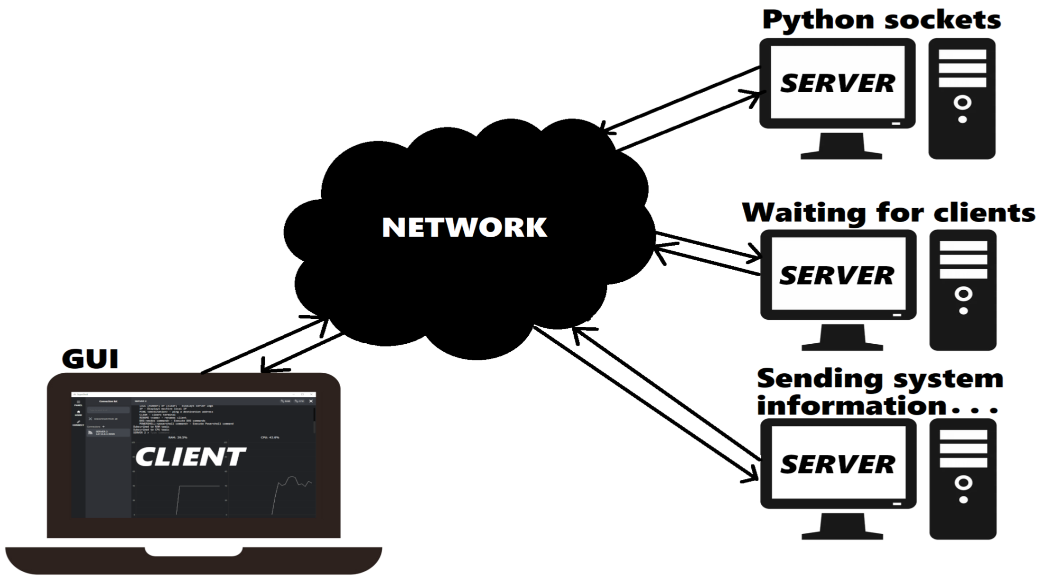
TECHNICAL DOCUMENT

In this document I will be explaining the technical side of this project. The base functionality of this project was designed using python sockets.

To begin with,
this is what the architecture of my project looks like:



Each server machine (on the right) will be running the server.py script.
One or multiple clients will be capable of connecting to it (with the server's IP and PORT) and each of them will be able to execute commands asynchronously.
The server DOES NOT have a graphical user interface, however the client does.
This is a great decision because that means that the server script can be much smaller in size and much less resource intensive to run for the server.
On top of that, it is not necessary for a server to have a graphical interface since there's no actual practical use for one.

HOW TO START A SERVER PROGRAMMATICALLY?

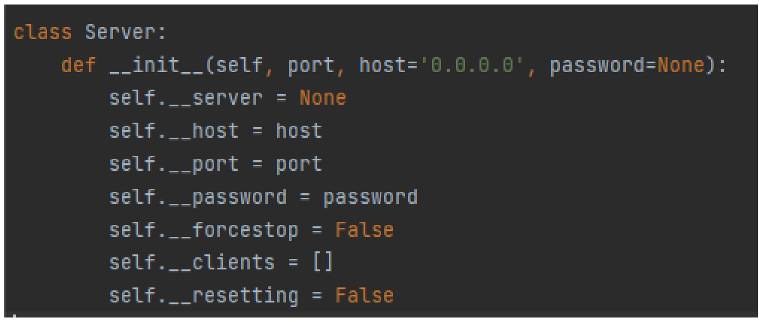
When you take a look at the contents of the server script you will find that there are two main classes:

A **Client** class and a **Server** class.

I decided to make the server into a class object because that would make all the methods and attributes easier to manage.

The server will create a **Client class object** for every connection that occurs therefore we can keep track of things such as whether this connection is **authenticated**, or **subscribed** to a topic (I will explain what a topic is later in this document).

On startup, create a Server class instance. It requires one positional argument (INT) which corresponds to the PORT on which the server will run.

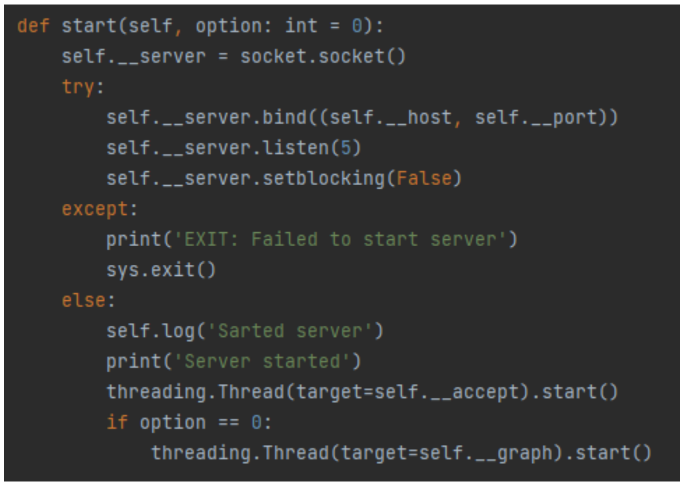


```python
class Server:
    def __init__(self, port, host='0.0.0.0', password=None):
        self.__server = None
        self.__host = host
        self.__port = port
        self.__password = password
        self.__forcestop = False
        self.__clients = []
        self.__resetting = False
```

The '**start()**' public method needs to be ran from this Server object which basically runs all the essential code to get the server up and running:

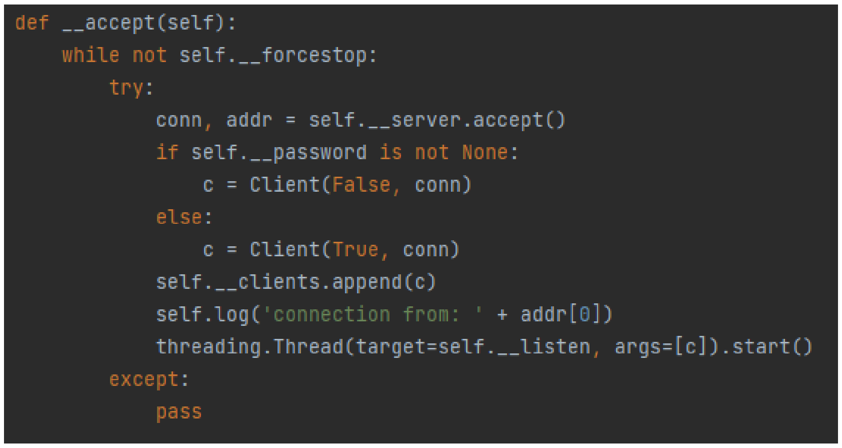
1) A python socket object is created using the HOST '**0.0.0.0'** by default which means that the server will accept connections from any network.

If the 'localhost' HOST address is used instead, the server will only work on local connections.
*If you want to use localhost use : Server.start(**PORT:int**, '**127.0.0.1**')*



```python
def start(self, option: int = 0):
    self.__server = socket.socket()
    try:
        self.__server.bind((self.__host, self.__port))
        self.__server.listen(5)
        self.__server.setblocking(False)
    except:
        print('EXIT: Failed to start server')
        sys.exit()
    else:
        self.log('Sarted server')
        print('Server started')
        threading.Thread(target=self.__accept).start()
        if option == 0:
            threading.Thread(target=self.__graph).start()
```

2) A thread is then created. This thread will be a never-ending loop (until the server is closed that is) from the class' __accept method:



```
def __accept(self):
    while not self.__forcestop:
        try:
            conn, addr = self.__server.accept()
            if self.__password is not None:
                c = Client(False, conn)
            else:
                c = Client(True, conn)
            self.__clients.append(c)
            self.log('connection from: ' + addr[0])
            threading.Thread(target=self.__listen, args=[c]).start()
        except:
            pass
```

IN DEPTH EXPLANATION:

While the object's __forcestop attribute is not set to True, this will run.
The attribute __forcestop makes it possible to manage the looped threads from outside.
When a connection occurs a client class instance is created using the socket connection as a parameter. The boolean variable is to specify whether the user is authenticated or not. If the server requires a password, the user will automatically be unauthenticated.
I'm using a python **try:** statement in order to ignore errors when a connection is not found, since I've set the blocking to False on my socket, (socket.setblocking(False)) this means that the server is no longer actually waiting for a connection, it is now polling for a connection. **If a connection could not be accepted, an error is raised.**
Unfortunately, this means we are now polling for connections.
The reason why I've decided to remove this blocking was because if we are listening to a client but the socket is closed, the thread will still be running since the server is still waiting for a message. This was a problem that could potentially make the server close improperly by leaving some unwanted threads running on the machine.
After a lot of testing, I found that this solution was far more stable after having more than 3 clients connected, than the latter using the socket blocking method.

A second thread gets created on startup and this is what I call the **topics**. They serve a purpose for being able to stream data continuously to the client who wishes to view the **CPU or RAM** usage graphs.
It is the most optimized solution I could think of, instead of having the client ask for new information every second, the client subscribes to the topic and the server recognizes this connection as a subbed connection and sends him the information. This solution removes a lot of useless requests from the client.
It is possible to disable this option on start-up by running the start method with an argument different from 0. Example:
Server.start(1)

Another thread gets created every time we accept a client's connection. It's the Server's __listen method.
The server will listen for every client's message asynchronously.
We first check if the client object is authenticated. If not, then we will not execute the commands sent by the client, instead we will verify if the message he sent corresponds to the server's password. If it does, we authenticate the Client.
At this level in the code, we look if the message decoded string corresponds to the **main commands** such as **KILL** or **RESET** or **DISCONNECT**.
We also handle the subscription to topics.
On **KILL,**
We completely terminate the server script.
On **RESET**,
We close the socket, close all connections, and then restart the socket. This allows us to flush the socket.
On **DISCONNECT,**
We disconnect the connection.
In order to subscribe or unsubscribe the client to topics, a message of this syntax must be received from the client connection:
<sub_cpu_true> or *<sub_cpu_false>*   *<sub_ram_true>* or *<sub_ram_false>*

HOW DO COMMANDS WORK?

Whenever we receive a message from a client we look if the message corresponds to the following list of commands:

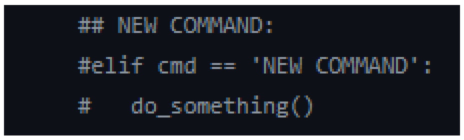
```
KILL
RESET
DISCONNECT,LEAVE,QUIT,EXIT
OS
NAME
RAM
CPU
LOGS
IP
PING
DOS:
POWERSHELL:
LINUX:
EXEC:
```

The commands such as **DOS:** , **POWERSHELL:** , **LINUX:** , **EXEC:** are particularly interesting because they allow the client to execute system commands on the machine running the server program.
In order to program such a thing, we use the subprocess python module. Example:

```
subprocess.Popen(CMD TO RUN, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
encoding='utf-8', shell=True)
```

HOW DO I ADD MY OWN COMMANDS?

You can add your own commands easily. You must first choose the operating system that you wish to create the command for. If you want to create a Linux specific command, find the **linux.py** file in the commands directory.
For windows, the **windows.py** file. For other operating systems, the **other.py** file.
Once you have chosen, open the script and look for the following lines (towards the end of the script):



```
## NEW COMMAND:
#elif cmd == 'NEW COMMAND':
#   do_something()
```

What must be done is pretty straight forward,
remove the comments on the lines, replace ‘NEW COMMAND’ with the name of the command that you wish to create and put whichever code you want.

If ever you encounter any issues or have any questions, feel free to write me at my email address : *pro.raphaelromeo1@gmail.com*