# SFC Python Project

## Introduction

The goal of this project is to create a chat system with a client-server model.

The client–server model of computing is a distributed structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

– Wikipedia on the Client-Server Model

In the case of a chat server, the client represents a prompt where a user may enter text to send to a server. If they send a message the server, aware of all clients which have connected to it, will distribute that message to everyone.

To accomplish this, each client will send carefully formatted messages to the server which include all the information it requires to perform its job. For example, when a client first connects to a server to make it aware of its existence (ie. when a user opens a chat prompt) it will send it a connection message. If a user needs to authenticate with a server, it will send it an authentication message which will provide a password.

To communicate between one another, clients and servers will need some form of connection. The client will need a connection to the server, and indeed conversely, the server will need a connection to the client. ‘Sockets’ provide general connections of this kind: they provide the ability to read and write between two processes regardless of whether they are on one machine or distributed across many. Here sockets will send binary data – one byte at a time – so messages sent across them will be ‘packed’ (converted into binary).

## The Message Protocol

Messages on the chat system to be produced should be sent in the following format:

message\_type, message\_text\_length, message\_text

Where the *message\_type* is an integer value representing *NORMAL, JOIN, USER, PASS, DIRECT, COMMAND, SERVER* messages.

Their meaning will become clear as you move through the steps. You do not need to complete every task, so do not worry if you haven’t used some of these the message types.

*message.py* has been provided and will send messages in the appropriate format via a given socket. See the skeleton programs provided in step 1 for example usage. You will not need to modify message.py until later in the exercise, and even then the changes required are minimal.

## The Tasks

The goal of the exercise is to demonstrate competency with python and programming in python. This *could* be accomplished with a very small code example, but it is likely we will need to see more in order to make a fair assessment. You should therefore work through the tasks we have listed below and complete them with the criteria we have listed in mind.

After each step you should make a copy of your work, as you will be submitting the code for the current step you are working on (whether it works or not) as well as the code for the previous step that you finished. Ensure everything has a back up.

Steps 1 – 4 are preliminaries for which we have provided solutions if you need them. They are there to get you comfortable with the initial problem and it’s strongly recommended that you work through them.

Steps 5 to 14 are the key tasks for the exercise and are very roughly in order of difficulty -- but you can do them in any order you like.

Advanced programmers may want to consider using object-oriented or functional features of python.

## Preliminary Tasks

### Step 1 - Creating a simple server and client

You should start with the client.py and server.py program skeletons provided *(src/step0)*, as they show how to make use of message.py to send and receive messages.

In this step you should make the server program create a socket, listen for a single client connection, accept it, then wait for and print messages received from it in a loop. The client program should connect to the server, then send messages from the terminal to it in a loop.

You should use the message.py file provided.

### Step 2 - Getting the server to respond

After the client sends each message, it should now try to receive a message from the server.

After the server receives each message, the server should send it straight back to the client. The server itself may want to print this message to show what it is receiving.

Both the server and the client are alternating between sending and receiving here.

Use the solution to step1 (*src/step1*) as a guide to completing this and further tasks.

### Step 3 – Allowing multiple clients using non-blocking operations

Use python's *select* statement in your server's receive loop to either read from client sockets or accept a new connection and add it to your list of clients.

Make sure the server still only responds to each message by sending it back to the client that originally sent it. At this stage the client is unable to receive multiple messages at a time.

### Step 4 – Broadcasting messages to allow cross-client communication

Use python's *select* statement in your client's send loop to either read from stdin (the keyboard) or read a message from the server socket.

Get the server to relay each received message to all clients now, not just the one that sent it.

NB. Using *select* on stdin avoids using functions like raw\_input, which blocks program execution until the return key is pressed.

## Key Tasks

It is now recommended that you work from the Step 4 solutions provided *(src/step4)*. However, you should not proceed beyond this point without understanding each of the previous steps.

You may still use your own files providing they are functionally equivalent to our solution, that is, you ensure:

* multiple clients can connect to the server
* all messages received by the server are broadcast to all clients
* read and write operations do not block each other (the *select* statement works)

The following steps are roughly in order of difficulty, but you can do them in any order you like.

### Step 5 - Logging

All messages to and from the server are logged to a text file.

### Step 6 - Named clients

Clients should now be able to specify their name when the program first starts and have this sent to the server. You need to extend the message.py file provided to include the USER message type. You will also need to create a data-structure capable of storing the information about each connected client. For now this will just be the socket they are connected with and their name.

### Step 7 - Logging in

Clients can login with a set of credentials (hard-coded at this stage). Now you will be using the USER and PASS message types.

### Step 8 – Chat Rooms

Clients can join chat-rooms using commands (e.g. by sending messages such as “\join my\_chatroom”). These messages should use the JOIN message type.

### Step 9 - Direct Messages

Clients can send direct messages to individual users. These should use the DIRECT message type.

### Step 10 - Commands

Clients can use commands to request information from the server such as uptime, number of connected users, chat-room lists etc. It would be wise to stick to the same structure for command as with the previous step. E.g. sending messages such as “\uptime”. These messages should use the COMMAND message type.

### Step 11 - Registration

Clients can register with the system. List of usernames and passwords should be in memory at this stage.

### Step 12 - Administrator Commands

Make an administrator account that can kick users from a chat-room or the whole server. Add any other commands you like.

### Step 13 - SSL

All communication between the clients and the server is encrypted.

### Step 14 - Multi-server

The interpretation of this is left open.

## The Report

Your report can be in either txt or pdf format, and should include the following:

* Your name.
* How your program is run (e.g. is it just “python client.py” and the same with the server, or do we need to specify arguments).
* Which of the steps you've completed and which of the steps you're currently working on. If your program doesn't run at the moment, let us know and point us to the most recent working copy of your project (as well as the current non-working copy).
* What you found most challenging.
* What you're most proud of, if different from the above.
* What you don't like about your implementation and why (e.g. data structures, algorithms, object orientation, system features, or general code style).
* What you would do if you had more time.

Try to be specific in the above two sections – avoid stand-alone statements like “I would improve the algorithm for x because it's bad”. Instead say why you think x is bad and if possible, how you would go about improving it. If you're talking about a new feature, explain in a few sentences how you might go about implementing it.

### Project Submission

All project work should be backed up on your assigned external hard drive. Your final program and your report should be put into a single folder with your full name on it.