticating users

Background to the project

Initial Plan

The initial plan for this project was to build a voice recognition program using Python.

It would utilise one of Google's APIs to translate the speech received by the microphone (usually in a “.wav” or “.flac” file format) into a workable string of text that could be passed into the terminal to perform any basic terminal command.

However, problems with my build environment along with restrictions on data transmission size with Google’s voice recognition servers (only allowing for a few kbs to be processed at any given time) made this a lot tougher than it should have been.

Through some online research I managed to come across another speech API created by the people at CMU (Carnegie Mellon University) named CMUSphinx. It should have provided me with the tools I needed to translate the audio file into a string of text. But again it eluded me ; some evenings I would make progress with the problem from the evening before, only to be met with another issue almost immediately, and so, after three weeks of perseverance, I decided to switch to plan b.

Plan B

My second idea for “Do Something Cool with Python” was to create a network based chat application.

My first step on the road to my goal was to create the GUIs that would be required. For this I leveraged the power of Jython (rather than the traditional CPython) coupled with the Java Swing GUI Toolkit [1]. After many hours of experimentation with Swing, I felt happy enough to create the basic look of the client GUI.

Choosing A Layout Manager

Java Swing comes with many different layout managers available to the developer i.e. GridLayout, BorderLayout, GridBagLayout etc., all of which have their own pros and cons. With some previous experience using a geometry based manager (providing x/y coordinates to define placement of GUI widgets) I decided I wanted to try something a little different. In the end I choose the “GroupLayout” [4] manager to satisfy my needs. It works by adding widget components into both a Horizontal & Vertical grouping scheme in either sequential or parallel groups. Once the widgets have been added, the layout manager is able to calculate the desired position of the components based on their position within these groups.

After some time experimenting with the GUI tools [3][5][6], I was able to finalise my first draft of the client GUI. This left me with the simple task of adding the functional code to my project with the aim of tidying the GUI up once I had a working program.

Approach

From my perspective, there were two ways to approach this type of application:

1. Peer to Peer ; where the application serves as both a client and a server
2. Client / Server ; where the server acts as the central connection point for all clients and passes info between them

The peer to peer approach would require the coding of just one script that provided both client and server functionality between instances of the chat application, passing messages directly from one to the other. This seemed like it would be simple to code and to get working, but it lacked the “Cool” factor that a multiway chat application would not.

The final decision to choose the client/server approach enabled me to progress with further research into the inner workings of the program and how to approach some of the more intricate tasks involved within.

Synchronous I / Woe

For both the client and server scripts to work as they were intended there had to be a way for them to both receive & send data (almost) simultaneously. This would require the use of “Multi-Threading”; a separate process that splits from the main thread of the program to perform a given task. [8][9]

On the client side, the program needed to be able to listen for incoming messages from the server (on a separate receiving thread) as well as being able to send messages through an event call-back (button click) from the GUI.

The server side was where most of the issues arose during the development stage. The original implementation was designed to continuously listen on a default port for new incoming connections. When a new connection was established, the main thread would register the new user, provide them with a new unique port and then initialise a thread to deal with the new socket. The main thread then returned to listening for any new incoming connections. *It is important to note that there are two ways to implement threading in Python: the first is to just call the Thread method on an existing function, this is a good, functional starting off point but it is not as elegant as the second option which is to subclass the threading module and tinker with it's run method. Coupled with it's `__init__` method this gives the developer much more control over how the thread should function. This was the route that I originally chose to take.

With time and perseverance I managed to get the code to function as desired and I can safely say that during this period of playing around with threading I have gained more knowledge about Python, and programming in general, than I ever thought possible.

Once I had the handover functionality up and running, I was able to shift my focus to the sharing of messages between clients.

Each time a new message was received from a client, the server would identify the sender, attach the senders name to the message and then append the new chunk of data to a list for the other threaded instances to read from. This list variable had to be visible to all instances and it took quite some time to figure out how to make this happen. Eventually I managed to get this to work (without the use of global variables I might add) and now I had the functionality for not only creating new connections and threads, but also having these threads share the same variable state.

The final hurdle was to pass new messages to the client side as they came in. Each server thread would have to identify messages that were meant for them (messages not created by their client) in the list, it would then retrieve the message and send it across the socket to the client application which would then present it to the user. The issues that arose at this stage were ones of blocking sockets. *Looking back now, I think that I should have threaded the receive function inside each connection thread, in a similar manner to the way it is implemented in the finished client application, to handle incoming messages.

Locks, Queues, Semaphores and the Select() Method

After spending some time researching the ins and outs multi-threaded network programming (shout out to stack overflow) I came across a few different approaches to dealing with my blocking sockets issue. Locks, queues and semaphores fall into a category known as 'Synchronisation Primitives' [8]. Once called and assigned to a variable, these methods produce an object that can be acquired for a short period of time by each thread, this in turn initiates some processing before releasing the object and signalling the other threads to take their turn.

The 'select.select(socket_list)' method [7] tries to make this a little easier by treating each socket connection as a file. Once called, with a parameter in the shape of a list of available sockets, it pops out three individual lists for working with; a read_list, write_list and exception_list. The concept behind this is that each thread locks onto each list(s) in turn, checks it for whatever details it requires and then, like the synchronisation primitives mentioned earlier, releases it for the next thread to process. This should have worked (and it kind of did work in the end) but the issue with the blocking sockets prevailed.

After being stuck at this point for a decent amount of time, I decided to look into alternative methods for creating a server to handle my connections. It was by chance that I stumbled onto the 'Twisted Python' API, which was the pivotal point of the project.

Twisted Framework

Twisted is an event driven network programming framework that supports many different network protocols. It employs instances of "Factories" [12] to build, manage and maintain the input/output features required in such networking programs as the Chat Server.

The resulting code is implemented using call-back functions which are called by the framework once a given event has occurred. To pick up on these events a reactor object [13] is created. The reactor is then attached to the server object and told to listen for events that the program would be interested in.

Under the hood, the reactor uses a polling mechanism to produce the desired I/O features. Each time an event happens on the transport (which is basically a twisted term for a live readable/writable connection) the reactor calculates which function the event is associated with as well as what data it should then provide to the function.

After a brief introduction to Twisted (utilising some amazing online tutorials [10] coupled with a great book* [2] on the subject) I was able to construct a basic TCP based server application which was capable of connecting multiple telnet clients In a chat like scenario.

* The final Chat server code is based on an example extracted from this book.

Back to the Client

With the twisted server up and running (accepting user instances over telnet) the only real work left to do was to figure out how to connect the client into the new implementation of the server. At first I tried to connect directly in using standard streaming sockets as in the applications previous iteration, however this did not work as expected (that's not to say that it is impossible, but with time ticking away I decided to approach this in a different manner).

Next step was to try build my client application using twisted, but alas it was not meant to be as Jython would not accept the importing of some of the twisted modules (even though it was installed on Python 2 which Jython uses).

This got me to thinking; If I can make telnet connections from the terminal, why not make them also from the client application. After a quick search online I discovered a python module called “telnetlib” [14].

A short time later “An Seo” was born (and I did a funky dance around my kitchen).

User Manual

System Prerequisites

Server

The server application requires the package “python-twisted” to be installed before it can be executed.

The server is written for python 2.7 and can be executed from the terminal using the command
“*python path/to/file.py*”

** To shut down the server simply close the terminal window or ctrl+z. Once started, the server requires no interaction on behalf of the user.*

Client

The Client Chat application requires that both Java and Jython are installed before it can execute. It also requires the “telnetlib” module to be installed, which is provided by default in most Linux operating systems.

The client application can be executed from the terminal using the command
“*jython path/to/file.py*”

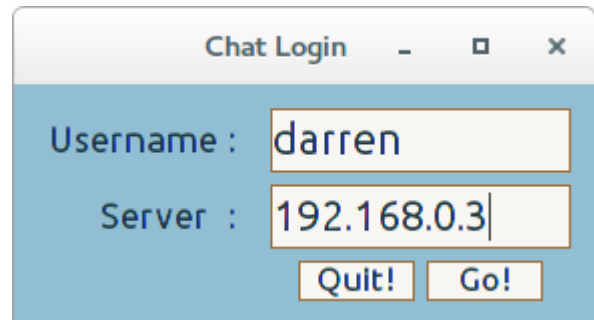
Login :

Upon initialising the client application the user is presented with a window consisting of two entry fields :

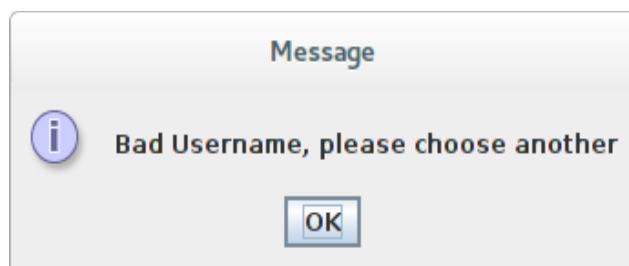
- 1: requires the input of a username
- 2: requires the IP Address / Hostname for the server

If either of these details are unacceptable (e.g. either the username is already taken or there is no connection available) then the user is presented with a Dialog Box explaining the error as well as allowing them choose what they would like to do next (re-enter details or quit).

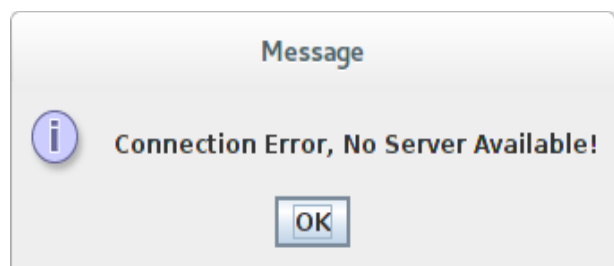
If all details are validated, the application proceeds to the main client GUI whilst hiding the login GUI behind the new window.



Login



Bad Username



Bad Host

Main GUI :

After validation the user is presented with the client GUI window. At this point their username is registered with the server application and they can proceed to chat with any other connected clients. If a new message is received by the client program, it is immediately displayed in the main content area, formatted in a way that identifies the client the sent the message as well as a delimiter to show where the message begins and ends.

To write a message the user types into the lower text field and clicks the send button. This event relays the message to the server who then passes the new message to all clients except the one that created it. This message is also displayed in the main content area as soon as the button is clicked.

While connected to the chat application, the user will be notified of any new clients who join the chat with a message in the content area. If a user departs the chat a message is broadcast to all current clients notifying them of the departure. This is also displayed in the content area.

To leave the chat, the user can click on the quit button or the default close window button. Once this event happens, the server removes the clients name from its database and broadcasts the departure of the client to all other connected clients.



Main GUI

Deafault Main Window & Initial message display

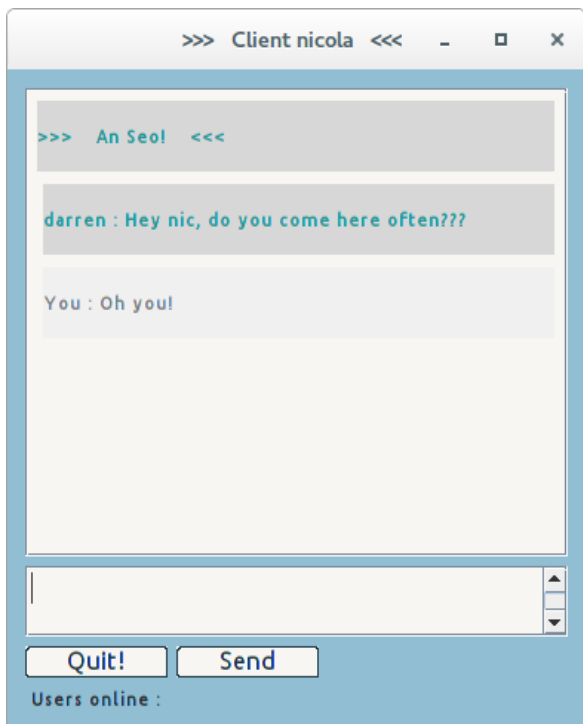


New Client

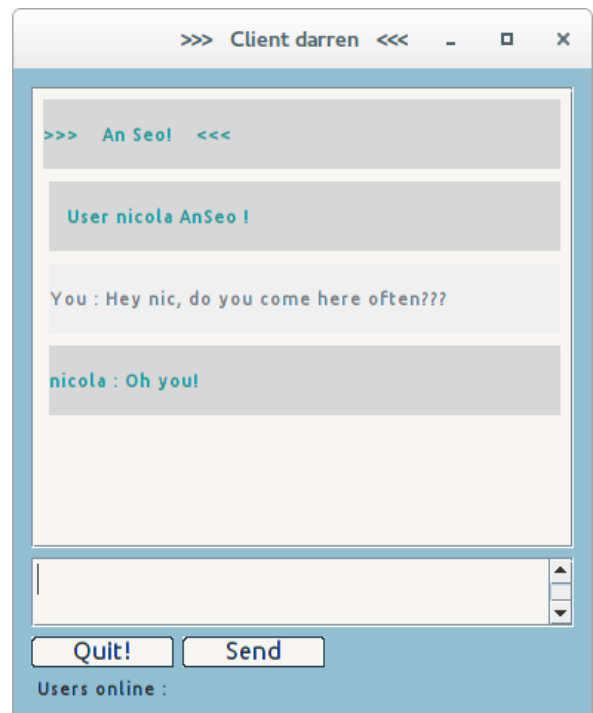


New Client Notify

New Client Joins & User Notification Received

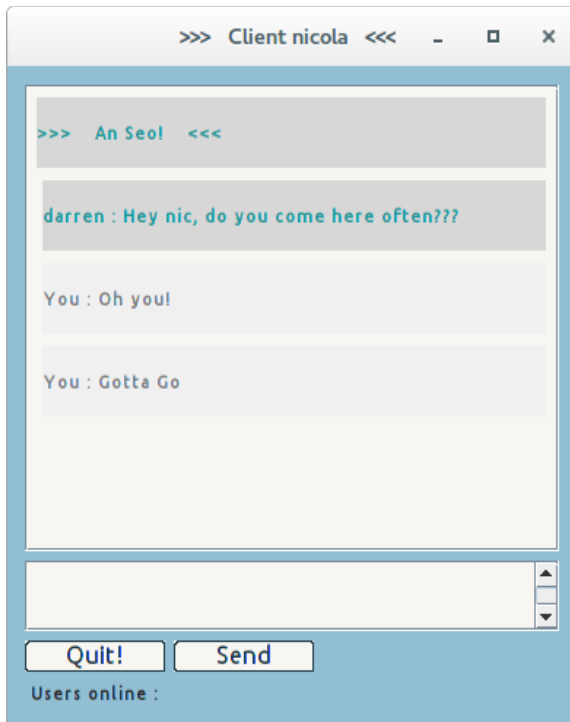


Nicola Chatting

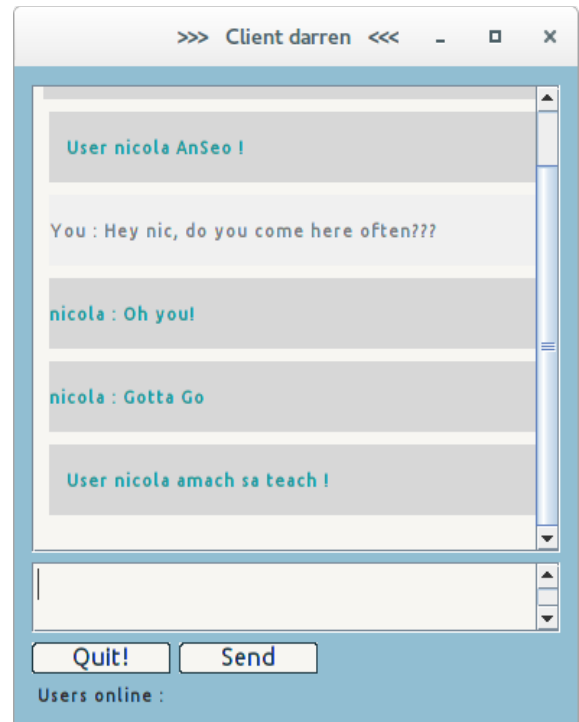


Darren Chatting

Client nicola receives the message from client darren & responds

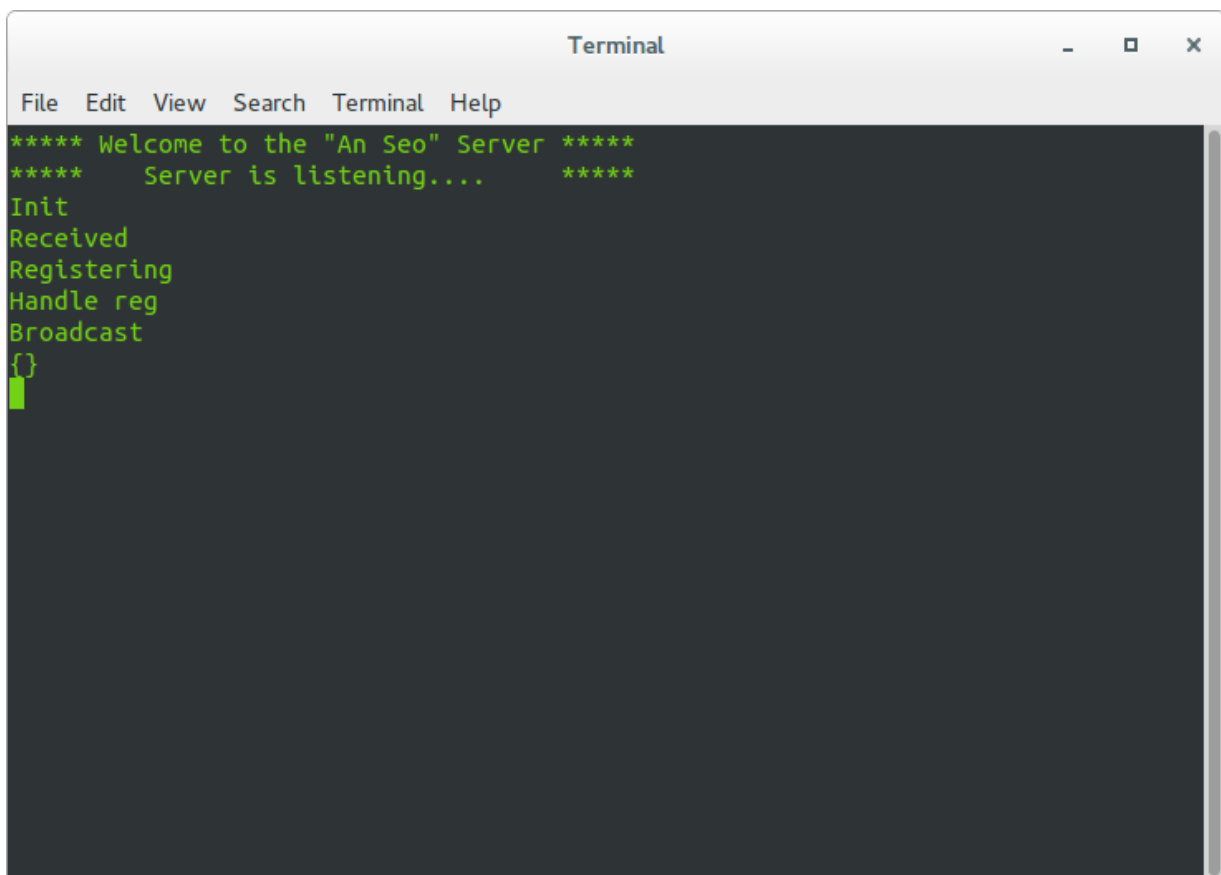


Client Nicola leaves



Departure Notify

User nicola departs chat & user darren receives notification



Server Side

Server Receives, Registers new users and handles chat message transportation

Specification Tables

Client Side

ChatApp Class : Init Function()

Input	Processing	Output
Self Instance	Call the super class Initiate the Login GUI	Base Class Instance

Pseudocode:

Call to super method to initialise the base class

Call to Initial GUI function

ChatApp Class : Initial-UI()

Input	Processing	Output
Self Instance	Create Layout Create Widgets Define widget actions Component Placement	Login GUI

Pseudocode:

Build the GUI:

Create widgets

Attach callbacks to the buttons

Add widgets to the Content Pane using group layout manager

Set the defaults for the GUI window

ChatApp Class : Close-Event()

Input	Processing	Output
Self Instance Event (on quit button click)	Display Goodbye message	Exit Application

Pseudocode:

If the users clicks “quit” button, close the application

ChatApp : Continue-Event()

Input	Processing	Output
Self Instance Event (on go button click) or Event (on return keypress) Strings from Text Fields	Grab strings from entry fields Try make a telnet connection <ul style="list-style-type: none">• send username• acknowledge connection• Display Dialog if username taken• Call Main GUI Except when none available <ul style="list-style-type: none">• Display Dialog	Telnet connection Dialog displayed Call to Main GUI

Psuedocode:

If the user clicks the “go” button:

 Grab the text from the entry fields

 Try to establish a connection:

 Send the username to the server

 Process the response

 If the response is a bad username:

 Present a dialog and ask the user to re-enter the details

 Hide the GUI & Call the main App

 Except If there is no connection available:

 Then present a error dialog box to the user

 Reset the text and allow the user choose what to do

ChatClient Class : Init Function()

Input	Processing	Output
Self Insance Username Host port telnet connection	Call the super class Initialise Variables Thread Receiving function Call the Main Client UI	Threaded function Main GUI Instance variables

Psuedocode:

Call the super method to initialise the base class

Set the instance variables

Create the threaded function, set to daemon and run

Call the Main UI

ChatClient Class : ClientUI Function()

Input	Processing	Output
Self Instance	Create Layout Create Widgets Define widget actions Component Placement	Main Chat GUI

Psuedocode:

Build the GUI:

Create widgets

Attach callbacks to the buttons

Add widgets to the Content Pane using group layout manager

Set the defaults for the GUI window

ChatClient Class : Threaded Receive Function()

Input	Processing	Output
Self Instance Telnet connection Incoming data	While connection available: Try receive message <ul style="list-style-type: none">Pass to AppendText function Except if there is no connection <ul style="list-style-type: none">Pass	Call to AppendText function

Psuedocode:

While the connection is alive

Try to receive a message from the connection:

If a message is received:

Call the AppendText function to display message

ChatClient Class : GrabText Function()

Input	Processing	Output
Self Instance Event (on send button click) String from text area	Retrieve string from text area Do not send empty string Set the field to empty & gain focus Write the data to the connection	AppendText function call Outgoing data

Psuedocode:

On “send” button click:

Retrieve the text entered in the text field

If it is an empty string:

 then drop the message

Otherwise:

 Format the message

 Pass the message to the AppendText function to display message

 Reset text and gain focus in the message area

 Send the message out the connection

ChatClient Class : AppendText Function()

Input	Processing	Output
Self Instance Message User	Create label & format depending on user Set label attributes Display label in content area	New message label Displayed in window

Psuedocode:

Create the message label

If user is current user

 format appropriately

Otherwise no user name

 format appropriately

Set the label attributes

Insert the new label to the content area

repaint the window (just to be sure)

ChatClient Class : KeyNow Function()

Input	Processing	Output
Self Instance Event (on return key press) String from text area	Create a Key class object Pass the Key object the event Test the returned value Call the grab text function	GrabText function call New class object Event

Psuedocode:

On return keypress (inside text area) call KeyNow function

Create a Key class object

Pass the objects KeyPressed function the event and store the returned value to a variable

If the value is equal to 10 (keycode for return key)

 Call the GrabText function

MainLine

Psuedocode:

If the name of this instance is “__main__”

Call the ChatApp class

Server Side

ChatProtocol Class : Init Function()

Input	Processing	Output
Self Instance Chat Factory Object User Dictionary	Initialise instance variables	Instance variables

Psuedocode:

Reactor Event : New Incoming Connection :

 Initialise the instance variables

 Set the initial user name to “None”

 Set the initial user state to unregistered

ChatProtocol Class : ConnectionMade Function()

Input	Processing	Output
Self Instance Reactor object Connection object	If New user Send Acknowledge Set Client state to New User	Client State Outgoing data

Psuedocode:

Reactor Event : Connection Complete

If the state is that of a new user:

 Send the client a greeting

ChatProtocol Class : ConnectionLost Function()

Input	Processing	Output
Self Instance Reason/Exception variable Reactor object Connection object User Dictionary	Remove the user from the user dictionary Notify all clients of departure	User removed from user dictionary Outgoing notification to clients

Psuedocode:

Reactor Event : Connection Dropped

Check if the username is stored in the user dictionary

Remove it if it is Broadcast a Departure message

ChatProtocol Class : LineReceived Function()

Input	Processing	Output
Self Instance Line/Connection object Reactor object State variable	If client state is new user <ul style="list-style-type: none">pass control to handle register function Otherwise <ul style="list-style-type: none">pass control to handle chat function	Handle chat function or Handle register function

Psuedocode:

Reactor Event : Data received on the transport

If this is a new client:

Handle the registration of the new client

Otherwise:

Handle the chatting function of the server

ChatProtocol Class : HandleRegister Function()

Input	Processing	Output
Self Instance Line/Connection object Reactor object User Dictionary	If name taken, notify client Else acknowledge <ul style="list-style-type: none">Send greetingSet the instance name to usernameAdd client to user dictionarySet the client state variable to Chatting	Client Notification Username added to dictionary State set to chatting

Psuedocode:

If the username already exists on the server:

Request a new name from the client

Otherwise:

Send an Acknowledgement & Greeting

Set the instance name to the that of username received

Update the user dictionary

Set the state to that of a registered client

ChatProtocol Class : HandleChat Function()

Input	Processing	Output
Self Instance Line/Connection object Reactor object Message string	Format the message Pass control to broadcast message function	Formatted Message Broadcast Message function

Psuedocode:

Format the message for transport

Broadcast the message to all clients

ChatProtocol Class : BroadcastMessage Function()

Input	Processing	Output
Self Instance Line/Connection object Reactor object Formatted Message User Dictionary	For each item in the dictionary that is not this client <ul style="list-style-type: none">Send the new message	Outgoing data

Psuedocode:

For each client in our dictionary:

 That is not this client:

 Send the new message

ChatFactory Class : Init Function()

Input	Processing	Output
Self Instance	Create the user dictionary object	Display server up message

Psuedocode :

Initialise the shared state dictionary

Display server greeting

ChatFactory Class : BuildProtocol Function()

Input	Processing	Output
Self Instance Address	Return a ChatProtocol object	ChatProtocol object

Psuedocode :

Call the protocol class (instance) for each new client

MainLine

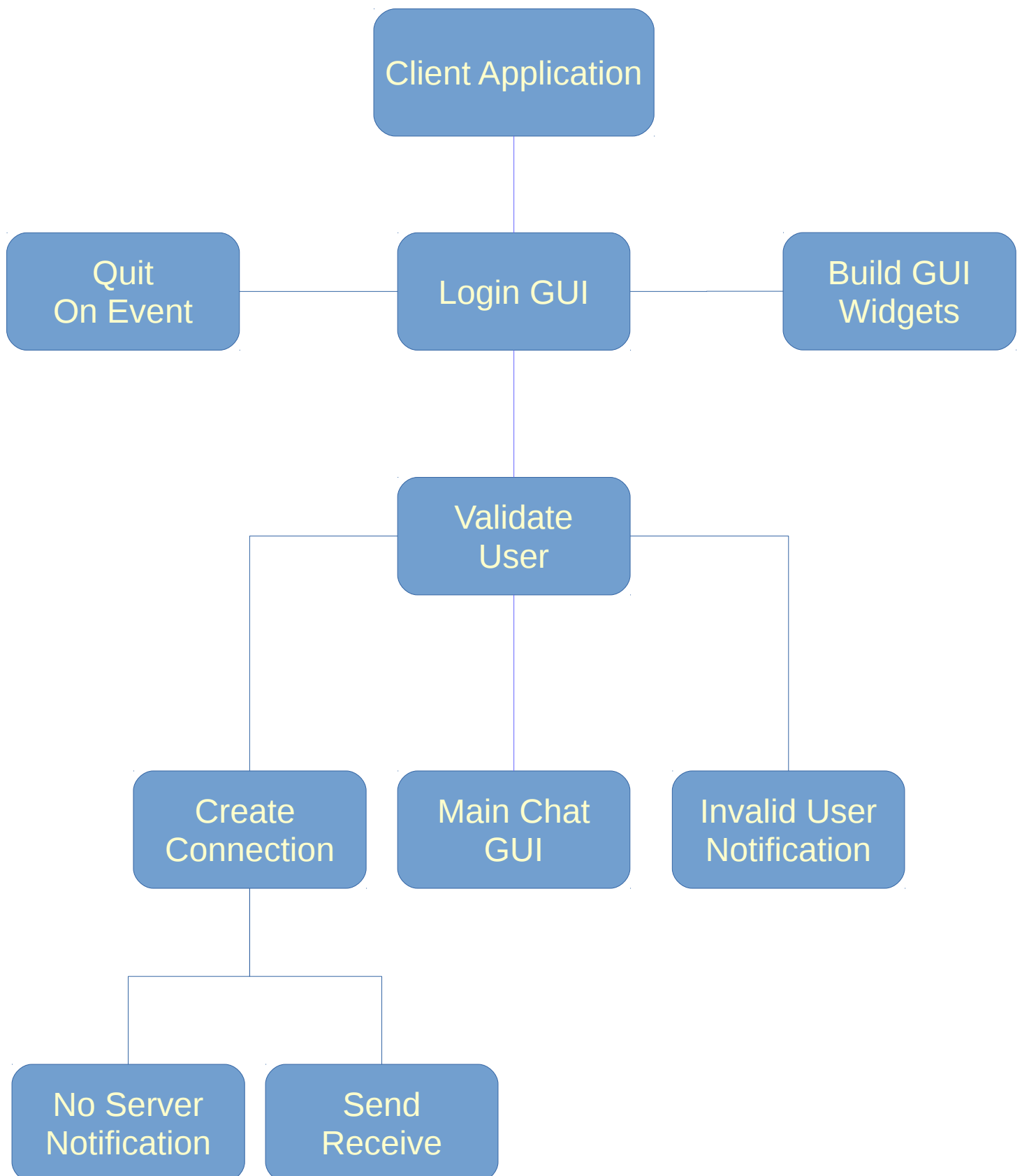
Psuedocode:

Create a Factory instance

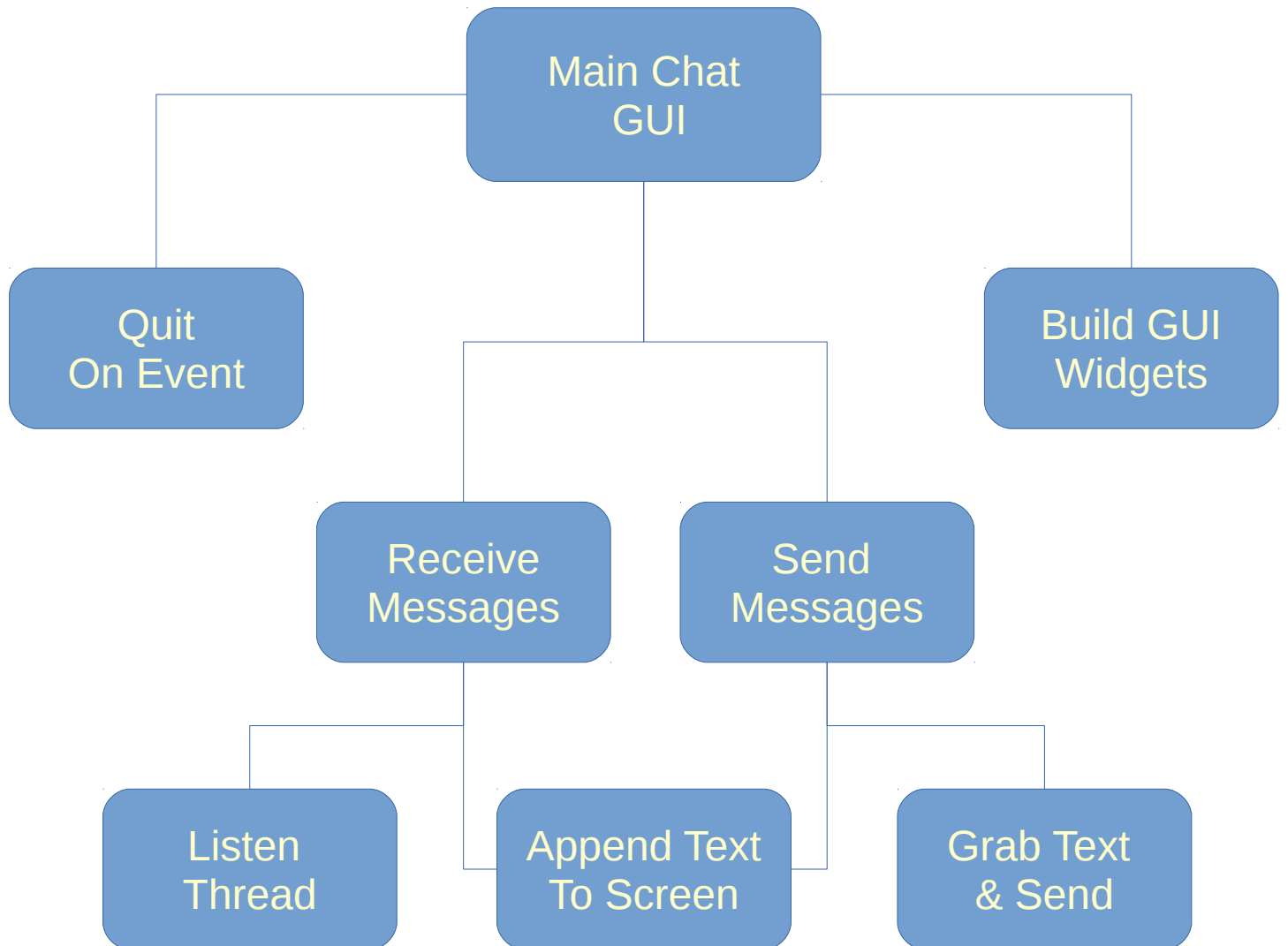
Notify the reactor to listen for incoming TCP connections on the default port

Initialise the reactor object

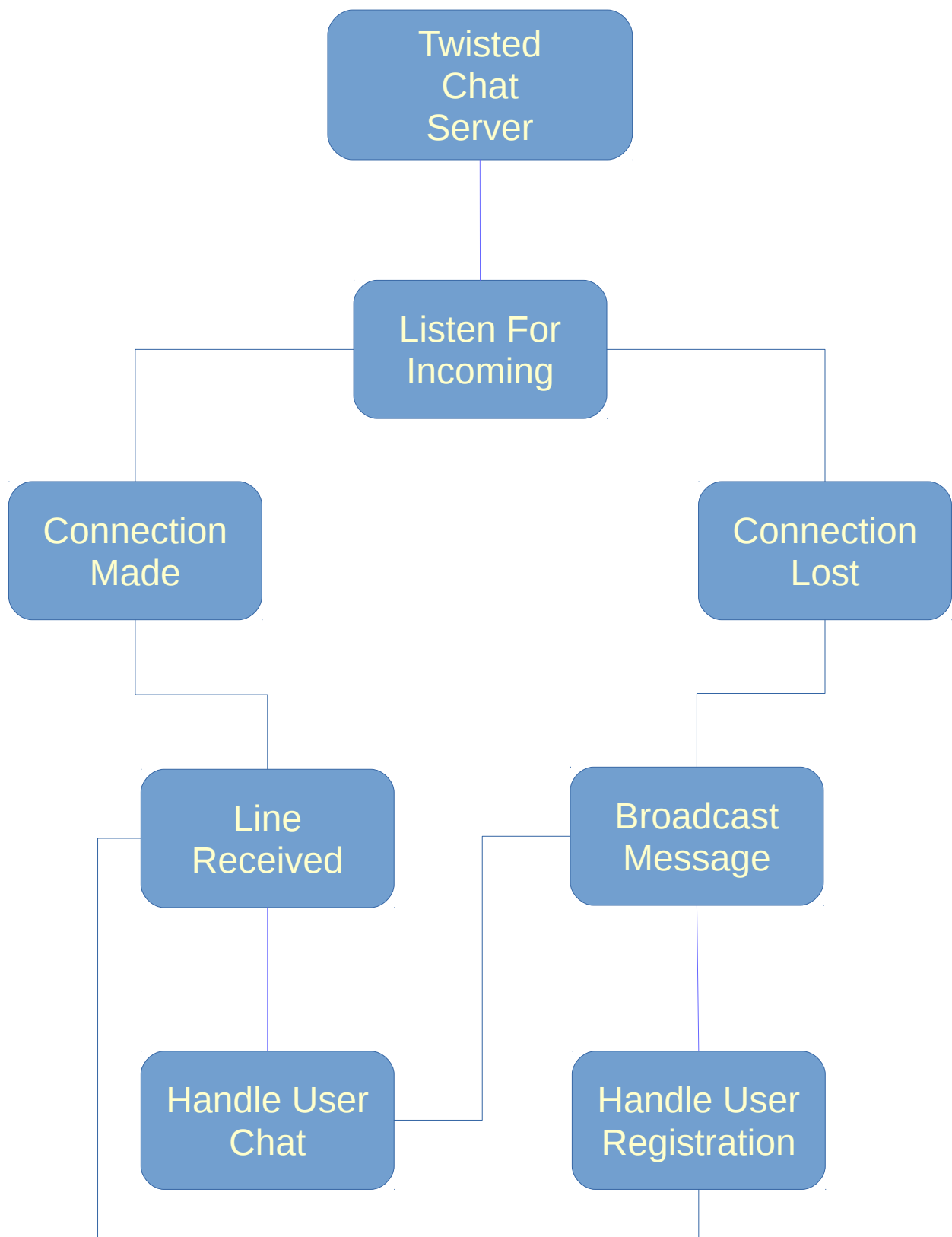
Client Application : Login GUI



Chat Application : Main GUI



Twisted Chat Server



References

1. Samuele Pedroni and Noel Rappin (2002). *Jython Essentials*. Ireland: O'Reilly. 144-155.
2. Jessica McKellar and Abe Fettig (2013). *Twisted Network Programming Essentials*. 2nd ed. Ireland: O'Reilly. 3-51.
3. *Layout management in Jython Swing*. [ONLINE] Available at: <http://zetcode.com/gui/jythonswing/layout/>.
4. *How to Use GroupLayout (The Java™ Tutorials > Creating a GUI With JFC/Swing > Laying Out Components Within a Container)*. [ONLINE] Available at: <http://docs.oracle.com/javase/tutorial/uiswing/layout/group.html>.
5. *Text Area Scrolling « Java Tips Weblog*. [ONLINE] Available at: <http://tips4java.wordpress.com/2008/10/22/text-area-scrolling/>.
6. *How to add JPanels to JScrollPane? - Stack Overflow*. [ONLINE] Available at: <http://stackoverflow.com/questions/26574496/how-to-add-jpanels-to-jscrollpane>.
7. *Concurrency in Python — training.codefellows 1.0 documentation*. [ONLINE] Available at: <http://cewing.github.io/training.codefellows/lectures/day09/async.html>.
8. *Understanding Threading in Python LG #107*. [ONLINE] Available at: <http://linuxgazette.net/107/pai.html>.
9. *Multithreading - Sending messages between class threads Python - Stack Overflow*. [ONLINE] Available at: <http://stackoverflow.com/questions/14508906/sending-messages-between-class-threads-python>.
10. *An Introduction to Asynchronous Programming and Twisted | krondo*. [ONLINE] Available at: <http://krondo.com/blog/?p=1209>.
11. *An Introduction to Asynchronous Programming and Twisted | krondo*. [ONLINE] Available at: <http://krondo.com/blog/?p=1522>.
12. Twisted. 2014. [ONLINE] Available at: <https://media.readthedocs.org/pdf/twisted/latest/twisted.pdf>.
13. *twisted.internet.interfaces.IReactorTCP : API documentation*. [ONLINE] Available at: <https://twistedmatrix.com/documents/14.0.2/api/twisted.internet.interfaces.IReactorTCP.html>.
14. 2014. *telnetlib — Telnet client — Python 2.7.9rc1 documentation*. [ONLINE] Available at: <https://docs.python.org/2/library/telnetlib.html>.

Diary:

I have done as much research as is humanly possible, given my knowledge and time constraints. Most of the early focus was based around gaining access to the Google Voice Recognition API, however this has proved to be difficult. Conflicts with modules and versions of the Python interpreter have driven me demented but, as it turns out, I would only be able to upload a tiny amount of data to the google servers at a time. Not much use really. I'm currently trying to get another module (sphinx/sphinx4) to work for me, either using pure python or jython.

I'm also currently delving deeper into GUI kits but it is pretty tough going considering the limited knowledge I have on the subject. It's fun though, and entirely do-able, so for the foreseeable future, I'm gonna stick with it.

Darren out.

[Edit]

I've spent the last few days messing about with swing gui through jython (see chat.py in upload area for proof) just so I have something to show for the time being. Still haven't given up on voice recognition.

[Edit-19-Oct]

Dear diary, sorry for not updating you lately, but I've been busy as hell with this project. Over the past 2 weeks I've spent an unhealthy amount of time coding, (48 hrs +), trying to figure out how to share variable states between threads. Last night I decided I've had enough of trying to get thread queues working for me, and so I've started to use a pair of lists for sharing messages between the running threads.

I'm now at a stage where the client apps can see what messages are being sent by other clients, however, this only displays in the message area of the client **after** the send button is clicked (instead of dynamically whenever a message is sent).

For example; Client A receives a text entry from it's user (this text is added to the list): Client B doesn't update it's message area with this new text **until** it's user sends a message! Very frustrating, but to be fair, it is kind of working so that's something.

I'm uploading the latest iteration of the project to show the progress I've made (I'm also emailing all my other tutors to apologise for not doing any of their work over the last two weeks!).

Peace out.

[Edit-29-Oct]

Dear diary, I'm so happy!, I managed to get my program fully working yesterday, Yay! I'd been racking my brain over the past 2 weeks trying to figure a way to work with concurrent thread handling (server side) using the "select.select()" method, with little or no joy I might add. However, after some extensive research into twisted python, I realised I was making it far too difficult on myself. So I created a twisted python server (with the help of a good book on the subject) that

handles all of the more intricate parts of shared state when dealing with threads or multiple instances. It worked great when using telnet to send messages back and forth but, when trying my client (using sockets at the time) the program would falter after only one round of message sending! But telnet worked! So I got to thinking; there must be a way to piggy back on telnet from python. Low and behold the telnetlib module!

It took a bit of playing around with to get it working, but once I'd gotten to grips with it, it was simple enough (note: message needed to be appended with "\r\n" to register a return entry from the client). Still there were problems on the receiving end. In steps threading again to continually monitor the connection for incoming data. It calls the recvFunction() which takes care of incoming data and appending it to the main text area in the GUI; the sending of data is taken care of in the main UI (click on send button) repeatedly grabbing the text entered and sending it on its way, so with both in place a conversation can take place. :)

Now my program features not a single socket and still manages to perform the tasks thrown at it. Oh, and it scales really well, in fact I'd say this thing could handle hundreds of attached clients (provided my laptop doesn't give out).

The new scripts will be uploaded shortly, thank you for your patience.

The Twisted server runs on python 2.7 and the client runs on Jython. To run the program some modules may have to be installed:

twisted python is here: <http://twistedmatrix.com/trac/wiki/Downloads>

jython is here : <http://www.jython.org/download.html>

There may still be some errors for modules I've forgotten, but the interpreter will spit out whatever is needed, and Google is your best friend.

Now the only thing left to do is making it purdy!

Gotta run,

Taxi !

[Edit 11-Nov]

Dear diary, I've been busy tidying up the GUI for the last week or two. I managed to get a few little bugs ironed out and I think I'm just ready to submit. I have also just finished typing up the documentation for the project. It took a lot more effort than I thought but I enjoyed doing it, it kind of helped a few things stick a little better, especially with some of the inner workings of the twisted api (it's like some sort of voodoo that goes on in there!). So this may be my last edit. The only thing left for me to do is the presentation and I'd like to get that sorted too, that'd really clear a path for exam revision, so it may be ready by the start of next week.

I'm going to upload the latest iteration of the programs, but these will probably be the final edit as I think i've dedicated enough time to the coding of the project already, and there are a few other subjects that require my attention.

Bub-Bye now.

[Edit 23-Nov]

Dear diary,

I have uploaded the final submission, documents and all, of my chat application. There are a few differences from the last iteration; the return key can now be used to send messages from the client (instead of having to click the send button each time), but the main change is the message window which now displays a neat little box for each message (colour coded to differentiate between the client messages and incoming messages). It looks great and I'm chuffed to bits with it, [this](#) was pretty much my reaction when it finally worked out.

So this is me signing off for the final time. It's been a long, and sometimes frustrating, road, but above all else, it has been extremely rewarding. I've expanded my knowledge ten fold over the last few months and I'm hungry for more. Onwards and upwards.

Darren.